From the vault of my engineering newsletter
"Systems That Scale"

Saurav Prateek's

Replication in Distributed Systems

Understanding how **Replicated** state machines support **Fault Tolerance** and **Availability**

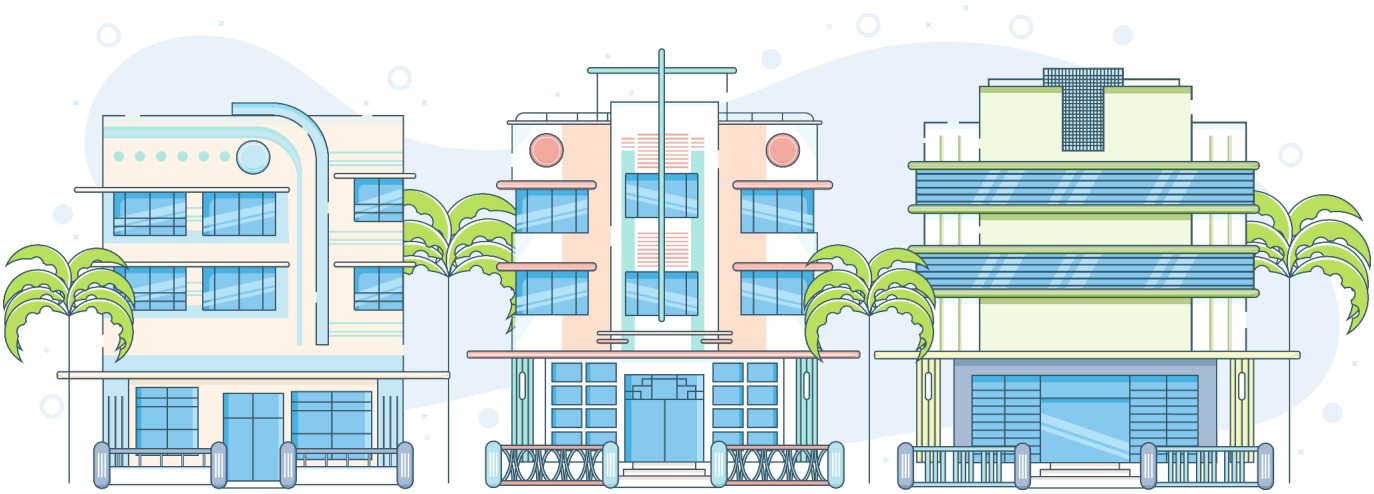


Table of Contents

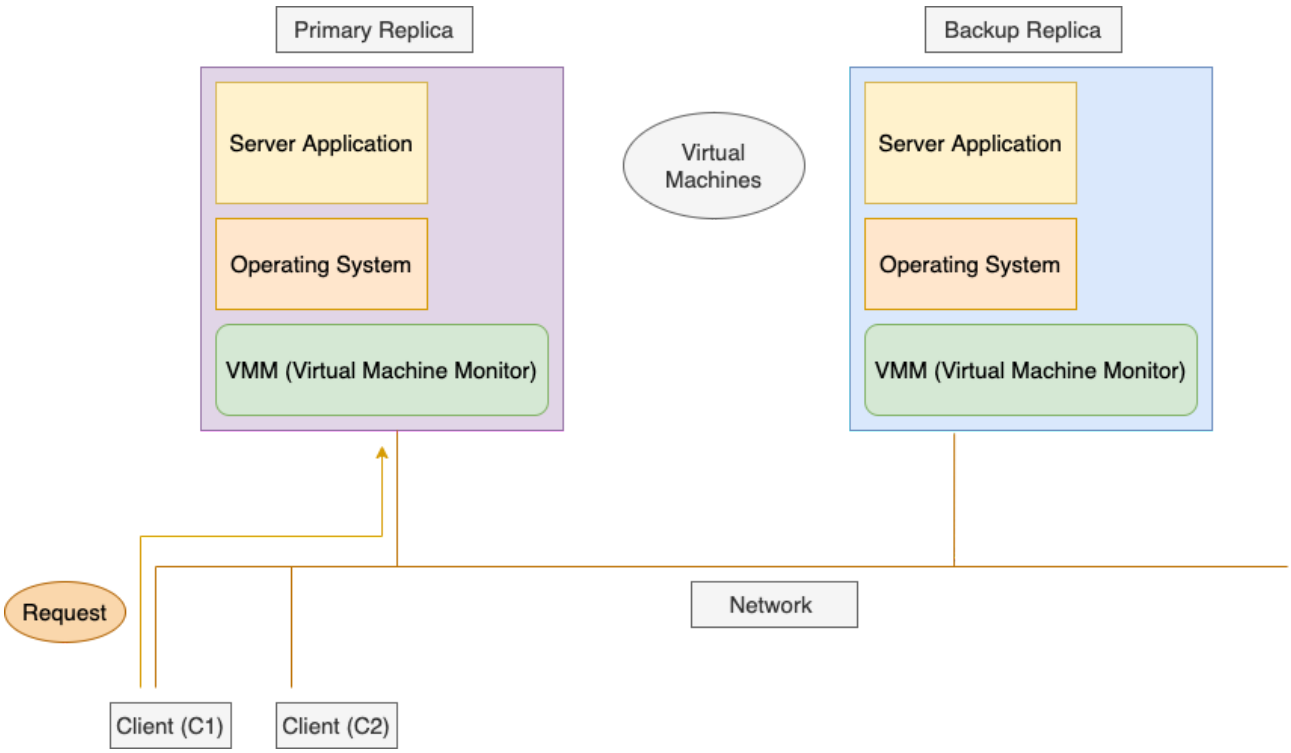
Chapter 1

Replicated State Machines

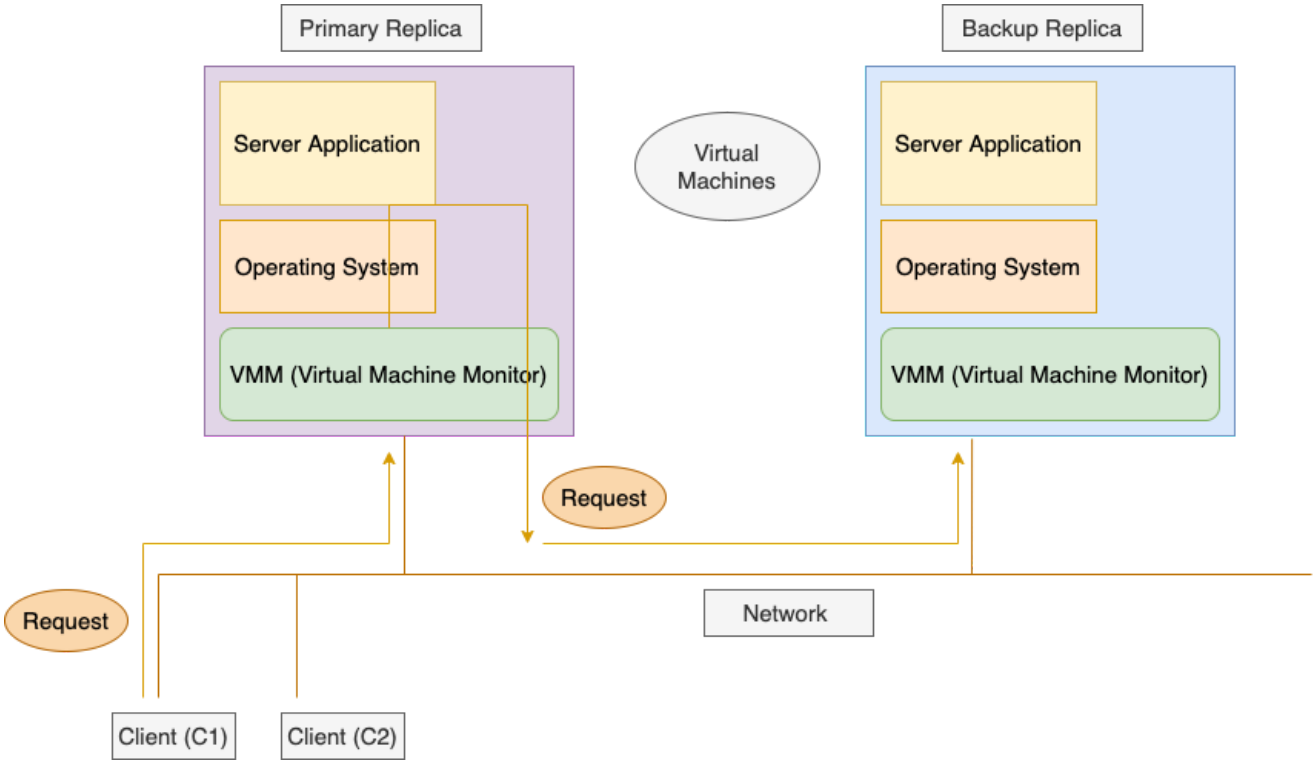
The chapter discusses the **Replicated State Machines** and how it ensures **Fault Tolerance** and High **Availability** in our services with the Distributed Systems. It also discusses "**The Output Rule**" used by this scheme to ensure fault tolerance.

Introduction

The **Replicated State Machines** is a scheme to perform data replication in order to ensure high availability across our system. It is a generalised approach for building **Fault Tolerant** services with the Distributed Systems. This scheme provides Fault Tolerance. They are specifically tolerant to those failures which could cause the server/machine to stop working such as Network Failures or Hardware Failures. These failures are also known as **Fail-Stop** errors.

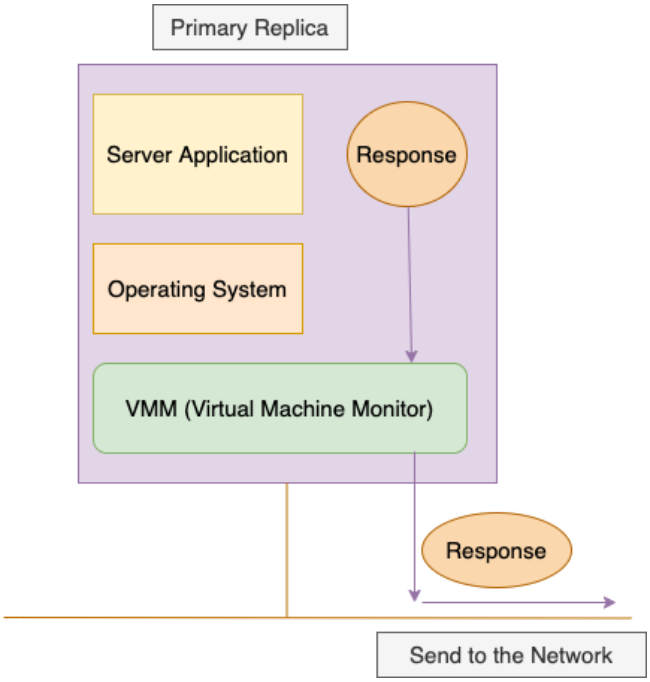


In the above architecture there are two machines. These are **Primary** and **Backup** replicas. The client sends the request (packets) to the Primary replica (machine). The request packet goes to the **Virtual Machine Monitor** (**VMM**). On arrival the VMM generates a network packet arrival interrupt to the operating system. In addition VMM knows that the request is an input to the Primary replica and hence it sends the copy of the request packet to the Backup Virtual Machine Monitor.

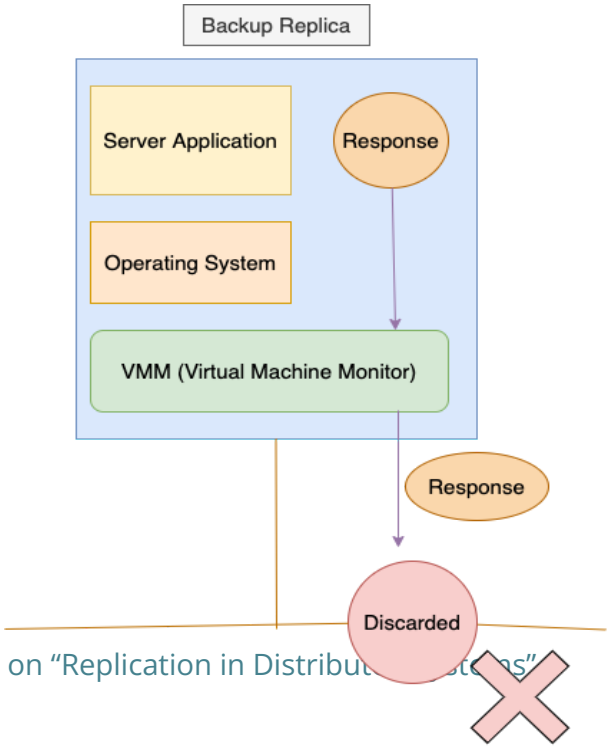


The **Backup** replica's **VMM** receives the packet from the Primary Machine. The backup machine then also fakes a network interrupt the way Primary Machine did earlier. Since both the Primary and the Backup machine gets the same request packet and also processes them in the same way, hence they stay synchronised.

