Detailed Design of ZooKeeper

Understand the design of ZooKeeper in detail and learn more about its components.

We'll cover the following

We have discussed the ZooKeeper architecture at a high level in the previous lesson. Now it's time to learn more about its components, the client API and the ZooKeeper servers, and how they interact.

Client API

The client API provides a set of functions allowing the client to communicate with the ZooKeeper server. The following are the functions/methods provided in the client API:

- `create (path, data[], mode, flag)`: This method creates a znode. `path` specifies the location in the coordination data tree at which the znode is to be created. For example, `/app1/` in the coordination data tree maintained in the ZooKeeper server's memory is a path at which we can create a znode. `data[]` specifies the data to be stored in the created znode, and `mode` allows the client to choose whether the znode should be regular or ephemeral. `create()` returns the name of the created znode. If the sequential `flag` is set, the number from the monotonically increasing counter will be appended to the name of the znode.

> The **regular mode** gives full control of the znode to the client. Only the client can create and delete such znodes. These znodes exist even after the client is disconnected. All the znodes in the ZooKeeper are, by default, regular unless specified otherwise. The **ephemeral mode** is created by the client, but the system has the right to delete it if the session has expired.

- `setData (path, data[], version)`: This method sets the `data[]` in the znode at the specified `path` and with the specified `version`.
- `getData(path, watch)`: This method returns `data[]`, which was set through `setData()` and the metadata of the znode at the specified `path`. The `watch` flag allows the client to set a watch on that znode.
- `getChildren (path, watch)`: This method returns all the children names of the znode at the specified

`path`, and the `watch` flag allows the client to set a watch on that znode's children.

- `exists (path, watch)`: This method checks whether there exists a znode at the specified `path` and returns its metadata information. The `watch` flag allows the client to set a watch on that znode.

> The `watch` flag will work only if the znode at the specified path exists. Watches are applicable only for the get and `exists()` methods.

- `delete (path, version)`: This method deletes the znode at the given `path` of the given `version`. If any of the parameters don't match, then the znodes will not be deleted.
- `sync (path)`: This method waits for all the requests on the znode of the `path` to be completed, which are generated by the client connected to the server.

> **Note:** By using these functions, one has the ability to develop a range of coordination artifacts.

ZooKeeper offers znodes as shared registers with watches as an event-driven method that is comparable to the distributed system's cache invalidation. ZooKeeper provides a simple yet powerful coordination service in this way.

Server

The ZooKeeper service is replicated, which means that all of its data is kept on a single server, and the same data is replicated on other servers to deal with the single point of failure issue. It distributes the load of requests and provides service availability at each server. The collection of these replicated servers is called the **ZooKeeper ensemble**. All these servers work together to provide services to the client. One server is elected as the leader, and the others become the followers.

Unlike Chubby, clients are not bound to only connect with the leader, called the **primary replica** in Chubby, but they can also connect with the followers (called **replicas** in Chubby) to perform operations. This design decision provides ZooKeeper to have high availability but does not provide strong consistency until clients go to the leader for reads as well. The `sync()` method discussed above can be used to perform synchronization but on a need basis. The leader and the followers differ in their roles as follows:

- **The leader:** The leader, on receiving a client's write request, broadcasts the operation to the followers, performs the write operation on the coordination data placed in its memory, and acknowledges the client.
- **The follower:** The follower can also receive and respond to write requests. Multiple write requests are queued in the server so that they can be executed in the FIFO order. However, only the write request needs to be forwarded to the leader, and the leader broadcasts the request to all other followers. After broadcasting the request to the followers, the leader responds to the follower who forwarded the write request to it. Then, that follower replies to the client's write request. The broadcasting of the write requests ensures that each server has eventually the same data to show the client. For read requests, the follower doesn't need to forward the request to the leader and can

process the request itself.

> **Note:** It might seem strange why a follower sends the write requests to the leader (which in turn broadcasts it to all the followers) and why the client doesn't send the request to the leader in the first place. One reason is that the end clients don't need to keep track of the current leader since the ZooKeeper service nodes will know it. Secondly, replicas can combine many requests in one communication with the leader, acting like a proxy server.

Question

We know that we have a collection of servers. What would be the minimum number of servers, and why?

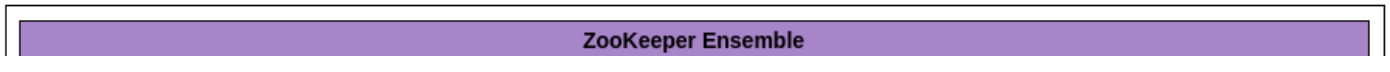
Hide Answer

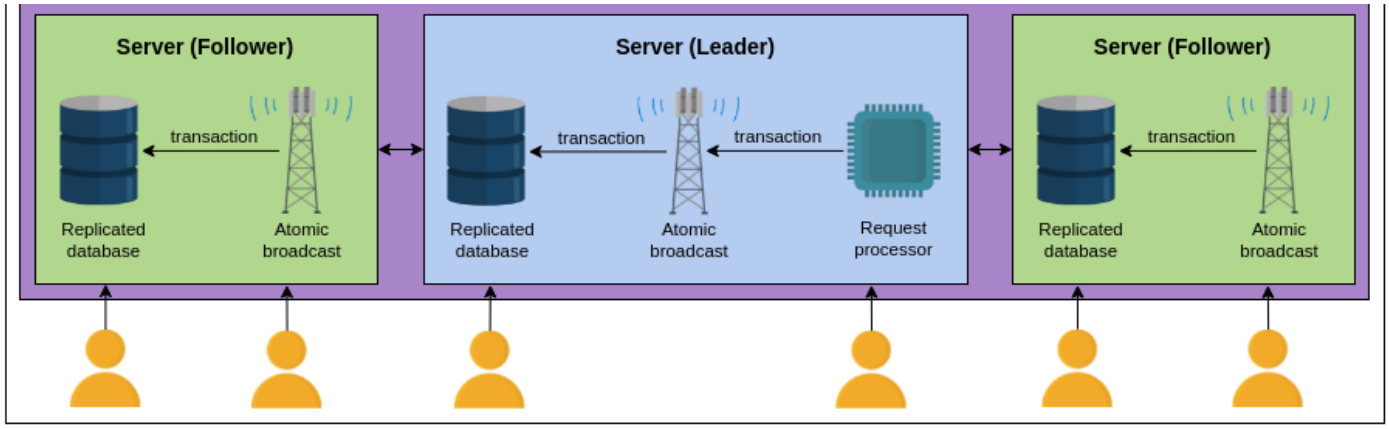
To answer all of our questions, we have the following reasons:

1. A single server node can always lead to Single-Point-of-Failure, and there won't be any followers.
2. Having two server nodes means that in case of a failure, we will not have the majority (quorum of $floor(\frac{n}{2} + 1)$, which will be at least $\frac{2}{2} + 1 = 2$ servers) for the leader election.
3. Having three server nodes means that in case of a single node failure, the leader election can take place (at least $floor(\frac{3}{2} + 1) = 2$ servers), which makes three server nodes the minimum requirement.
4. Having four server nodes means that in case of a single node failure, it will work perfectly fine. But if the two server nodes are down in two data centers, then there are chances that both data centers will lose a single server node. In such a case, both data centers will be unable to perform the leader election. As long as a majority of m out of n are available, the system can tolerate $n - m$ failures.

The ZooKeeper server has the following three services, as shown in the figure below:

- **Replicated database:** This is an in-memory copy of the database so that read and write operations can be done locally.
- **Atomic broadcast:** This is used by ZooKeeper to broadcast the write request to the replicated database on all the servers. In the case of a follower, it will send the write request to the leader.
- **Request processor:** This is like a manager that keeps all the transactions atomic. Only the leader can use this service.