Once the Primary replica receives the packet, it starts processing the request and then generates a reply/response packet and further sends it to its Virtual Machine Monitor. The VMM sees the response packet sent by the Primary replica and then sends the reply packet over the network back to the client.



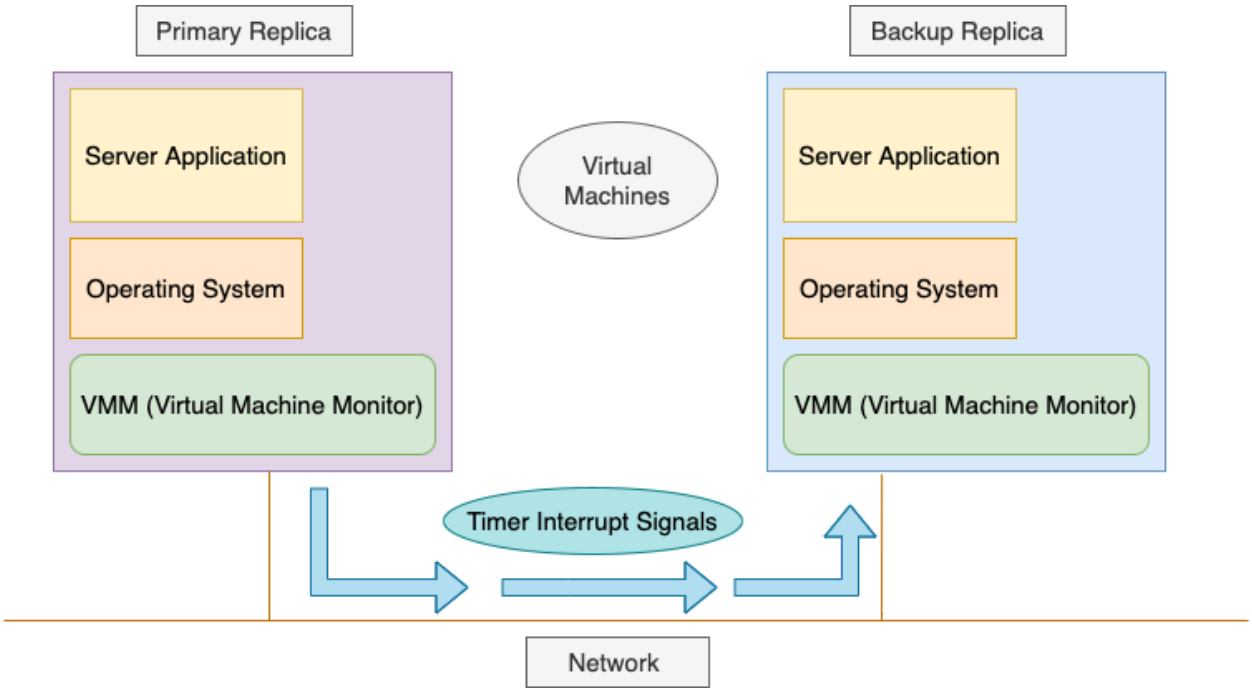
Since, the **Backup** replica (machine) is executing the same set of instructions, it also generates the Response packet and sends the packet to its **VMM**. Once the VMM receives the response packet, it knows that the response is generated from the Backup replica machine and it further drops the response packet. It doesn't allow the response packet to be sent over to the network back to the Client. Since, only the Primary replica machine is allowed to send the packets over to the network.



Fault Tolerance

In our previous sections we observed that every time the **Primary** replica (machine) receives a request packet, it forwards that to the **Backup** replica (machine) as well. Apart from this the Primary replica also keeps sending many timer interrupt signals regularly to the Backup replica. Hence, there is a continuous communication between the Primary and the Backup replica machines.

Hence, if the Primary Machine crashes then the Backup machine will stop receiving interrupt signals from it and will soon know that the Primary replica has crashed and stopped functioning due to some unfortunate reasons.



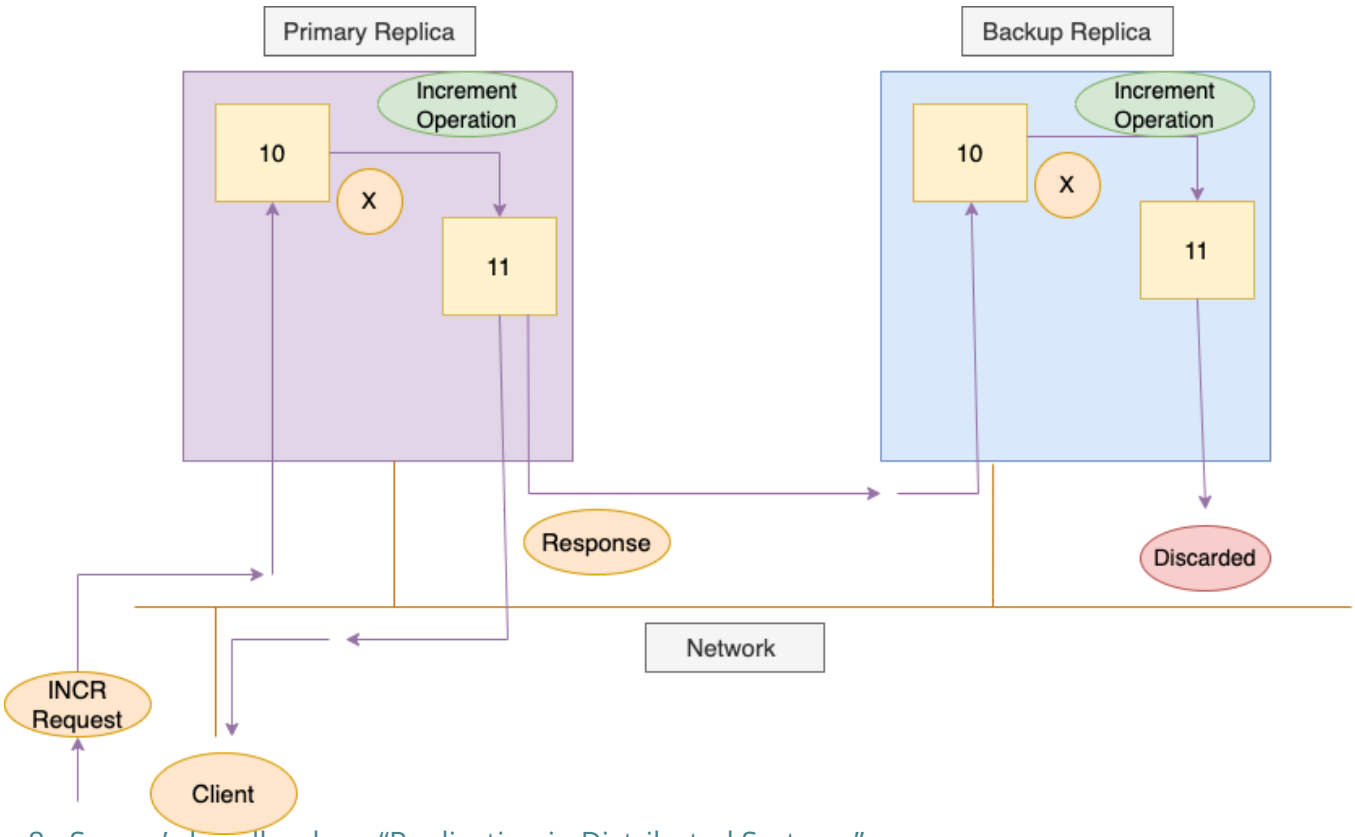
In that case the Backup Replica goes **Live**. This means it stops waiting for the Primary replica to send the event signals over the channel to itself. The VMM (Virtual Machine Monitor) present in the Backup replica now allows it to execute freely. This means that the VMM of the Backup replica machine does two things:

1. Allows the future Client's **request** packets to be sent directly to the Backup Replica machine.
2. It no longer **discards/drops** the response/reply packets generated by the Backup Replica machine. It allows the replica machine to send the response packets further on the network, back to the Client.

In the same way if the **Backup** replica machine fails then the Primary replica will stop sending the event signals to the Backup replica and will function as a **single non-replicated** server.

The Output Rule

Suppose we are running a simple **Database Server** over the **Replicated State Machines**. We have a data-item, say **X** and the operation that our database server supports is an **increment** operation. The Client sends the increment request and the Database Server increments the data-item's value and sends back the new value to the Client.

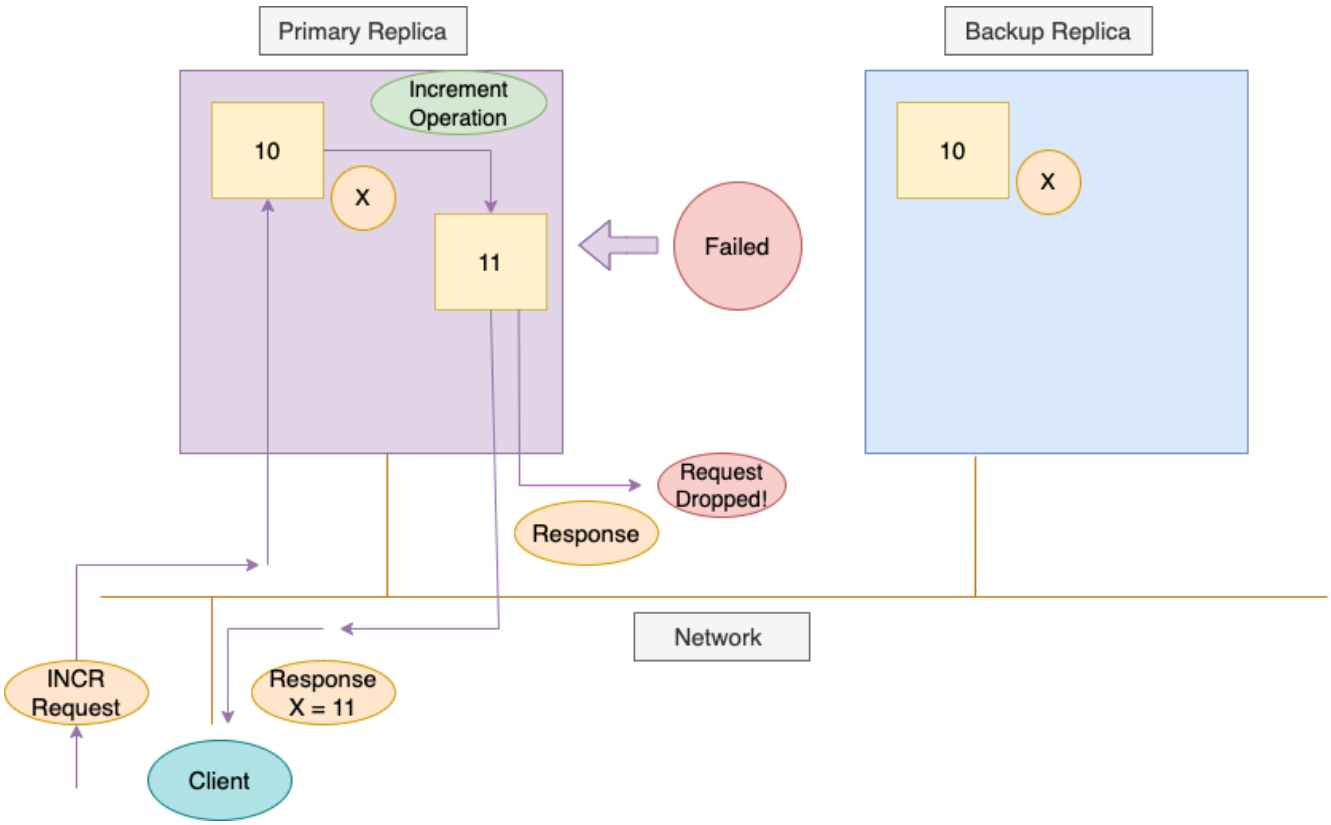


8 - Saurav's handbook on "Replication in Distributed Systems"

In the above architecture the current value of the data-item **X** is **10** in the memory of both the Primary and the Backup replica machines.

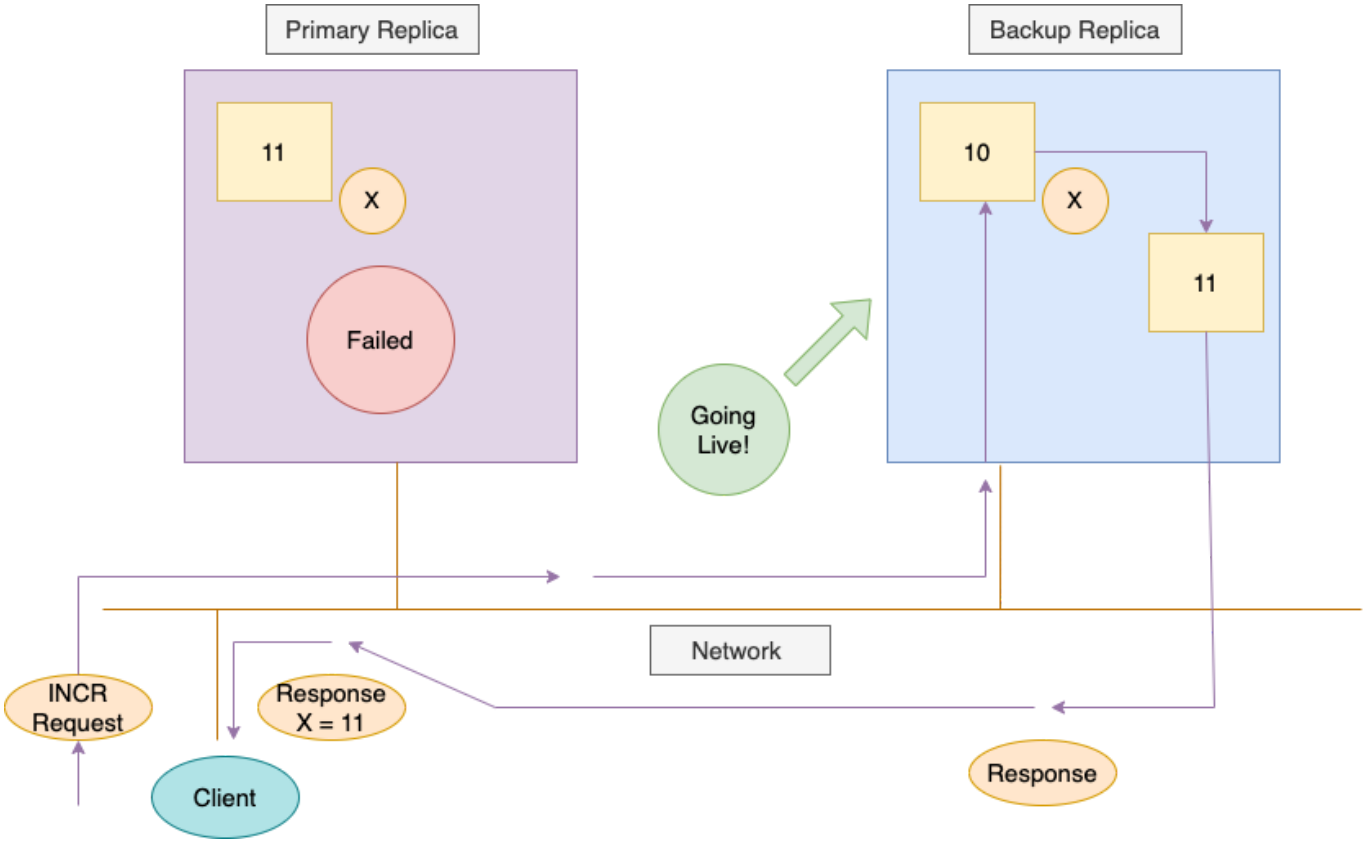
The Client **C** sends an **INCR** (Increment) request to the Primary replica. The Primary machine receives and executes the operation and increments the value of **X** to **11**. Then it sends the newly updated value back to the Client. The same request is also supposed to be sent to the Backup replica from the Primary machine. The Backup machine will also update the data-item X value and send the response which will further be discarded by its **VMM** (Virtual Machine Monitor).

Suppose the Primary machine dies after sending the response packet to the Client and before sending the copy of the request packet to the Backup replica machine.



The Client will receive the updated value (**11**) of the data-item **X**. But the Backup Replica still has the older copy (**10**) of the data-item **X**.

Now, the Backup replica will go Live since the Primary replica has died. The new client will send an increment request to this Backup replica and the replica will now change the value of **X** to **11** and return the incremented value as a response back to the Client.



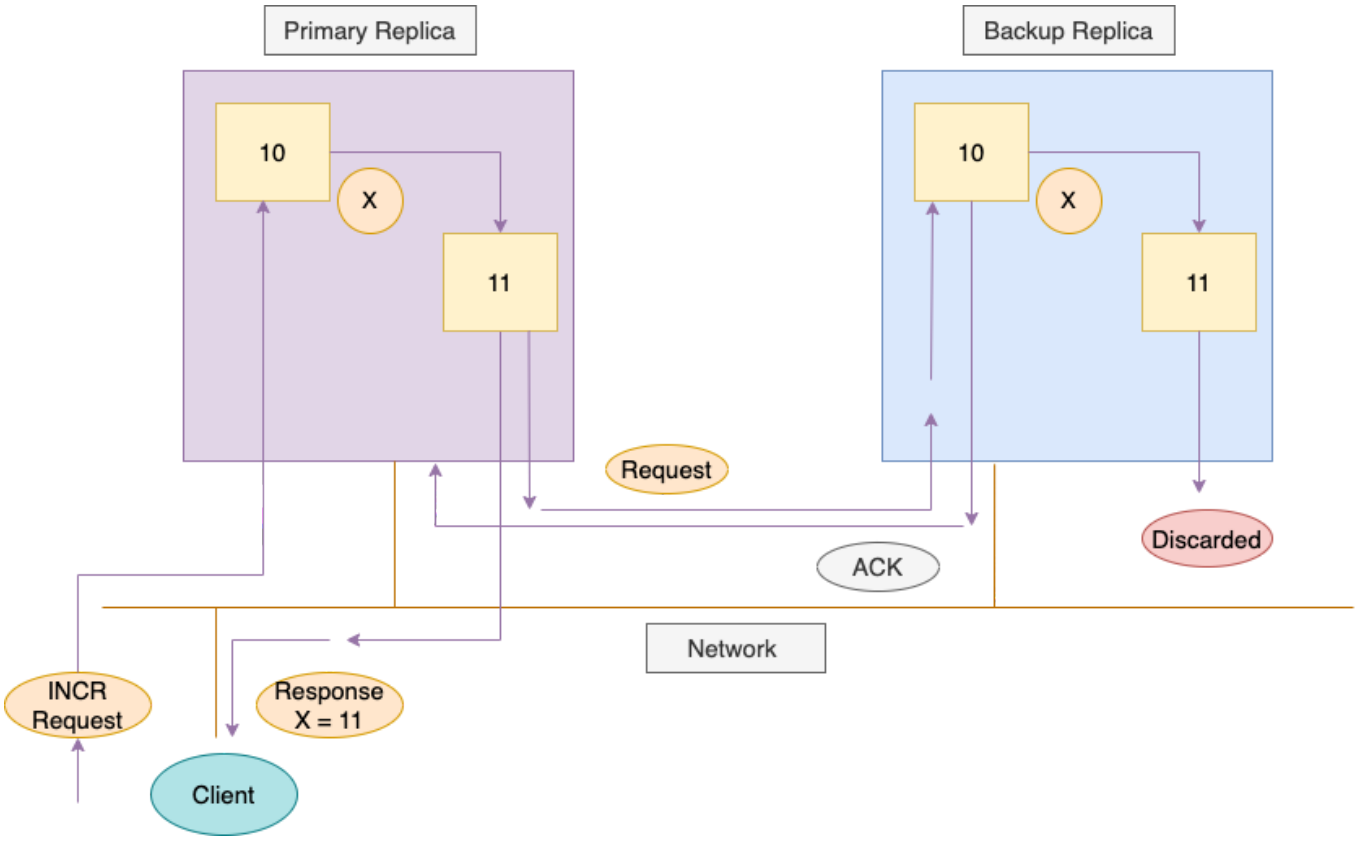
This can lead to **Data inconsistency** in our **Database servers** which can further lead to huge problems.

The way the **Replicated State Machine** avoids this Problem is through **The Output Rule**. The Output Rule states that:

> *"The **Primary** replica is not allowed to send any output response packet to the Client until the **Backup** replica acknowledges that it has received all the event records till that point"*

Hence, in the previous scenario even if the Primary Replica generates the Response packet having incremented value (**11**) of the data-item **X**. It is not allowed to send it back to the Client till it receives the **acknowledgement** from the **Backup** server that it has also received the increment request packet.

This will avoid the existence of **Data-inconsistency** in the Database server.



Chapter 2

Single Leader Replication

The chapter discusses **Replication** in detail and its multiple schemes. It also discusses the **Single Leader Replication** scheme, **Synchronous** and **Asynchronous** replication processes along with the **Fault Tolerant** schemes.

Replication

In general terms **Replication** means distributing multiple copies of your data on multiple machines. These machines are connected to each other via network. There can be multiple reasons to replicate the data over these machines:

1. To keep the data **geographically** closer to the Client. We can replicate our data on multiple servers and these servers can be geographically distributed. In this way clients can request the data from the server which is closest to them.

2. To avoid **Single Point of Failure**. We distribute the same copy of data to multiple machines, so that even failure of some of these machines won't lead to the loss of our data.

3. To offer **Scalability**. Some systems are read-heavy and distributing our data to multiple machines can allow all of them to take up the read requests from the Clients. This distributes the load and helps the system to scale.

These are some obvious reasons to introduce **Replication** into the system. If the data being replicated doesn't change over time then the entire replication process becomes fairly simple. We just need to keep copying the data to multiple machines and scale up the Read requests. The real challenge comes up when the data being replicated can change over time. This change of data in one machine will demand the change to be replicated to all the other machines that hold the copy of the similar data.

Multiple Replication algorithms are used in order to handle the changes in the replicated data. These are:

- Single Leader Replication algorithm
- Multiple Leaders Replication algorithm
- Leaderless Replication algorithm

In this edition we will talk about the **Single Leader Replication** algorithm.

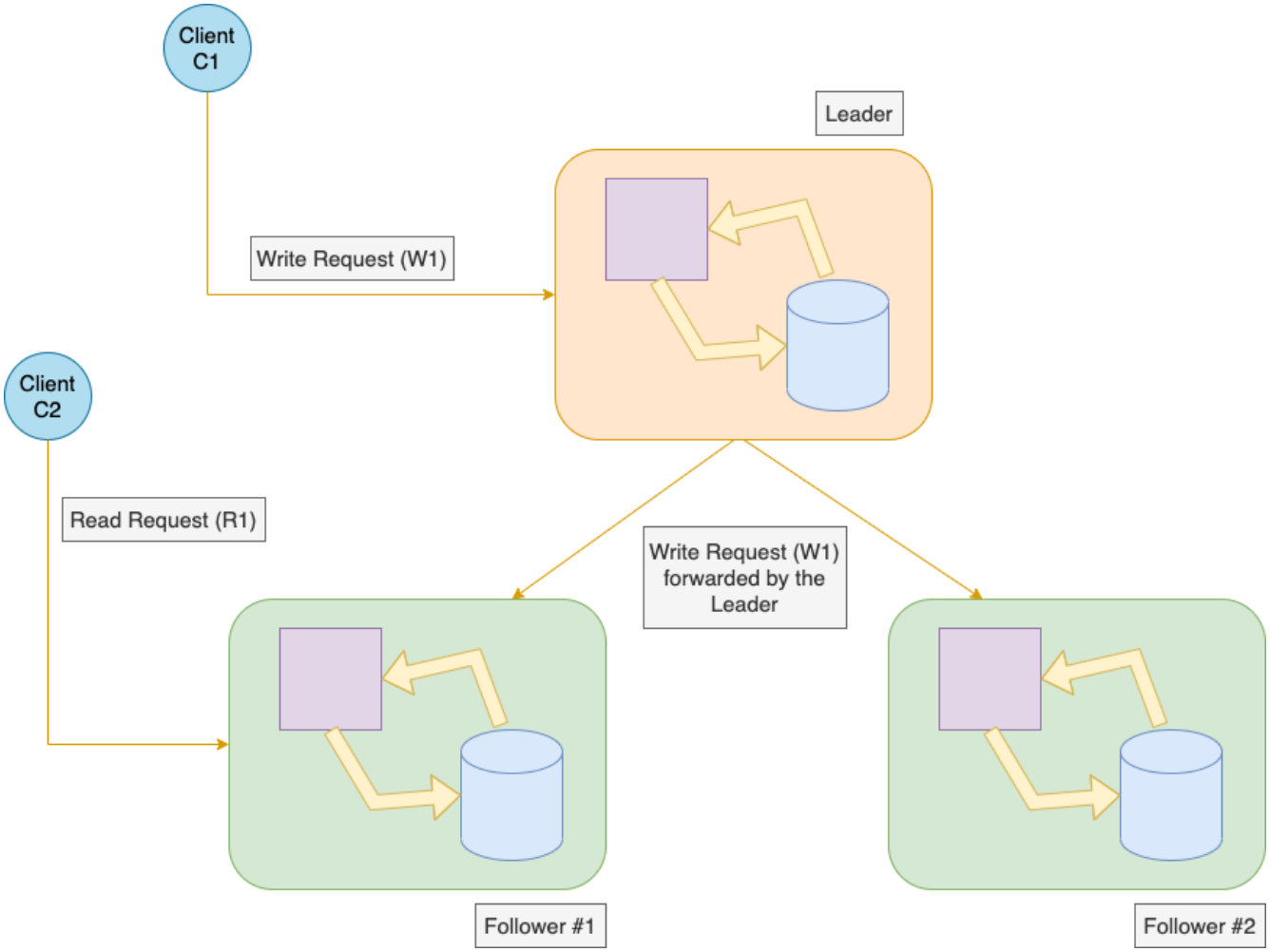
Single Leader Replication

Under this scheme we have a single Leader and all the other replica machines are termed as **Followers**.