The detailed architecture of ZooKeeper

Replicated database

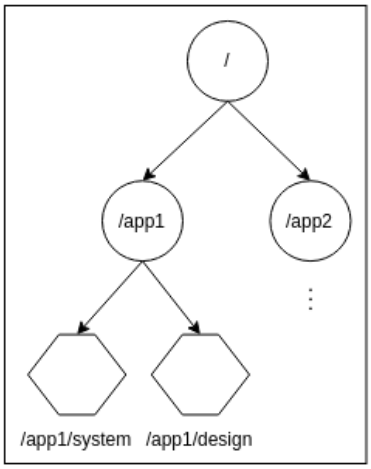
The **replicated database** is the storage component of ZooKeeper. When the client calls either the `create()` or `setData()` method, data is stored in an in-memory data object called a znode (Zookeeper node) in the form of a tree. Keeping all the data and metadata in-memory makes it faster to access data and metadata. It also decreases the latency of read/write operations, since disk operation is not the first one to be performed. Each write operation is logged on the disk before it is performed on in-memory database. After logging it on the disk, if there is any other operation currently being performed on the same data, then this request goes to the queue. Otherwise, the operation is performed.

To enable fault tolerance, ZooKeeper takes periodic snapshots of all the delivered messages (in the case of a leader, messages are delivered to other servers via atomic broadcasting, and in the case of a follower, messages are delivered to a leader). These messages are stored on the disk for recovery. Since this recovery is a background process and doesn't lock the ZooKeeper state, we call these snapshots, **fuzzy snapshots**. If any server fails before the next snapshot has been taken, we perform a depth-first scan of the tree to read every znode's metadata and data atomically and then write that metadata and data onto the disk. This tree is extracted from the write-ahead log which was stored on the majority of nodes as part of consensus.

The changes in the state are forwarded during the creation of the snapshot. These may have been applied, in part, to the final snapshot. It's possible that the state of ZooKeeper, at any given moment, does not correspond to this snapshot. Since states are atomic, this is not a problem even if we apply the same changes in the state twice, but these two changes should be sequential. Let's say that a follower sends a write request W_1 twice to the leader, and the leader puts either one or both in the queue. After executing W_1 once, when the same request is received from the queue, the leader ignores the second request and fetches another request from the queue if there is any. We can call this **at-most-once** execution, and it is

achieved by giving a unique identifier to each request and noting that in the Zab log.)

When the fuzzy snapshot starts, let's say that two nodes in a ZooKeeper data tree, `/app1/system` and `/app1/design`, contain values of `systemDataOne` and `designDataOne`, respectively. Both are at version `1` and the following stream of state changes arrives with the `transactionType, path, value, new-version` format:



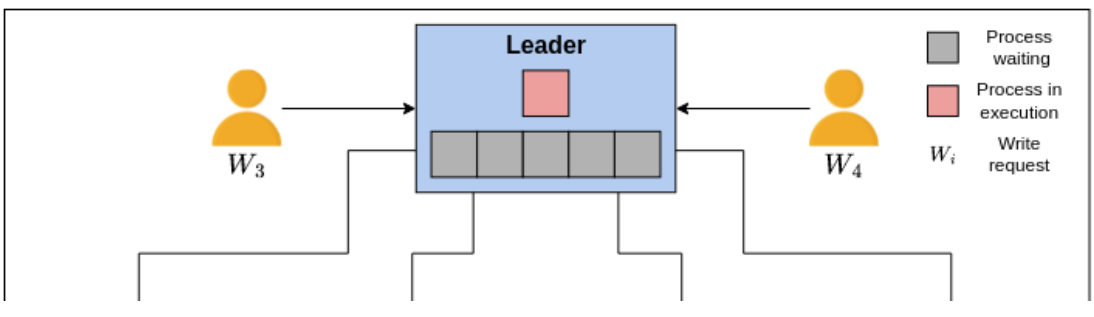

```
⟨SetDataTXN, "/system", systemDataTwo, 2⟩#TXN means transaction
⟨SetDataTXN, "/design", designDataTwo, 2⟩
⟨SetDataTXN, "/system", systemDataThree, 3⟩
```

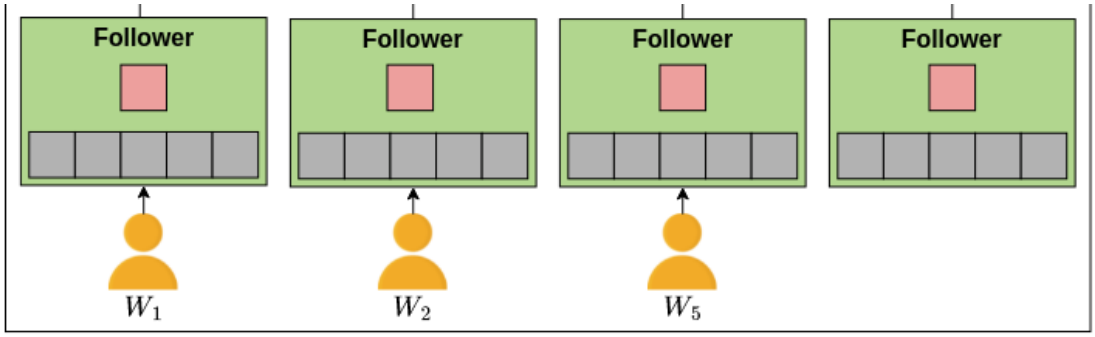
Following the processing of these state modifications, `/system` and `/design` have values of `systemDataThree` and `designDataTwo` with versions `3` and `2`, respectively. Although this was not an acceptable state of the ZooKeeper data tree, the fuzzy snapshot may have shown that `/system` and `/design` had values of `systemDataThree` and `designDataOne` with versions `3` and `1`, respectively. If the server fails, the resulting state corresponds to the service's previous state, which is recovered using the latest snapshot. The leader uses the atomic broadcast protocol to redeliver the state changes.

Atomic broadcast

Any follower receiving the write requests that update the ZooKeeper's state needs to send those requests to the leader. Once the leader receives that request, it executes such requests and broadcasts it to all the followers using the Zab protocol. The server receiving the write request responds to the client application when it performs the write operation and sends the update state message to other servers. By default, the Zab uses a majority quorum ($2f + 1$, where f is the number of faults we can tolerate).

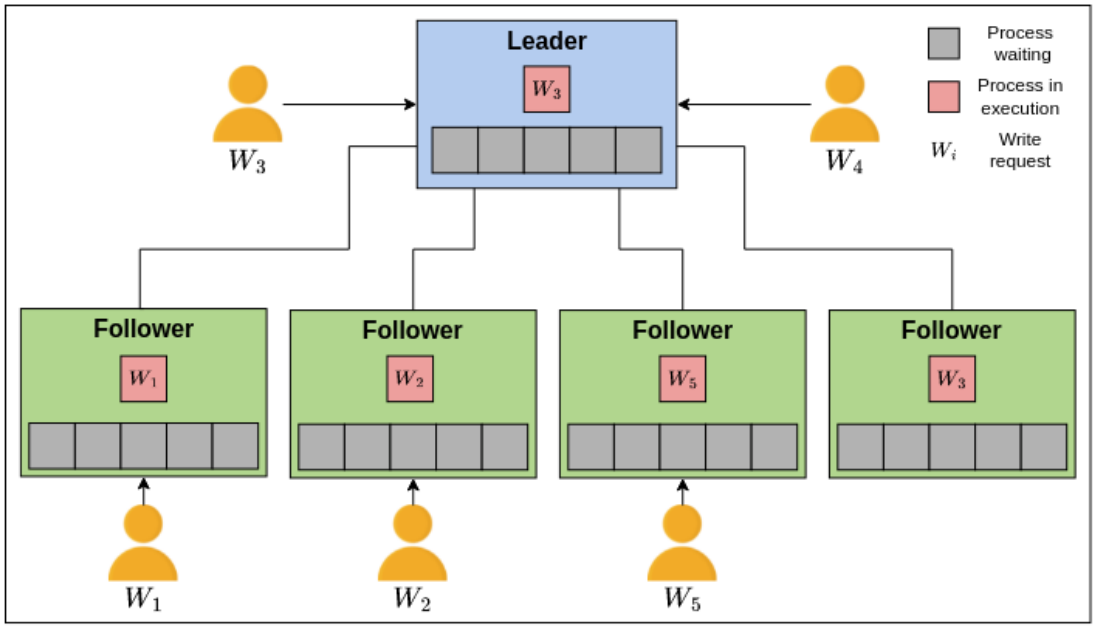
The ZooKeeper uses the FIFO ordering of requests, allowing followers to receive the requests in the same order as they were sent by the leader. In case of leader failure, a new leader is elected, and the new leader ensures all the updates from the previous leader are incorporated into its replicated database before broadcasting its own requests, as shown in the animation below. The newly elected leader calls the `sync()` only on itself and does not broadcast it. It performs all the operation's requests from the clients to stay up-to-date like the previous leader.