Let's first understand what **Leader** and **Followers** actually are. Suppose we have a Distributed System having multiple machines connected over a network. All the machines store a copy of the database. They are also known as **Replicas**. Among these replicas, one of them is elected as a Leader. So Leader is a machine which accepts all the Write requests from the Clients. Whenever any client wants to make any changes in the database (update/delete) then it sends the Write request to the Leader replica.

Now the Leader receives the Write request and makes the required changes in its copy of the Database. Once the changes are made, the Leader sends the Write request to the rest of the Followers. These followers then update their copy of the data. All the Followers perform the write requests in the same order in which those requests were executed by the Leader.

When a Client needs to perform a **Read** request, it can freely send its request to any replica machines, be it a Leader or the Followers.



Synchronous and Asynchronous Replication process

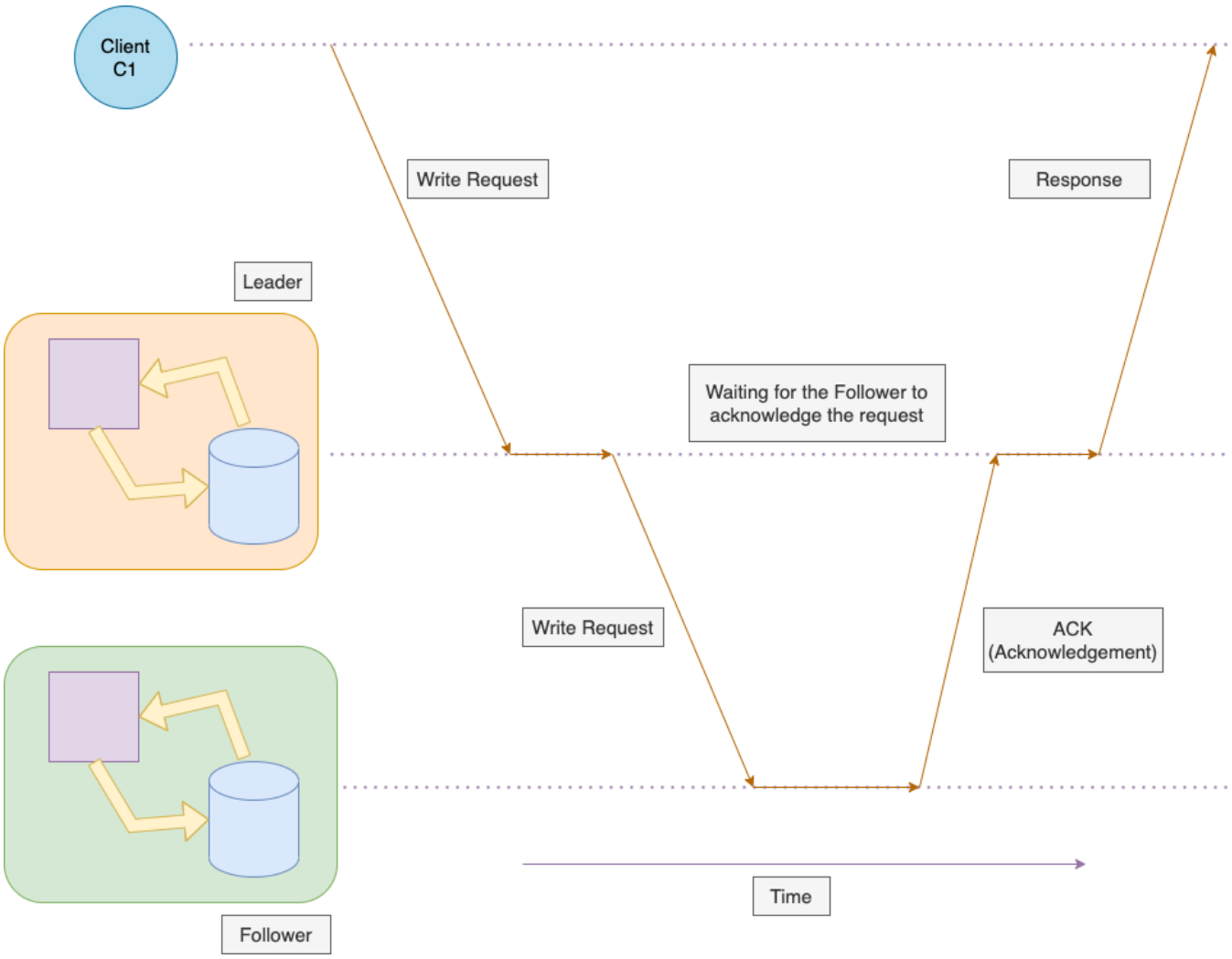
There are two ways in which the Replication can take place in the systems. It can happen synchronously or asynchronously.

Synchronous Replication

Let's take the previous example where the Leader received an incoming Write request from the Client. Afterwards it forwarded the Write request to all the Followers. Now in the Synchronous Replication process the Leader will wait for the followers to confirm that it has successfully received the incoming Client's request.

Only after a confirmation from the Follower, the Leader will send a response back to the Client.

Let's say there's an architecture having one Leader (**L**) and one Follower (**F**). The Synchronous replication scheme will look something like this.



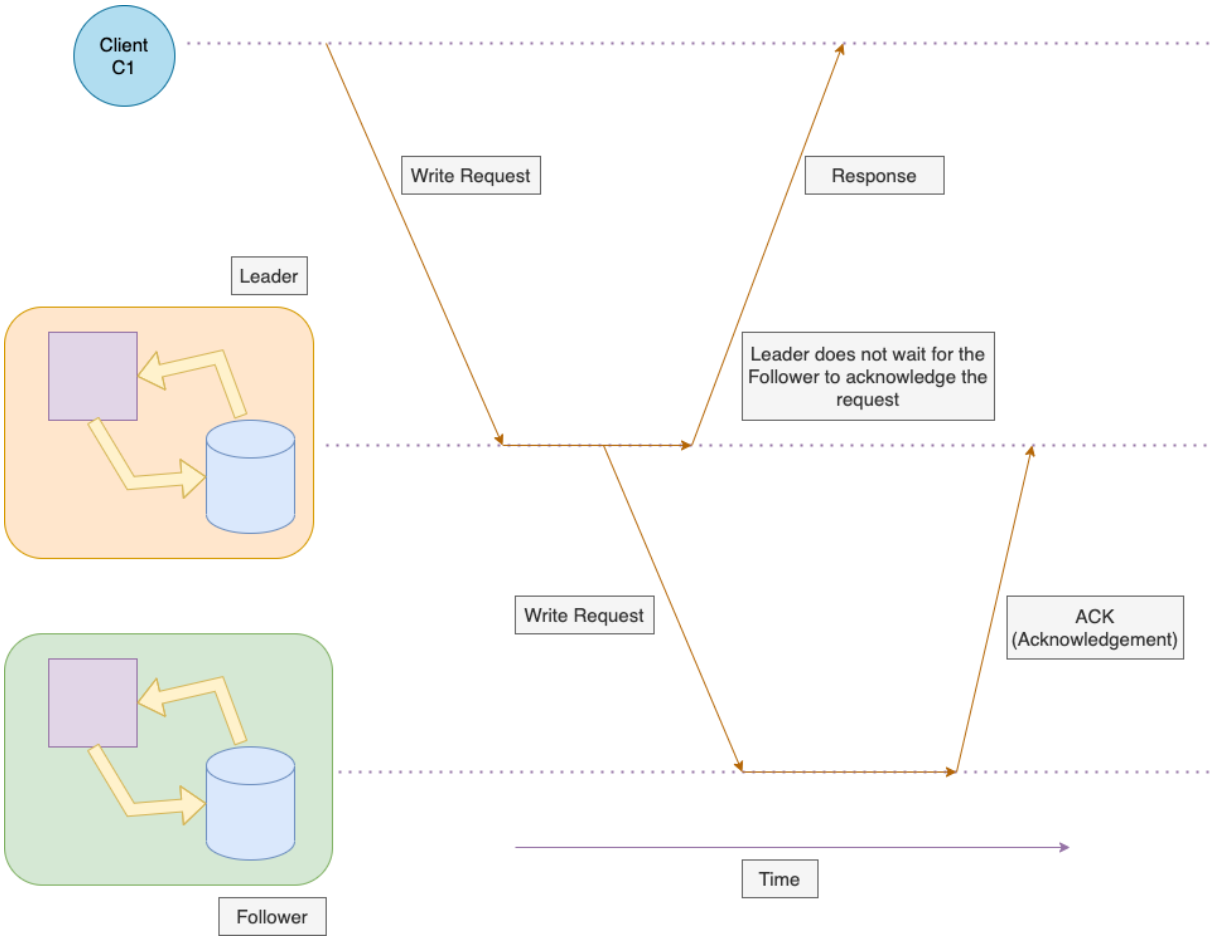
One of the major advantages of **Synchronous Replication** is that there is a very less chance of **Data Corruption** or **Data Inconsistency**. The follower is guaranteed to have the up-to-date copy of the data. Hence, if the leader dies, any of the Followers can simply replace it to become a new Leader.

One of the major disadvantages of this scheme is the high chance of the system to fail. Even if one of the Synchronous Followers stops responding, then the Leader will keep waiting forever for the Follower to respond and this could cause the system to halt.

Asynchronous Replication

Once the Leader receives the Write request from the Client, it will forward it to all the Follower replicas. After forwarding the request to the Followers, it won't wait for them to acknowledge the receipt of the request. This is known as the Asynchronous Replication process.

Let's take another similar example of a Distributed System having a Leader (**L**) and a Follower (**F**). The Asynchronous Replication scheme will look somewhat like this.



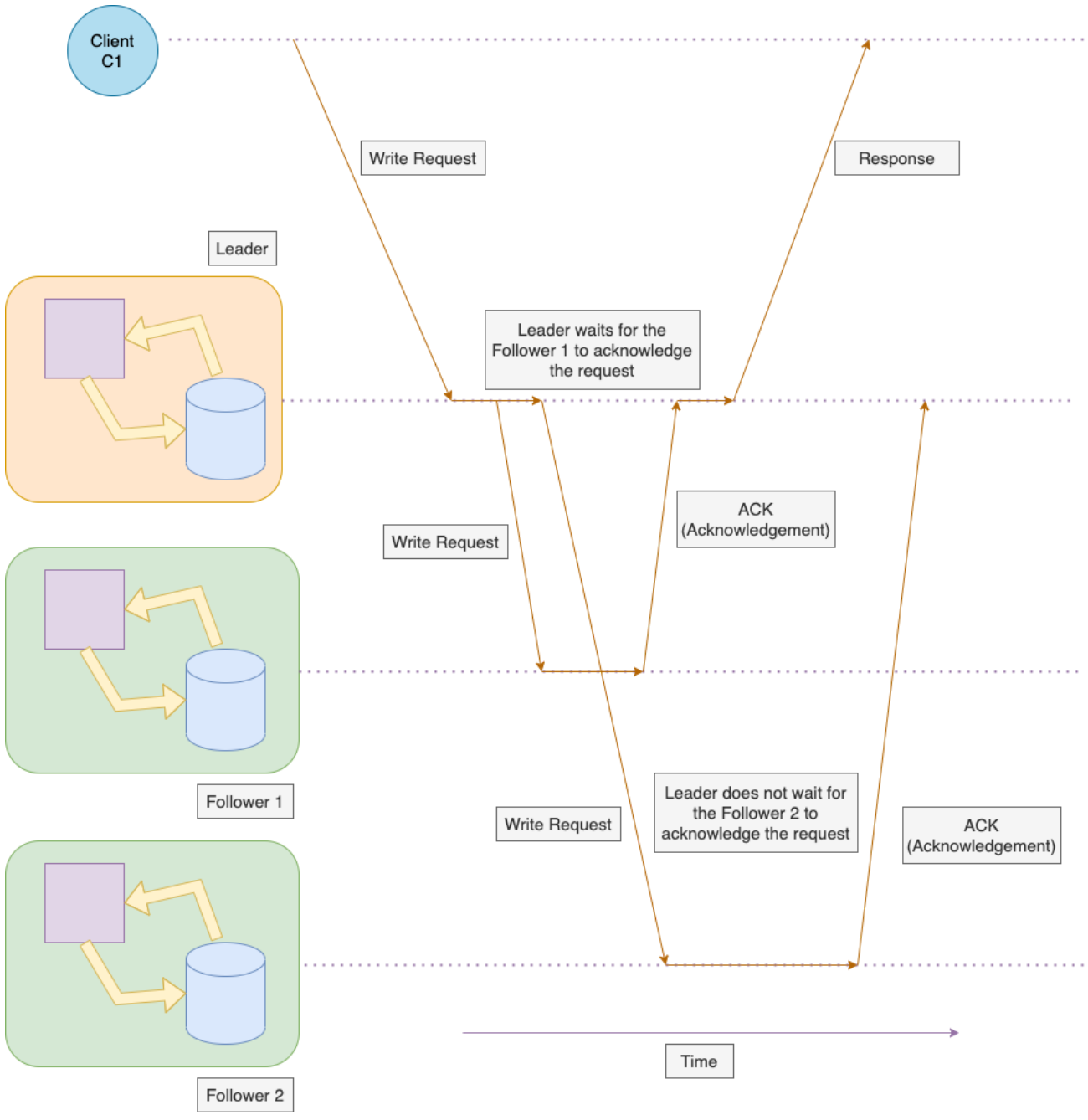
17 - Saurav's handbook on "Replication in Distributed Systems"

Since the Leader does not wait for the Followers to acknowledge the Write requests, hence it is not guaranteed that the Followers will have an up-to-date copy of the data. There can be Data Inconsistency in the system. But the failure of any Follower won't lead to the failure of the entire system in the asynchronous replication scheme. Even if the followers stop functioning, the system can still continue its execution.

Semi-Synchronous Configuration

In this configuration, one follower is **Synchronous** while the rest of the other followers are **Asynchronous**. Since only one **Follower** is made synchronous, hence it reduces the chance of system failure due to the failure of the Follower. If the synchronous follower stops functioning then one of the asynchronous followers is made synchronous. This ensures that one of the synchronous followers and the leader has an up-to-date copy of the data.

This configuration reduces the chance of data-loss, since we have up-to-date copy of data in at least two machines.



Fault Tolerance

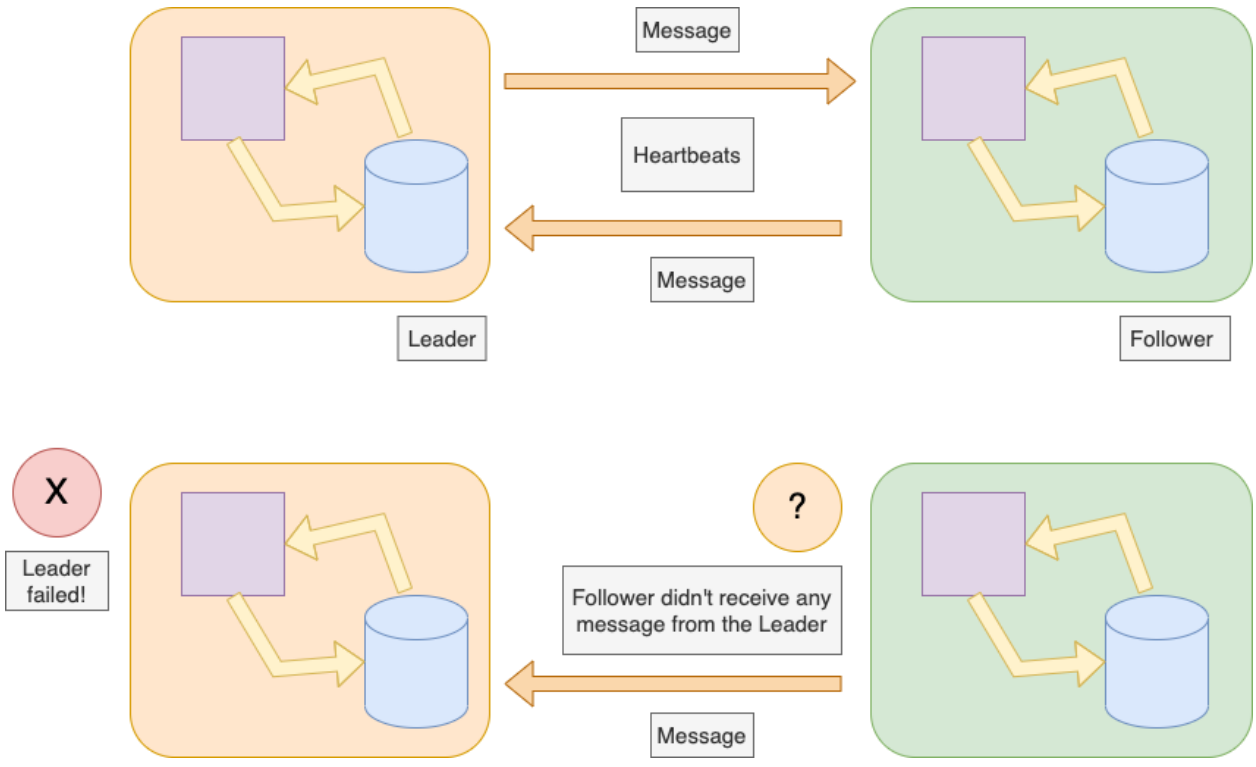
Since we are discussing Distributed Systems having multiple machines, there is a huge chance for some of these machines to fail. We need a solid fault tolerant

scheme to handle these replica failures. Ideally this can happen in two ways: either the Leader can fail or one of the Follower replicas might fail.

Case 1: Leader Failure

When a Leader in the system fails then a process is needed to be followed to handle the failure. This process is known as **Failover**. The Failover involves three stages:

1. **Stage 1**: In this stage, the rest of the **Followers** in the system detects the **Leader** has failed. This is achieved by the process of Heartbeat. All the replicas in the system keep on exchanging messages back and forth between each other. These messages are also termed as Heartbeats. So, when one of the nodes dies then the others stop receiving messages from its end and they detect that the node has failed. There is a certain period of time till which a node waits for the message from the other node. If it doesn't receive anything till that point of time then it considers that node to be dead.



2. **Stage 2**: Once the Leader has been declared dead, now the system needs to appoint a new Leader. The new Leader can be appointed by the process of **Election**. The replica receiving the majority of votes is elected as a new Leader. Ideally the best candidate for a Leader is the replica with the most up-to-date data changes.

3. **Stage 3**: Once the new Leader has been appointed, now all the clients must send their write requests to the new Leader. Also if the old Leader comes back, it must not act as a new Leader. It should know that a new leader has been already appointed and it should become a Follower.

Case 2: Follower Failure

Handling a Follower failure is less complicated as compared to the Leader Failure.

Every follower maintains a Log of data changes received from the Leader on its Local Disk. In future if the follower dies or gets disconnected from the network then it can recover back easily with the help of its Log. The follower once joins back the network can again connect with the Leader and request all the data changes after the last transaction that was processed in its Log. Hence it can receive all the data changes that happened during the time when the follower was disconnected.

Chapter 3

Multi-Leader Replication

The chapter discusses the concept of the **Multi-Leader Replication** scheme in detail. It also discusses multiple Use Cases where the use of a Multi-Leader replication scheme can be justified. It also introduces the problem of **Conflict Resolution** which is faced by this replication scheme and steps to resolve the **Write Conflicts**.

Introduction

We discussed **Single-Leader Replication** in our previous chapter. In the Single Leader Replication scheme we had a single leader and all the Write requests made by the clients must go through that leader. A major drawback with the previous approach was that we had a single dependency. If the leader was down or if there is a network interruption between us and the leader then we won't be able to send our requests to the system at all.

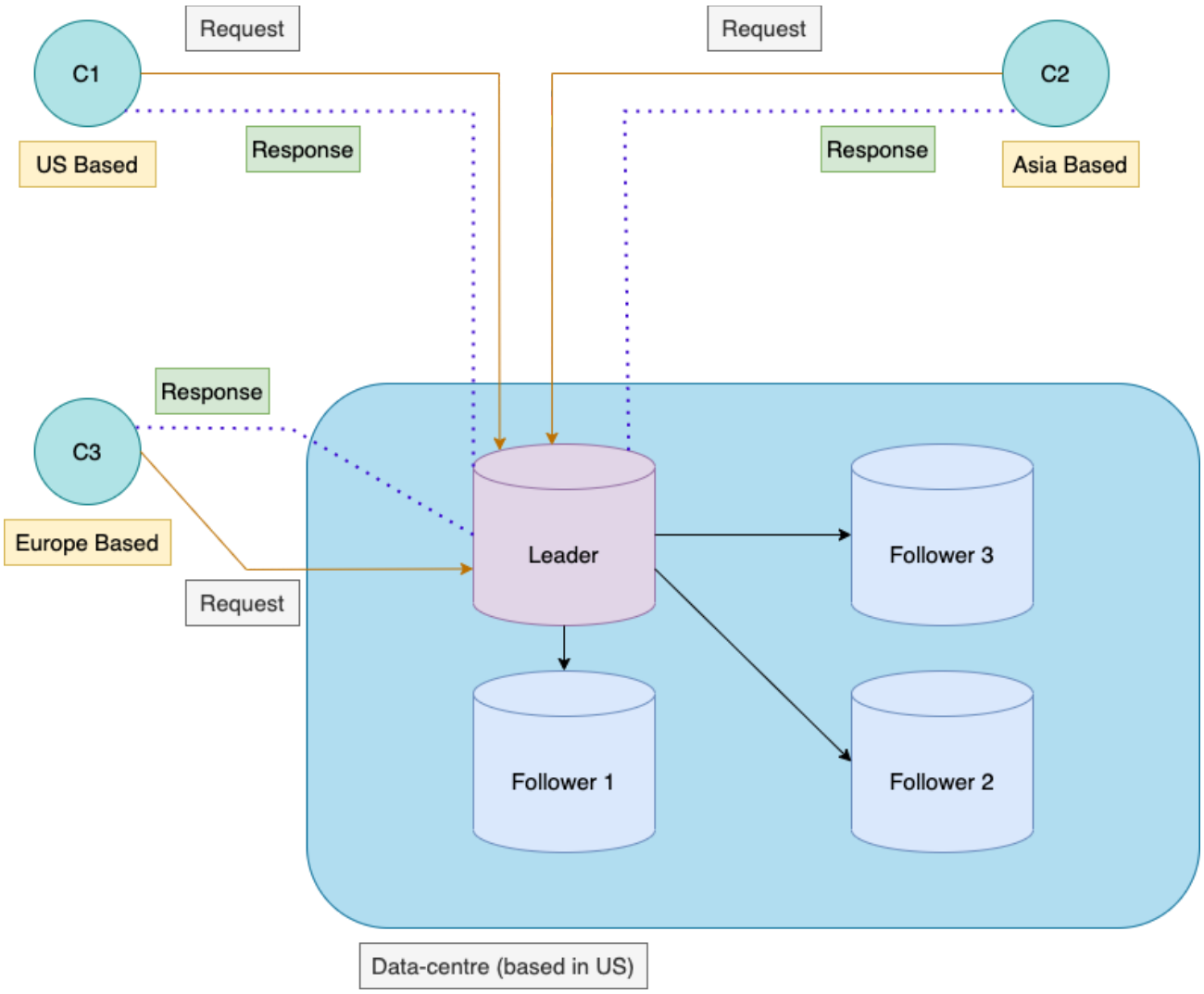
Here we can introduce multiple leaders in our system that can take requests from the Clients. In that case if a client somehow cannot reach Leader A then he might connect with Leader B to process his requests. This is a **Multi-Leader Replication** scheme. In this scheme every leader sends the update to all the other nodes present in the system.

"Each Leader also act as a Follower to other Leaders"

There are multiple places where using a Multi-Leader replication scheme can make sense. Let's discuss them one by one.