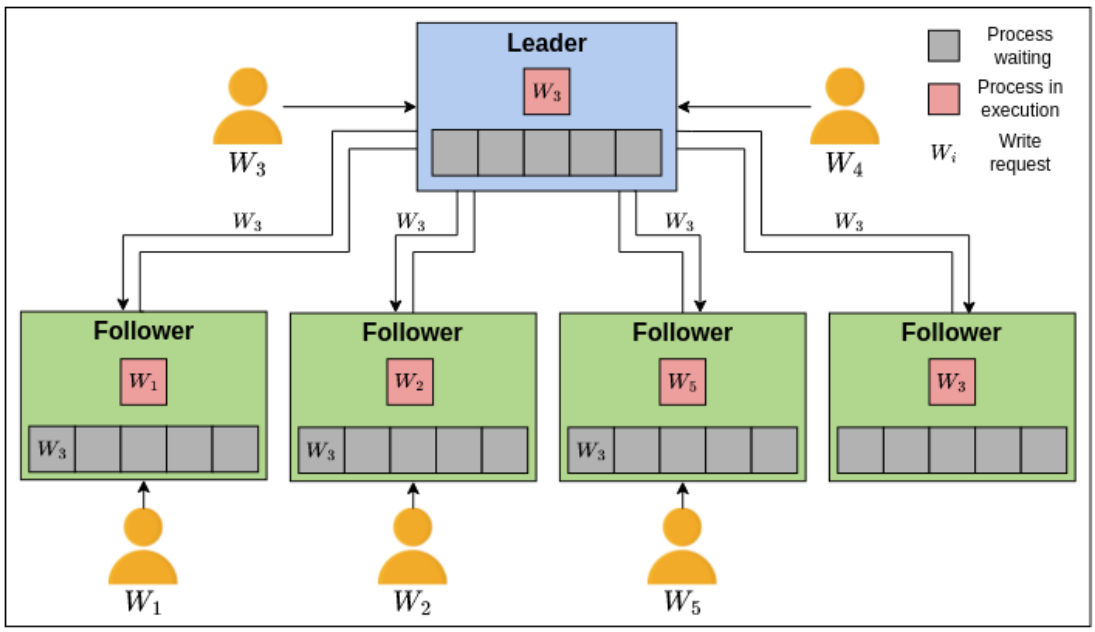
Five clients connect with different servers to perform multiple write operations on the same data

1 of 20

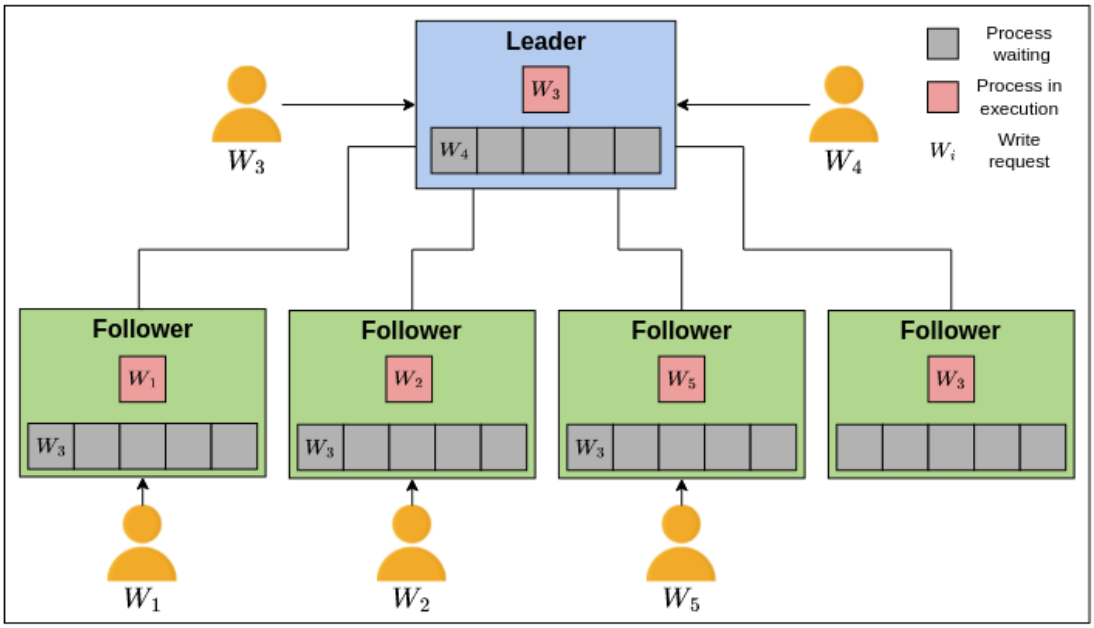


All the servers start working on the write requests they received, and W4 on the leader will be queued once W3 has been broadcasted