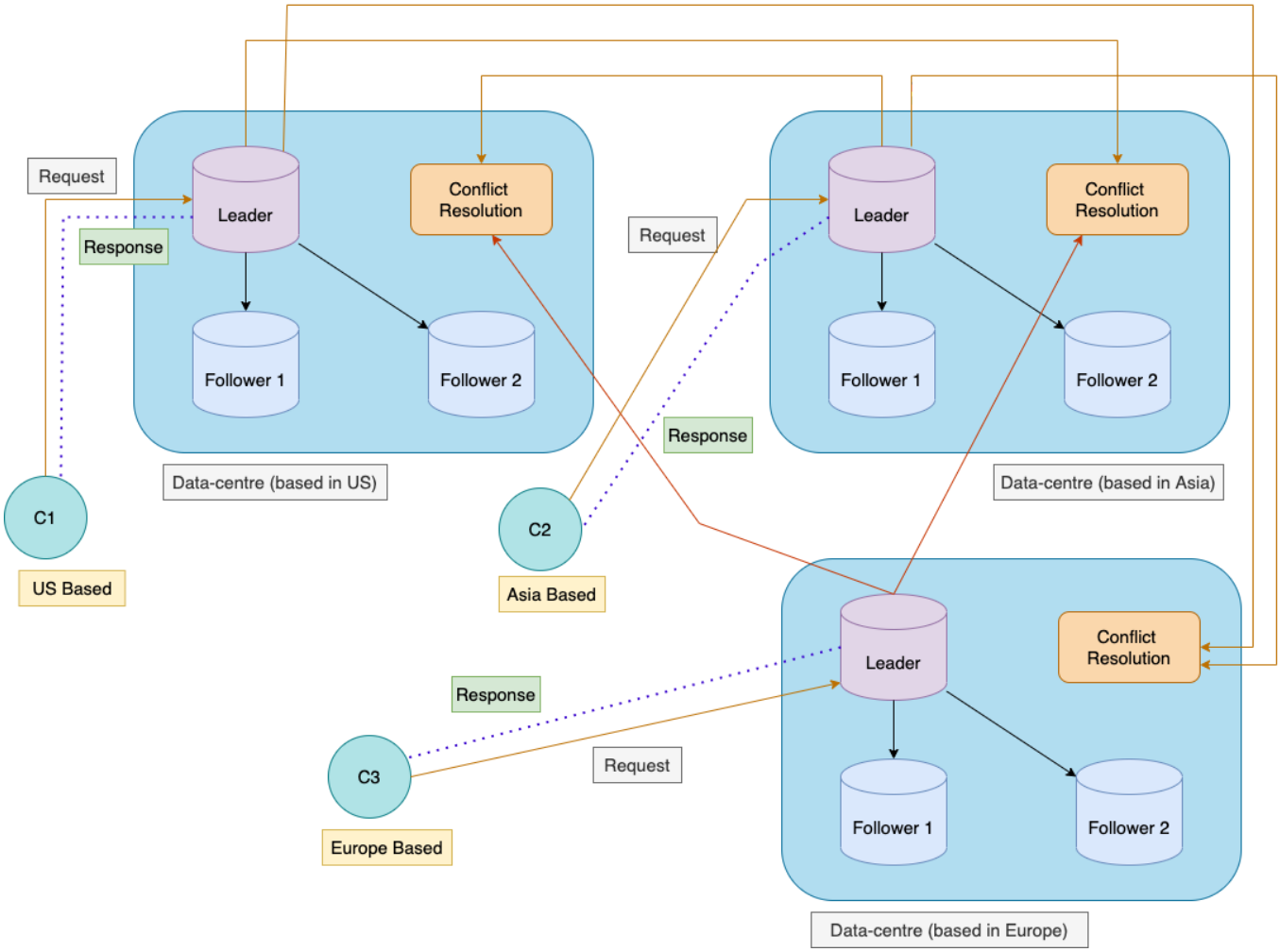
Use Case 1: Systems with Multiple Data Centres

Suppose you have a big system and you are planning to serve the users world wide. But currently your organisation is based in Silicon Valley, California and you have a single data-centre there. Your clients from other continents (like Asia and Europe) might face network delays while connecting to the server or while getting their requests processed.



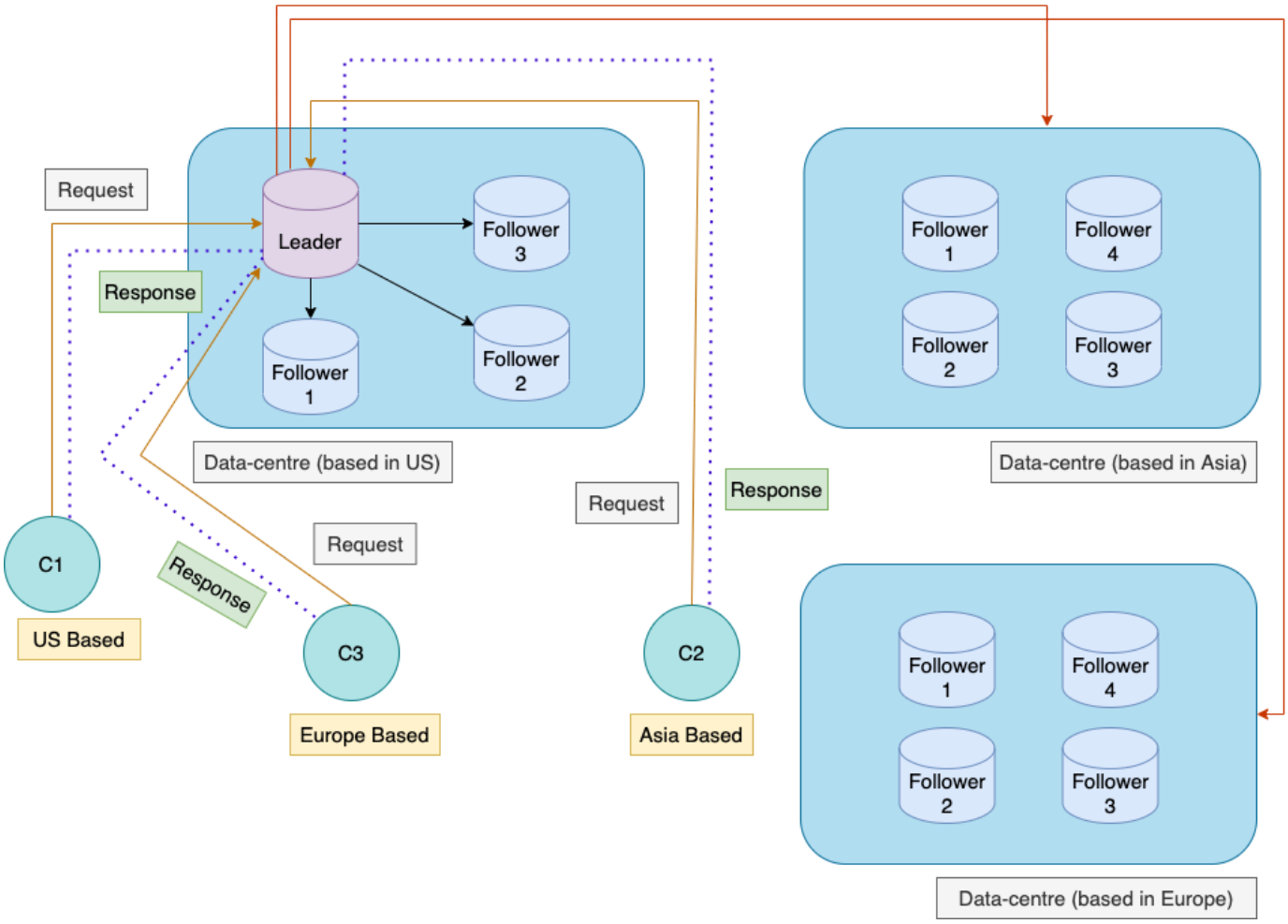
In the above configuration of **Single-Leader** configuration we can observe that three clients from US, Asia and Europe send their requests to the Single Leader present in the Data-centre based in the USA.

Now, you plan to scale your system and hence start building data-centres in Asia and Europe as well. These data-centres hold the replicas of the same database. Every data-centre will have its own Leader and the clients can now connect to the Leader present in the closest data-centre to them. The architecture might look like this.



In the above architecture we can observe that every Leader forwards the changes made to all the other Leaders residing in different data-centres.

Now, if we had a **Single-Leader** replication scheme in the multiple data-centre configuration then there would be only one Leader residing in one of the data-centre. Let's take the example of our previous architecture. Suppose we have a Single-Leader Replication scheme and one Leader is present in the US based data-centre. Now the clients from Asia and Europe will also need to send their requests all the way to the Leader present in the US based data-centre. This will introduce extra latency and contravene the purpose of having multiple data centres in the first place.



Even though we had data-centres based in Asia and Europe, still clients were sending their requests all the way to the US data-centre without leveraging the benefit of the presence of local data-centres.

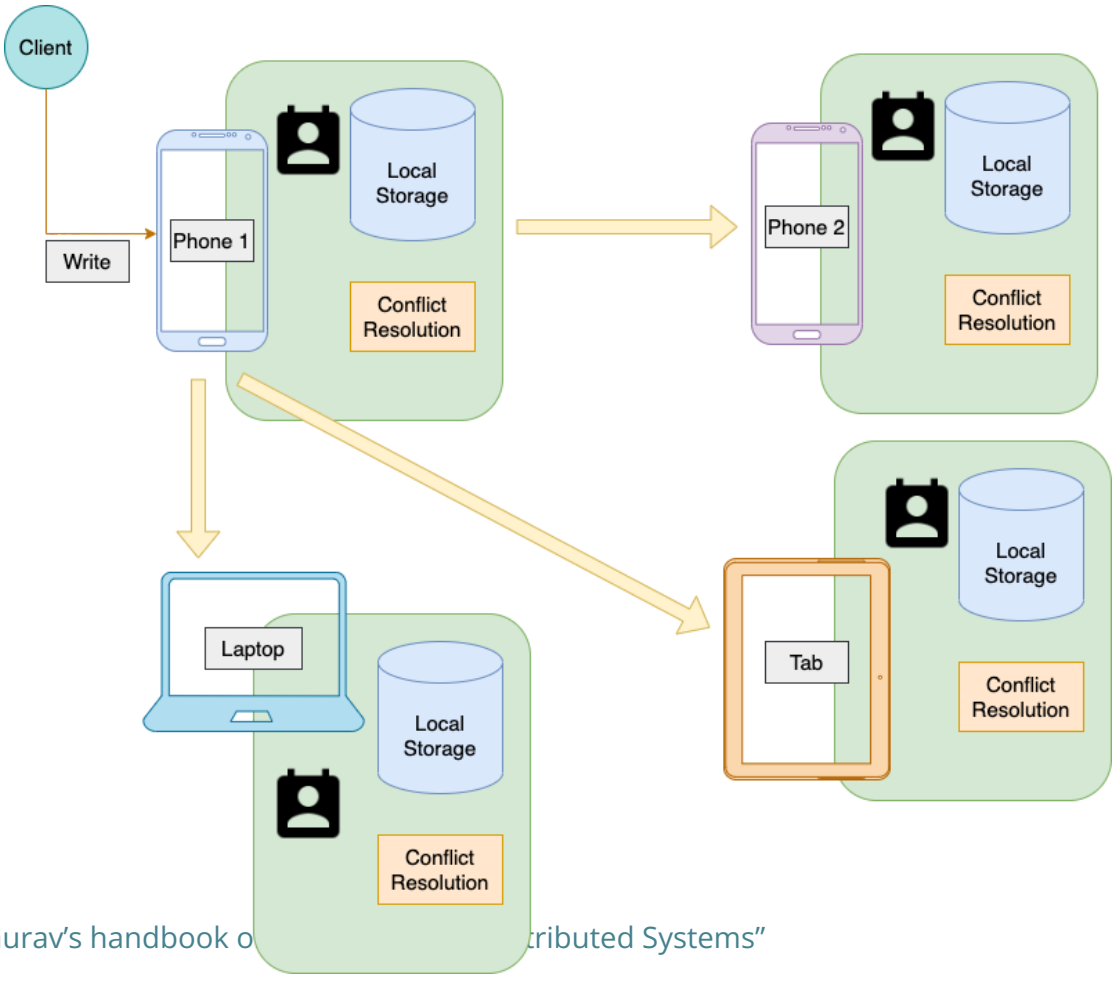
But with the Multi-Leader replication scheme in picture, the client can now send their requests to the data-centre closest to them. This will save some latency and also reduce the response time of the servers. This is possible since every data-centre had their own Leader and each data-centre functions independently.

Use Case 2: Apps like Google Calendar

Let's take an example of **Google Calendar**. We might want to use it even though we are not connected to the internet. We might need to add an event/reminder to our calendar or remove an event from the app even though we are offline and would expect it to be synced with the calendar app logged-in to other devices when we go online.

Suppose we have our Google Calendar signed into two of our phones, one laptop and one tab. We added an event to our Calendar through one of our phones while it was offline. At that time the event will be saved to that phone's local storage and will be synced with all the other devices when our phone goes online.

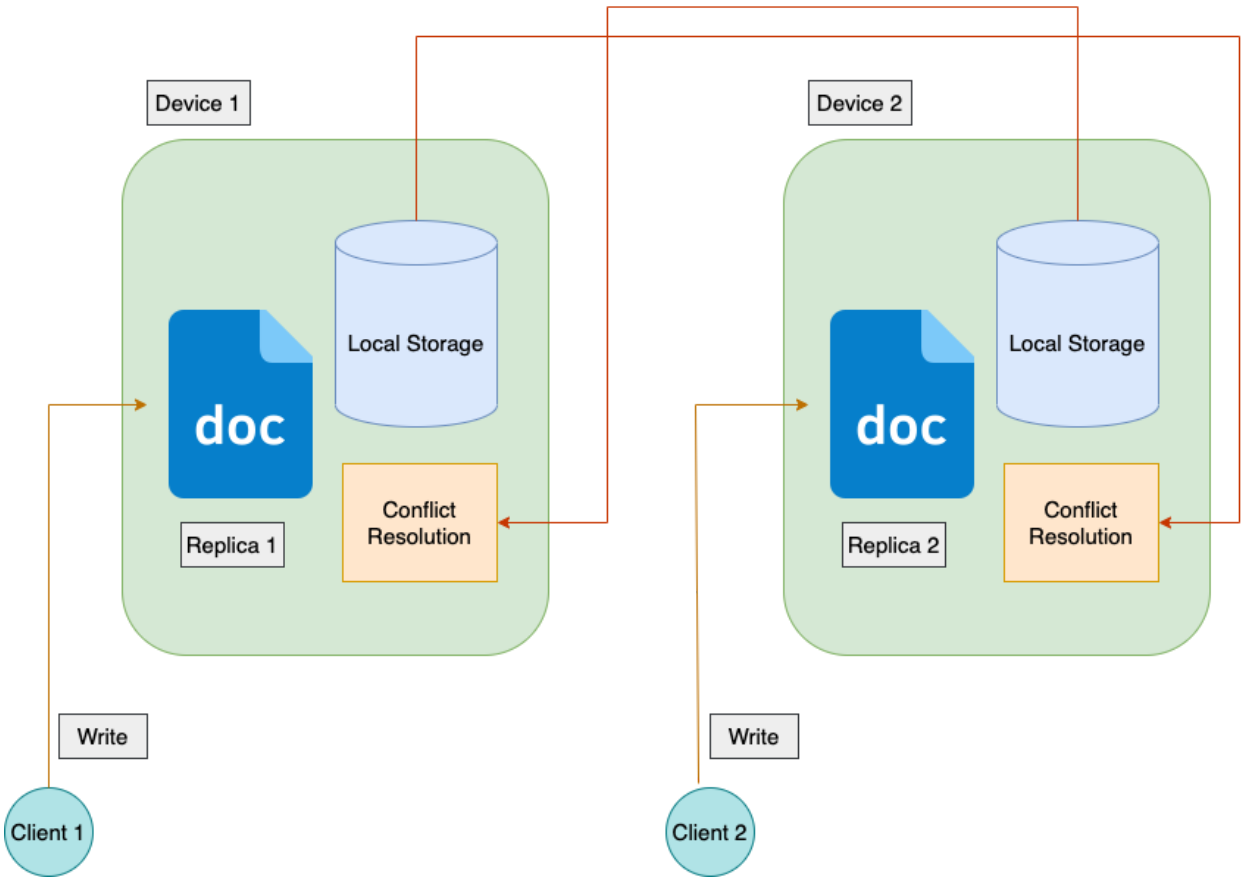
This is similar to the Multi-Leader replication scheme where each device acts as a Leader since it accepts the reads and writes made by its clients. There is an asynchronous multi-leader replication scheme between the replicas of our calendar present in all of our devices.



The above architecture is similar to our previous example of data-centres. Here every device acts as an independent data-centre and the network between them is extremely unreliable.

Use Case 3: Collaborative Editing in Google Docs

Google Docs provides a Collaborative Editing feature where multiple users can edit a single document simultaneously. This can be similar to our previous use case. Suppose a user is offline and edits the document and then the changes are stored in the local storage available in their browser or client application and then asynchronously replicated to the servers and other users editing the same document.



This architecture can lead to serious **Write** conflicts where two users edit the document at the same time and at the same location (in the doc). We need to resolve this conflict by accepting one of the changes and discarding the other. This is one of the major challenges faced in the Multi-Leader replication scheme.

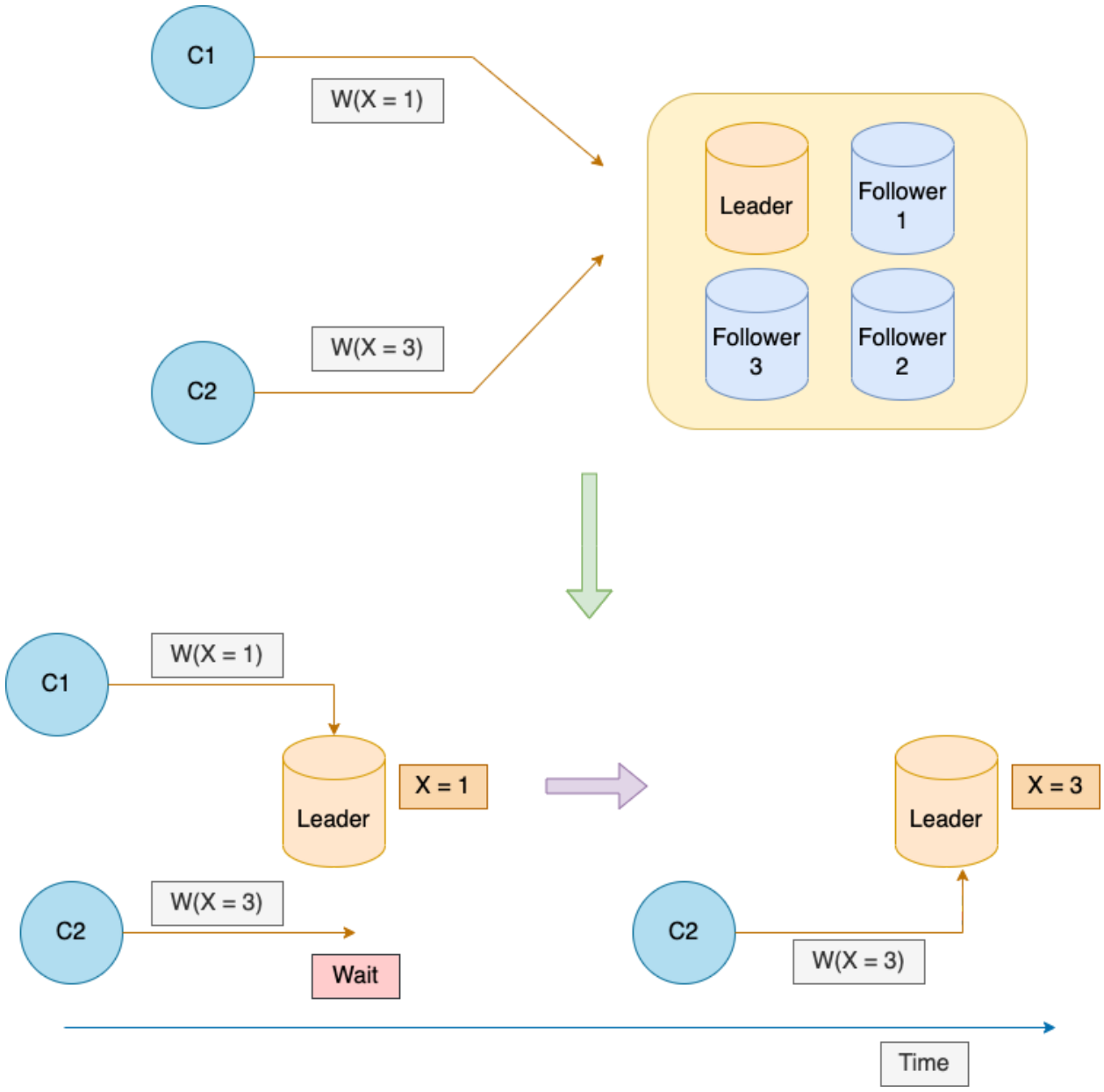
One way to resolve this is by the process of acquiring locks. This will ensure that there will be no **Write Conflicts**. The user must obtain a lock over the document before editing it and if another user wants to edit the same document then he/she might need to wait until the first user has committed their changes and released the lock. This is equivalent to the Single Leader replication scheme but will make the entire concept of collaborative editing process super slow with a terrible user experience.

Write Conflicts

Write Conflicts can be a situation where our system ends up with two different values for a single item and not able to make a decision on which of the copies to keep and which one to discard.

Synchronous Conflict Detection

In case of a **Single Leader Replication** scheme, we have a single Leader to which all the clients will be sending their updates/writes. Hence if two clients say **C1** and **C2** try to update the value of data-item **X** at the same time then the Leader will accept the write request of one of the clients while the other client may wait or its request can be discarded.

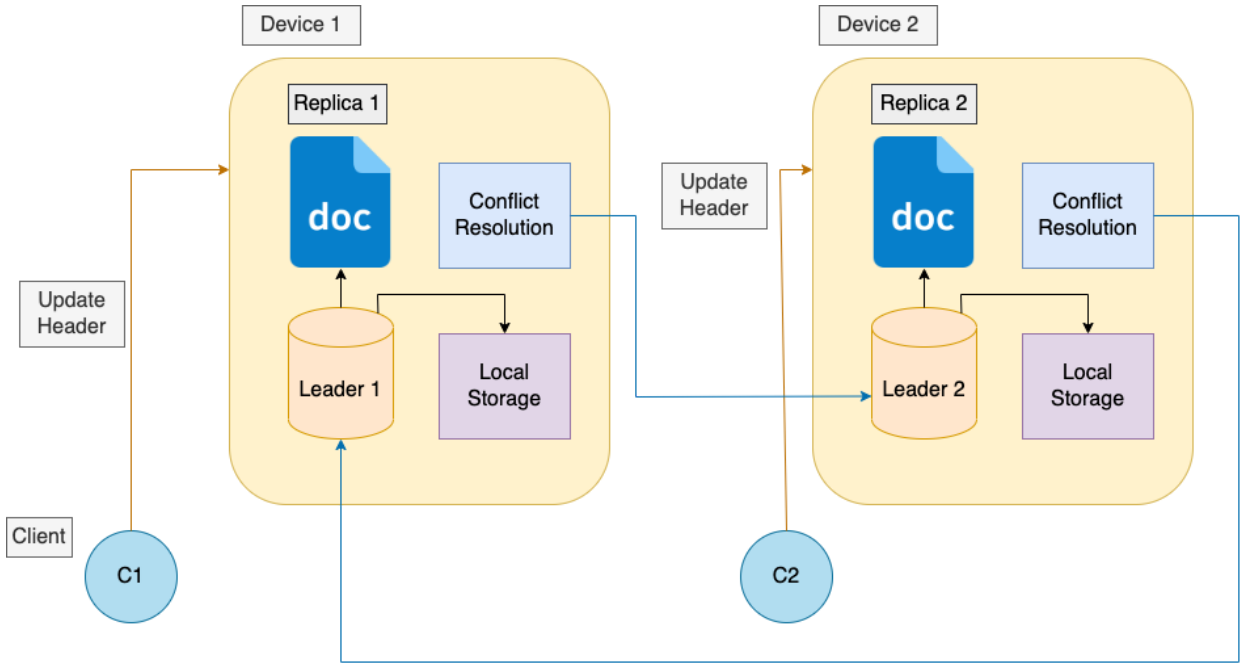


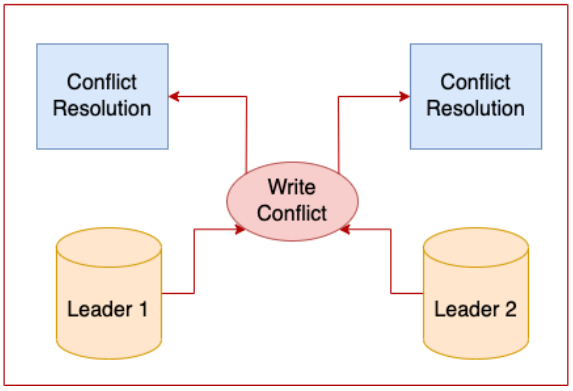
In the above scenario, since Client **C2** request was taken up by the Leader at the end, hence the value of data-item **X** will be **3**. The Conflict Resolution process is very simple in this scheme. The write which takes place in the end will persist in the data-store (when being written over the same data-item).