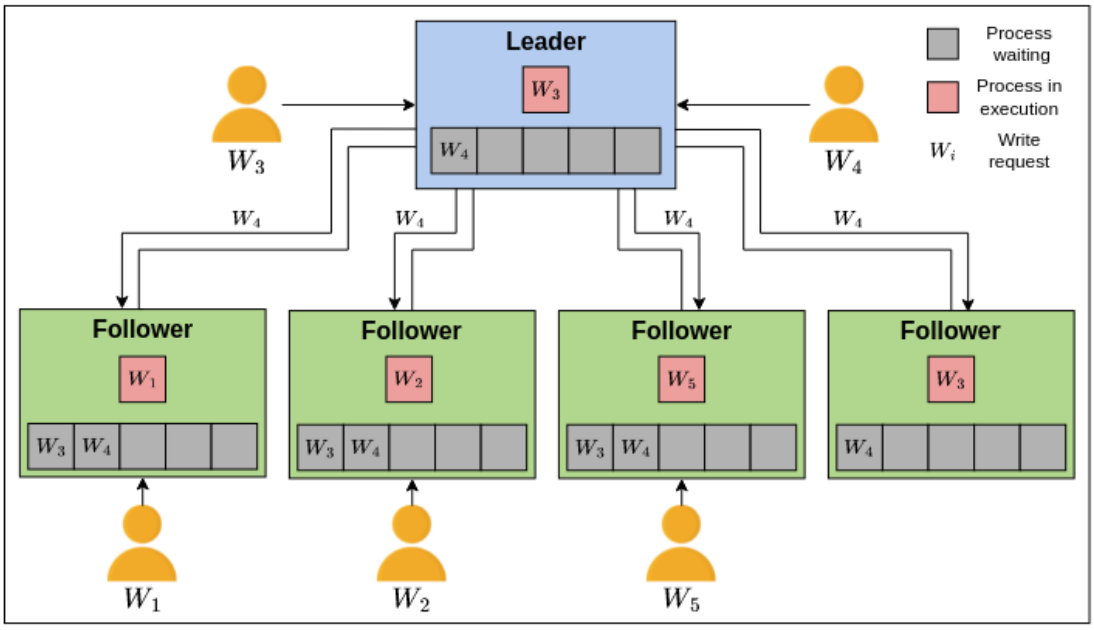
2 of 20



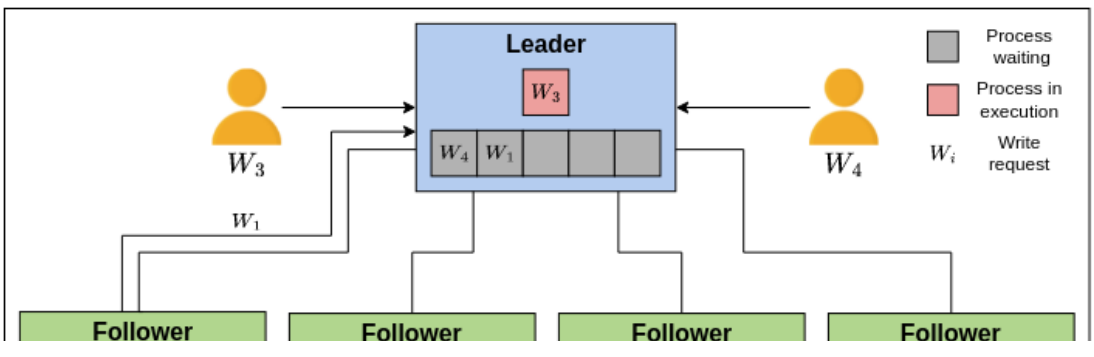
The leader broadcasts the write request it is processing to all the followers, and the followers put that request in the queue since they are currently working on one

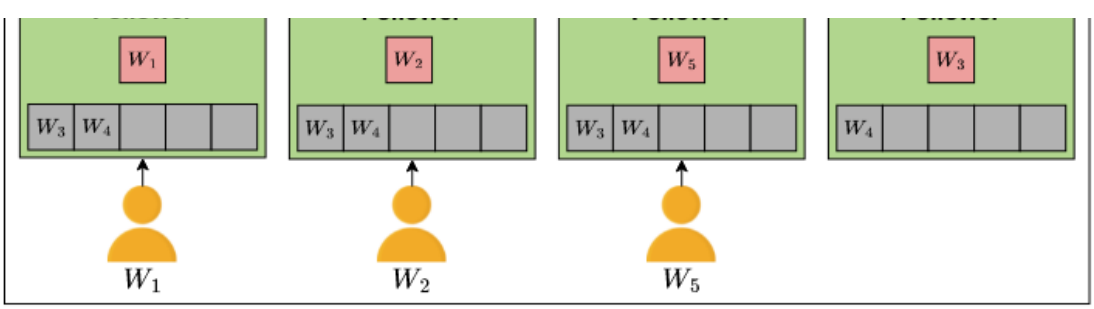


The leader puts the other write request in the waiting queue

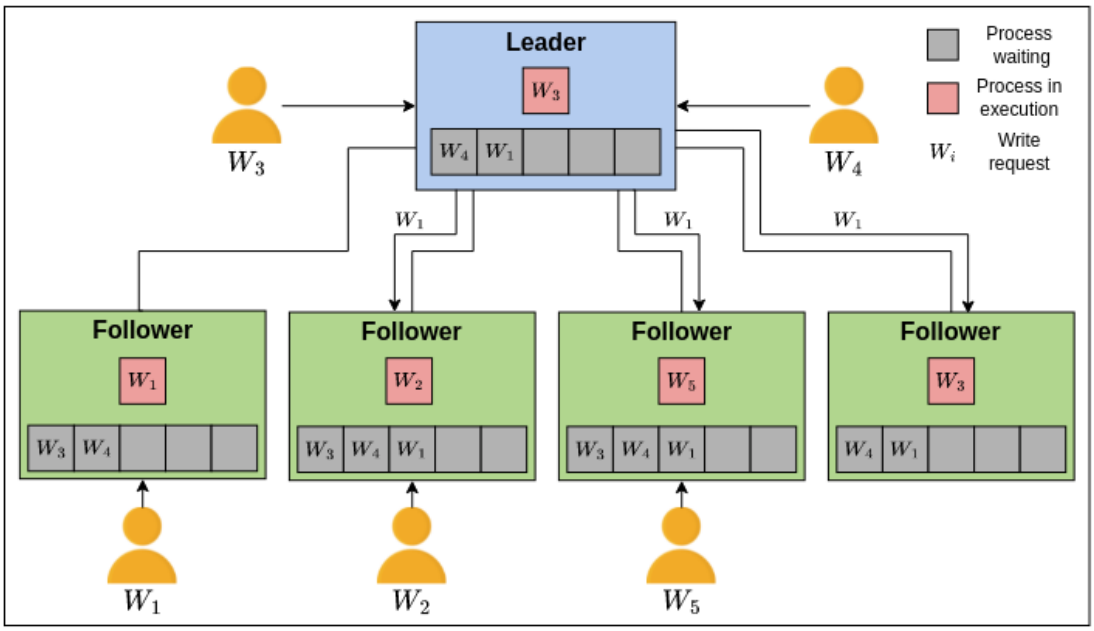


The leader broadcasts the queued write request to all the followers, and the followers put that request in the queue since they are currently working on one

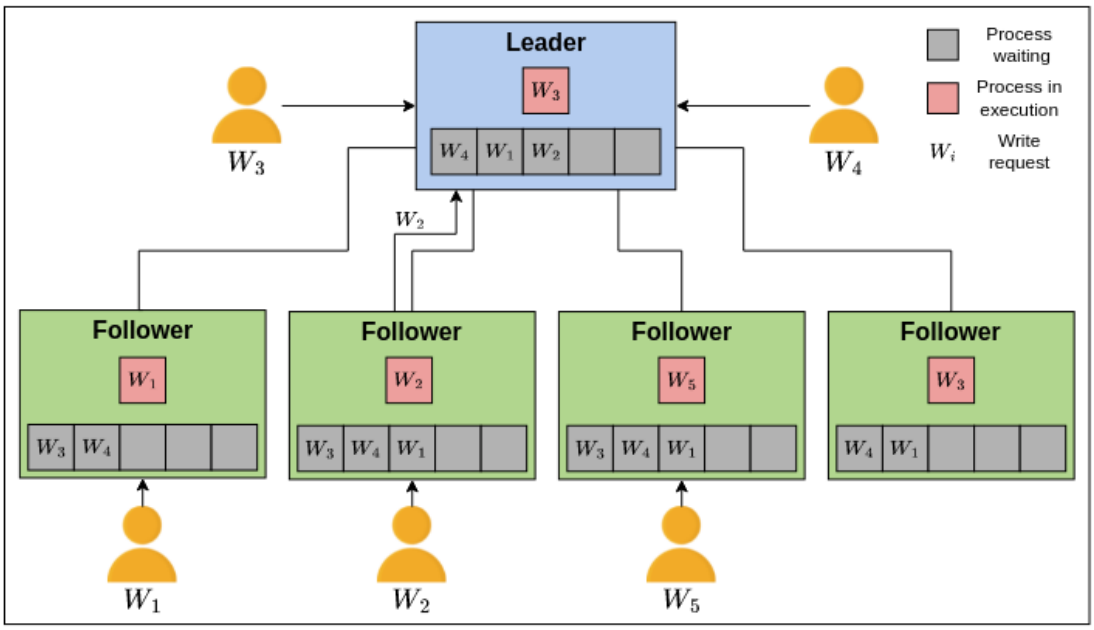




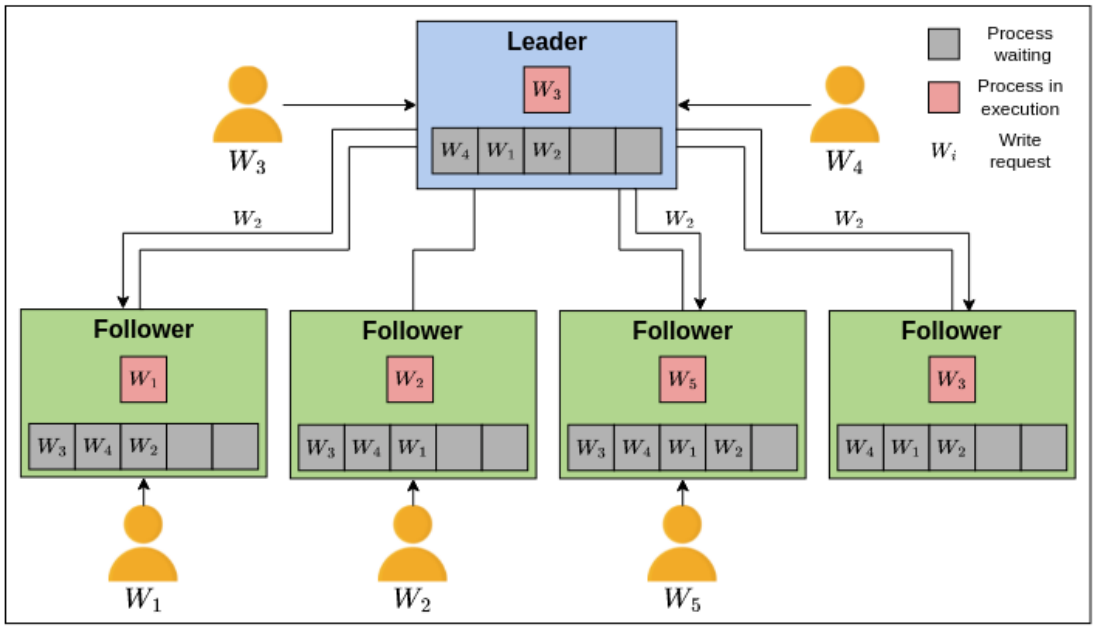
A follower sends its write request to the leader so that it can be broadcasted, and the leader puts it in the queue



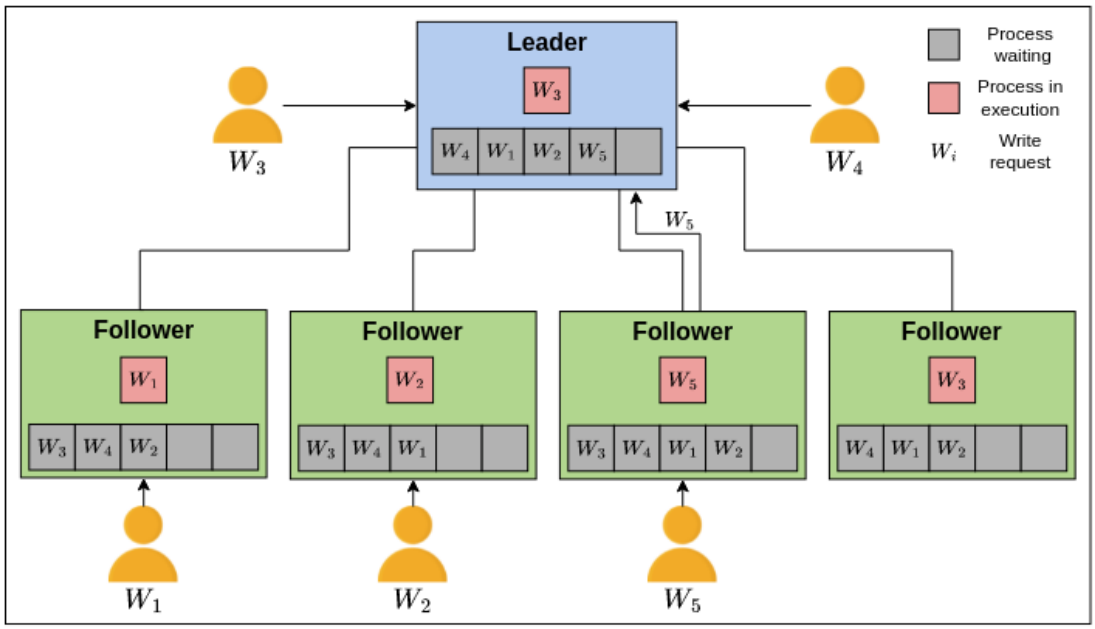
The leader broadcasts the queued write request to all the followers, and the followers put that request in the queue as they are currently working on one



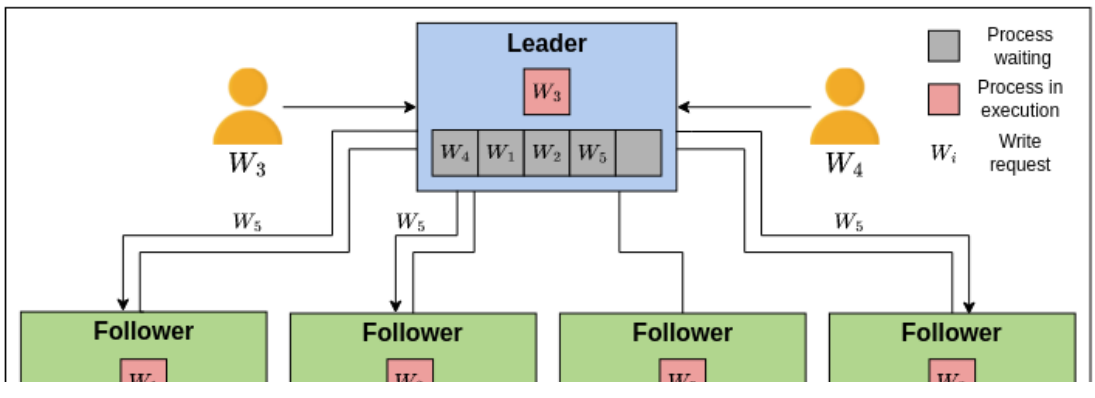
A follower sends its write request to the leader so that it can be broadcasted, and the leader puts it in the queue

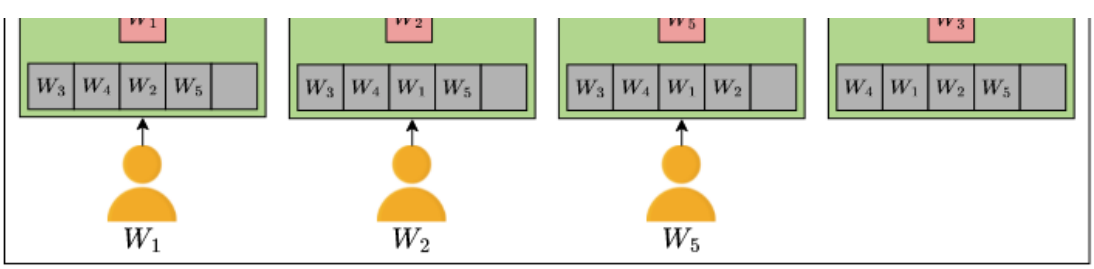


The leader broadcasts the queued write request to all the followers, and the followers put that request in the queue since they are currently working on one