Asynchronous Conflict Detection

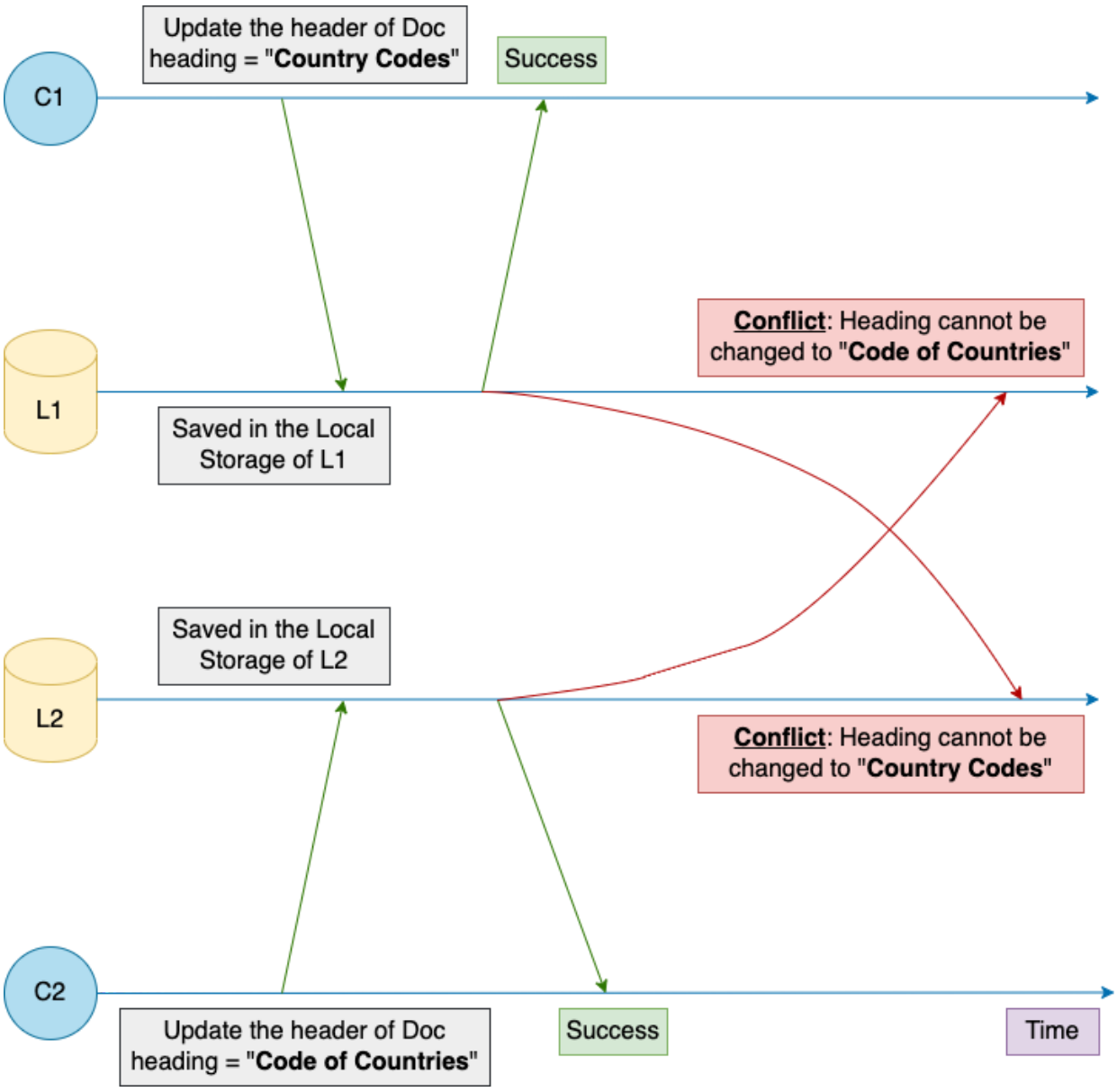
In the **Multi-Leader Replication** scheme the **Conflict Resolution** is tough. There is a strong chance that Write conflicts will occur in this scheme.

Let's take the previous example of **Google Docs** collaborative editing feature (example used in previous section). We have a common document whose replica is present in two local devices. Now **Client 1** tries to update the heading of the doc in **Device 1**. Now at the same time **Client 2** also tries to update the heading of the same doc in **Device 2**. Since both the device holds the replica of the same doc and have independent Leaders, the writes will succeed at that time. The update made by the clients will be stored in the local storage of the respective devices. However when the changes are synchronously replicated, then the conflict will be detected.





The problem with the Multi-Leader replication scheme is that the conflict is realised at a later point of time. Then it's hard to resolve the conflicts at that point.



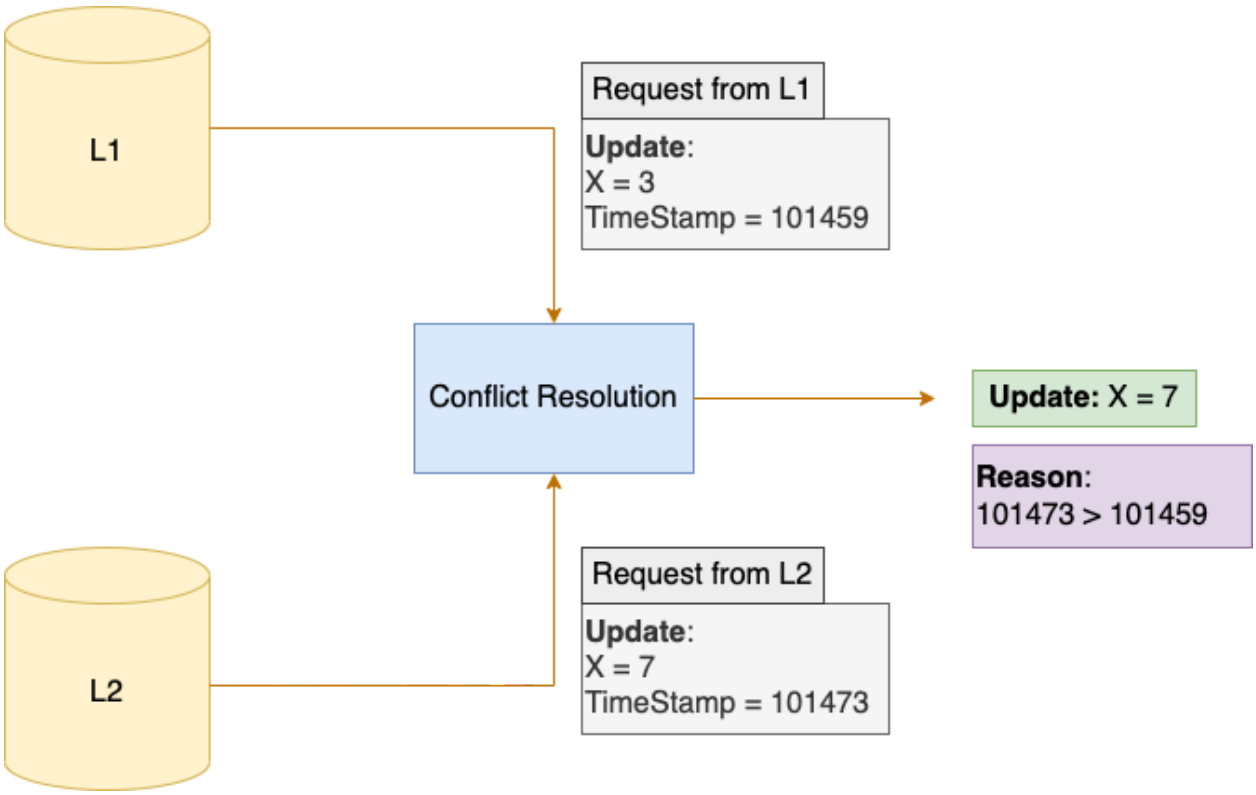
As we saw in the previous diagram we can say that in the **Multi-Leader** configuration there is no default ordering of writes, hence it's not clear what the final value of a data-item should be. Whereas in the **Single-Leader** replication configuration the writes were applied in the sequential order. Hence if there are

multiple writes applied to the same data-item, then the last write determines the final value of that data-item.

For the Multi-Leader replication scheme we will look around ways to resolve the Write Conflicts. The end goal is to make all the replicas agree on a single value for the data-item.

Method 1: Last Write Wins (LWW)

One approach to resolve the Write Conflicts is to assign a **unique ID** to every Write request. This unique ID can be a timestamp, a random long number or hash of the key and value. Once we have assigned every Write a unique ID, then in case of conflict we just need to accept the Write with the highest unique ID and discard the rest of them. If we use timestamp as a unique ID, then the process is known as **Last Write Wins** (**LWW**).



In the above scenario, both the Leaders **L1** and **L2** have two different copies of data-item X. During the Conflict Resolution process, the update from Leader **L2** was accepted while **L1's** request was discarded. Since the timestamp for **L2** request was higher than that of **L1**.

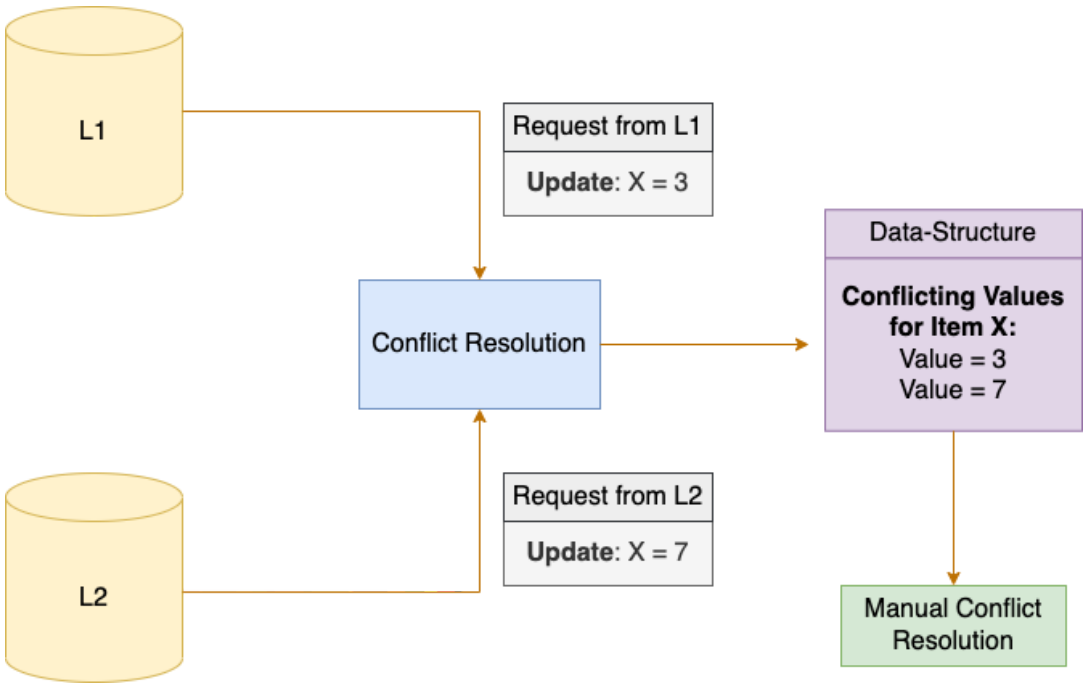
Method 2: Assign Unique ID to the replicas

This is similar to the previous process but in this case we will assign a unique ID to every replica present in our system. After that, the update sent from the replica with a higher unique ID will be accepted while the request from the rest of the replicas will be discarded, in case of Write conflicts.

Note: Both the above Conflict resolution schemes will lead to **Data Loss**.

Method 3: Manual Conflict Resolution

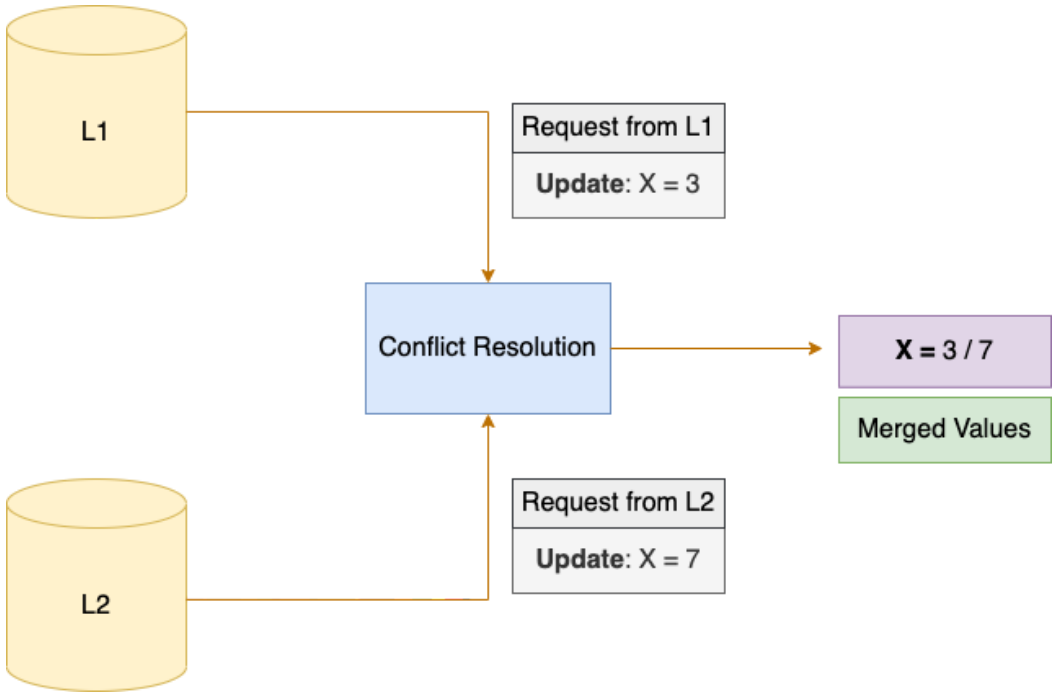
In our previous two ways the entire process of Conflict Resolution was automatic. In this scheme we will store all the conflicting values for a data-item in a data-structure and will later present the conflicting values to the user and wait for the manual conflict resolution.



In this scheme the user will select which value of the data-item (among the conflicting values) is needed to be preserved while the rest of the values can be discarded.

Method 4: Merging Values together

One possible solution could be to merge the values of the data-item together in case of Write Conflicts. The process can look like this.



Thank You! ❤️

Hope this handbook helped you in clearing out the concept of **Replication** in **Distributed Systems**.

Follow me on Linkedin
Do subscribe to my engineering newsletter “Systems That Scale”