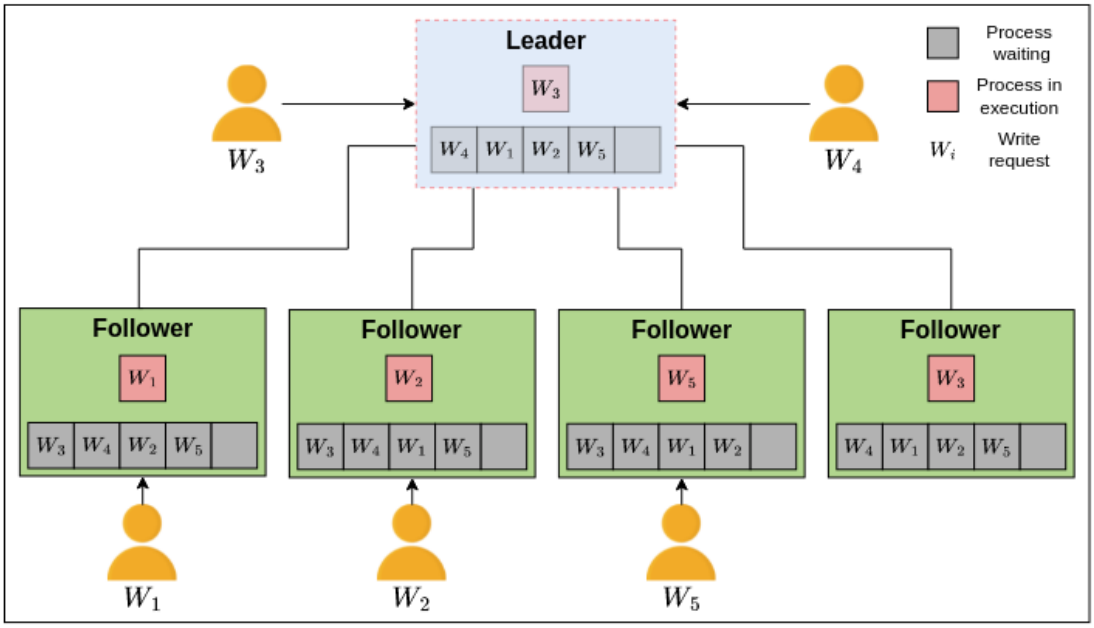


A follower sends its write request to the leader so that it can be broadcasted, and the leader puts it in the queue

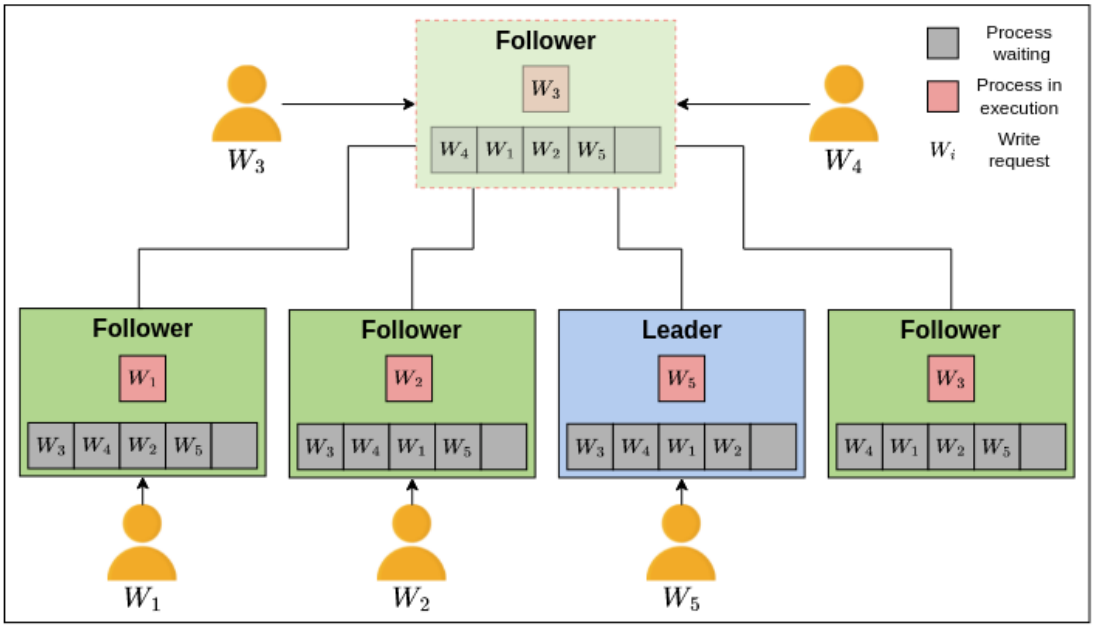




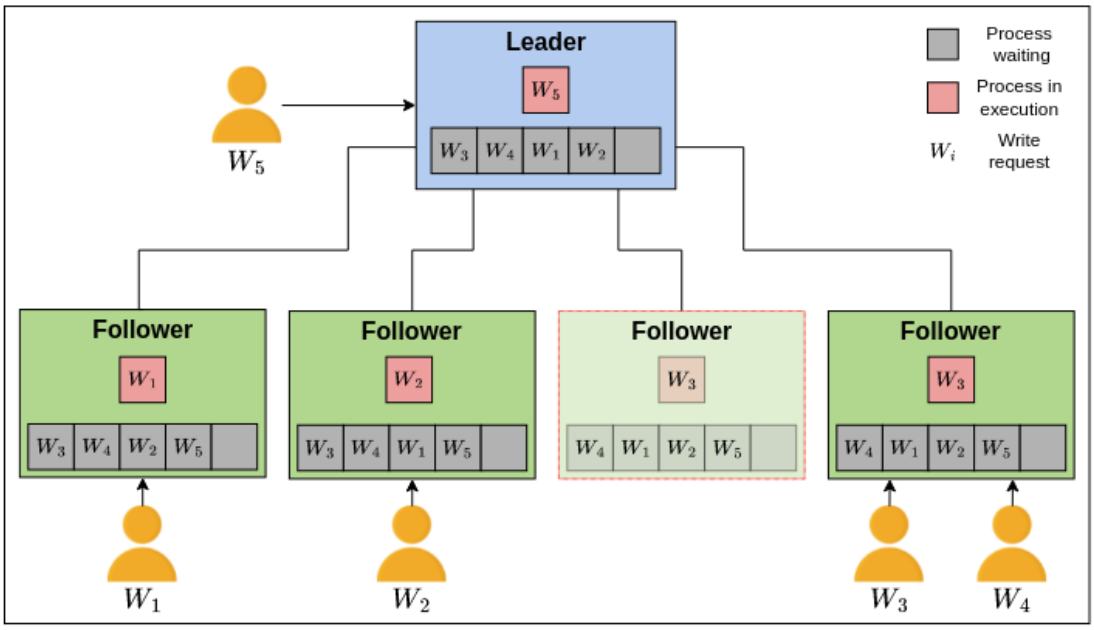
The leader broadcasts the queued write request to all of the followers, and the followers put that request in the queue since they are currently working on one



The leader suddenly fails

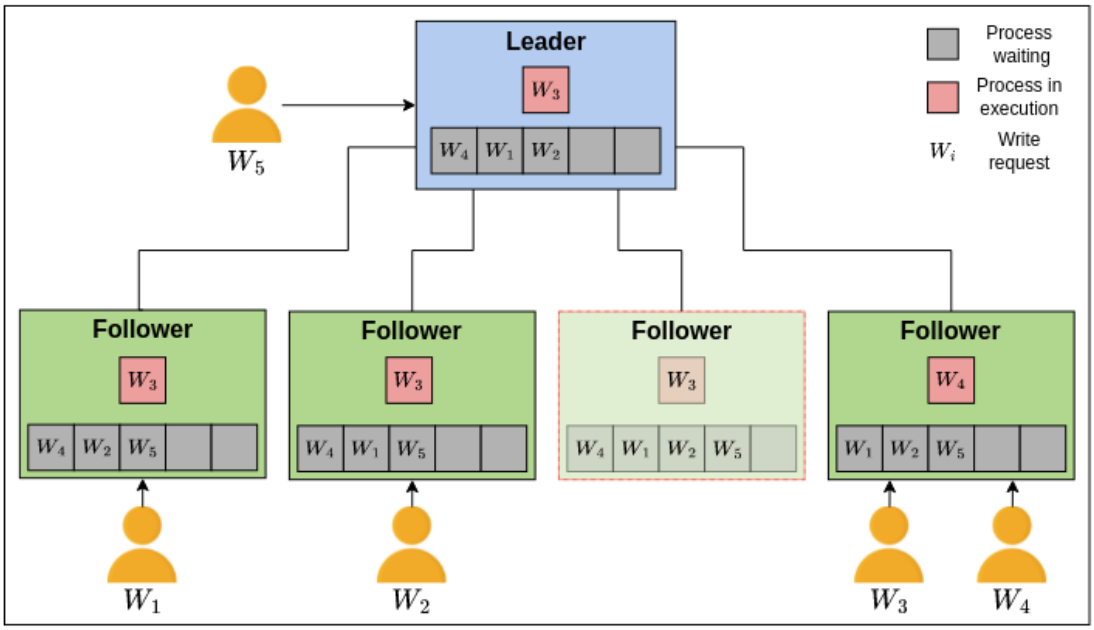


The new leader is elected



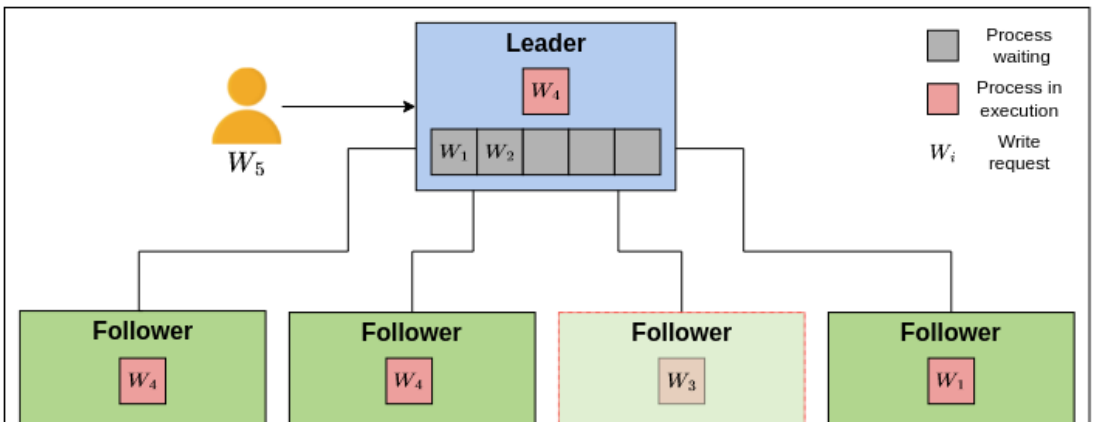
The clients connected to the previous leader are now connected to another server that will give the clients the same interface as the previous server

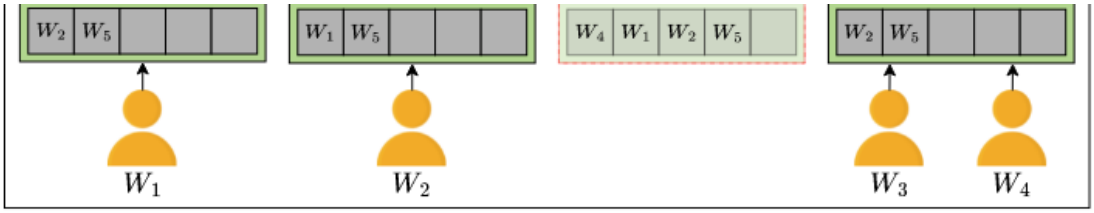
14 of 20



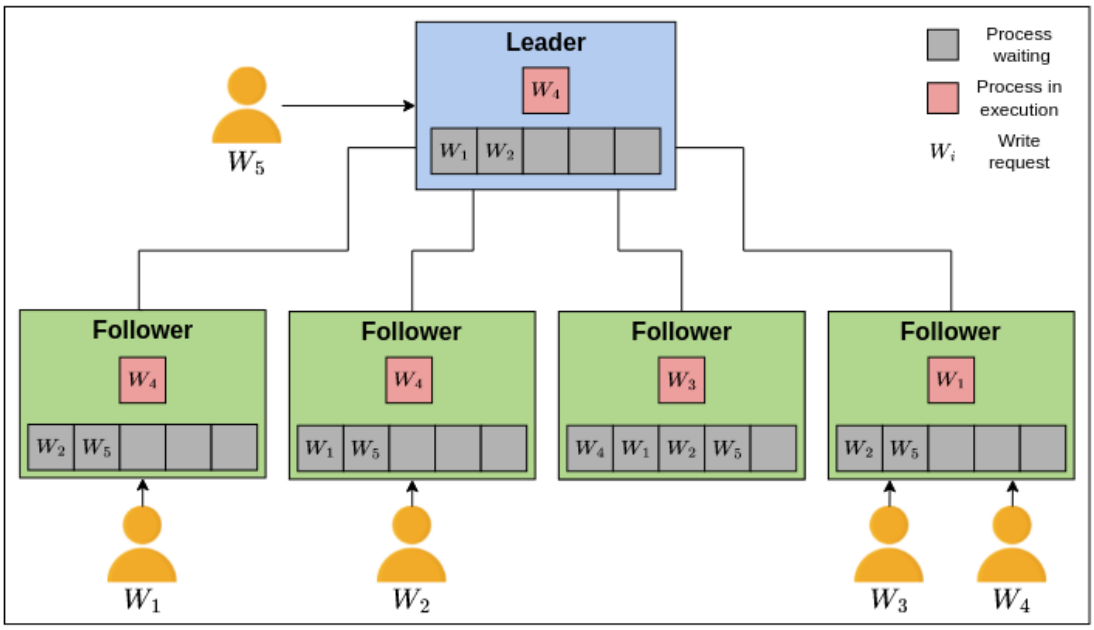
All the active servers have successfully performed the write operation that was in progress

15 of 20

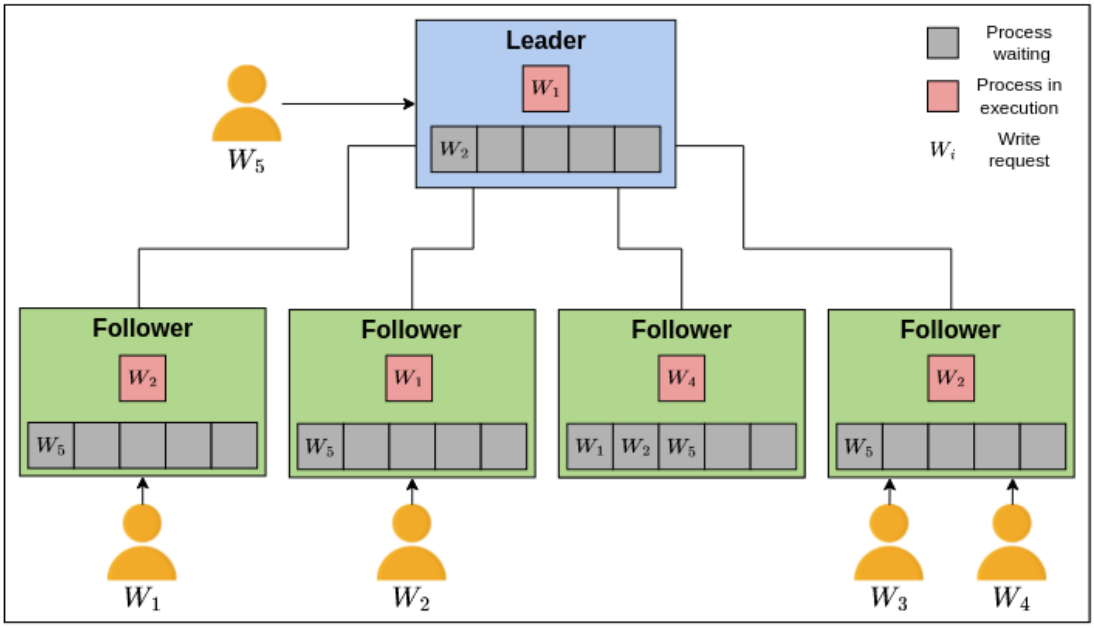




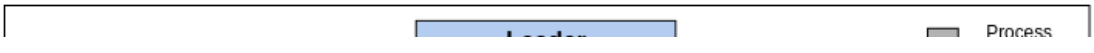
All the active servers have successfully performed the write operation that was in progress

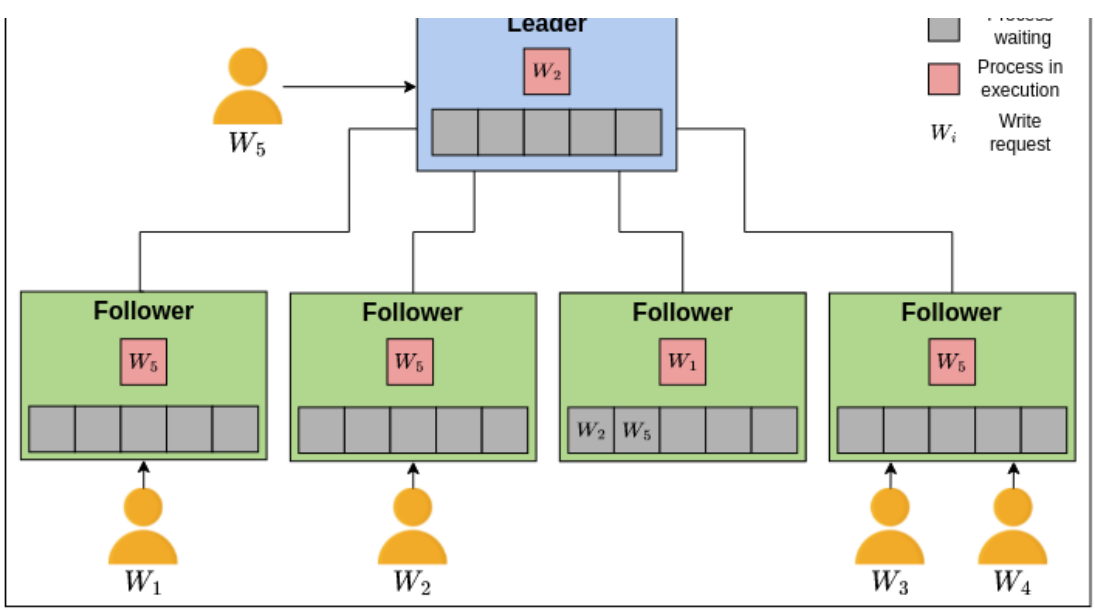


The down server is now recovered and has started to follow the new leader

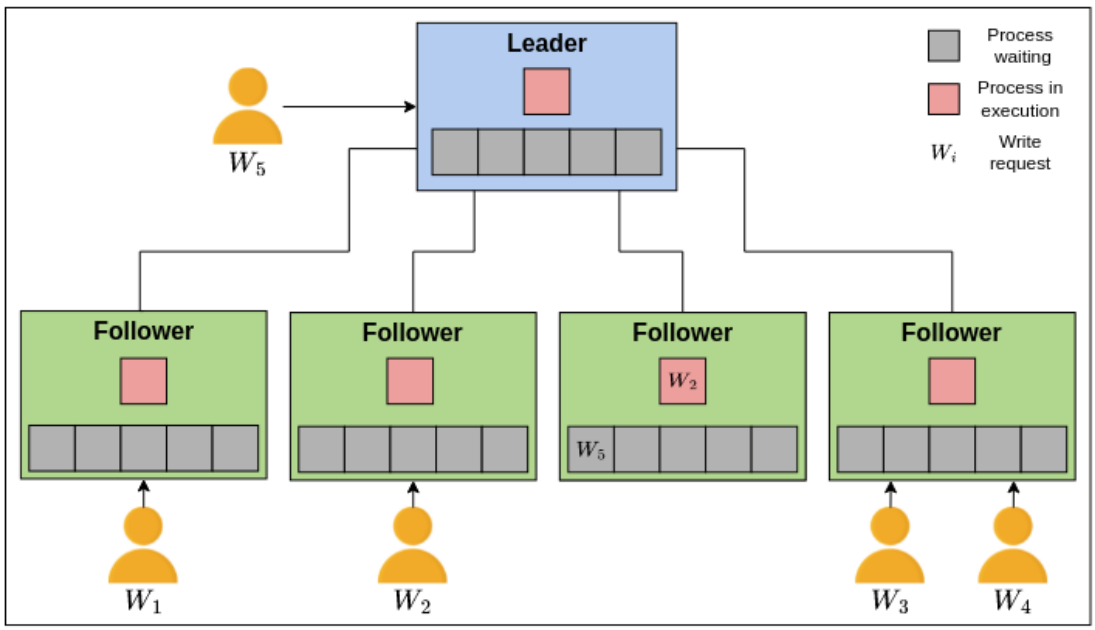


All the servers have successfully performed the write operation that was in progress and have started working on the next request from the queue





All the servers have successfully performed the write operation that was in progress and have started working on the next request from the queue



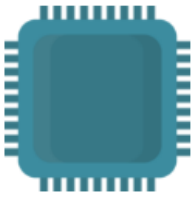
Only one server is left to update itself while rest of the servers are up to date

We simplified our implementation by using TCP in our transport layer, and the network's ordering of the messages is now maintained. The leader election and transaction creation are done through Zab to make it easier for the leader to broadcast the transaction. We need to ensure that the same data is not written on the disk twice, and for that, we have used write-ahead logs to keep track of the write requests that are performed in the in-memory database.

Normally, all requests are sent in order and only once, but in case of a failure, a server can resend the same request during recovery. Since Zab does not keep track of the write requests on the disk, the leader

can receive the same request twice. As long as the requests are in order, the idempotent transactions will bring the state to a final consistent state. ZooKeeper wants Zab to resend all the write requests after the start of the last snapshot.

Request processor



While some servers may have applied more transactions than others at any given time, we ensure that the local replicas never diverge because the messaging layer is atomic. The transactions are idempotent, unlike the requests that clients send. This is possible because we linearize the requests at the leader. After receiving a write request, the leader converts it into a transaction, `setDataTXN`, as shown in the previous section. This transaction captures the new state by calculating the system state that will exist once the write is implemented. Given the possibility of pending transactions that haven't yet been applied to the database of the leader, the future state also needs to be computed.

The service creates `setDataTXN` for updating a transaction containing updated data and an updated version with current time stamps required for the transaction. For instance, the client calling the `setData()` method contains the upcoming version number of the znode, then a new transaction, `setDataTXN`, will be generated for that znode. An `errorTXN` is generated in the case of a problem, such as a conflicting `version` or an unreachable znode that needs to be upgraded.

Client-server interaction

The read operations `getData()`, `getChildren()`, and others, are performed locally by each server without notifying others. The server generates a `zxid` for every read request and retrieves the most recently updated state of the server's data. This local processing of the read requests gives us extraordinary read performance because the in-memory data operation doesn't require any interaction with the storage disk or agreement protocol. By doing so, we can easily perform read-dominant workloads.

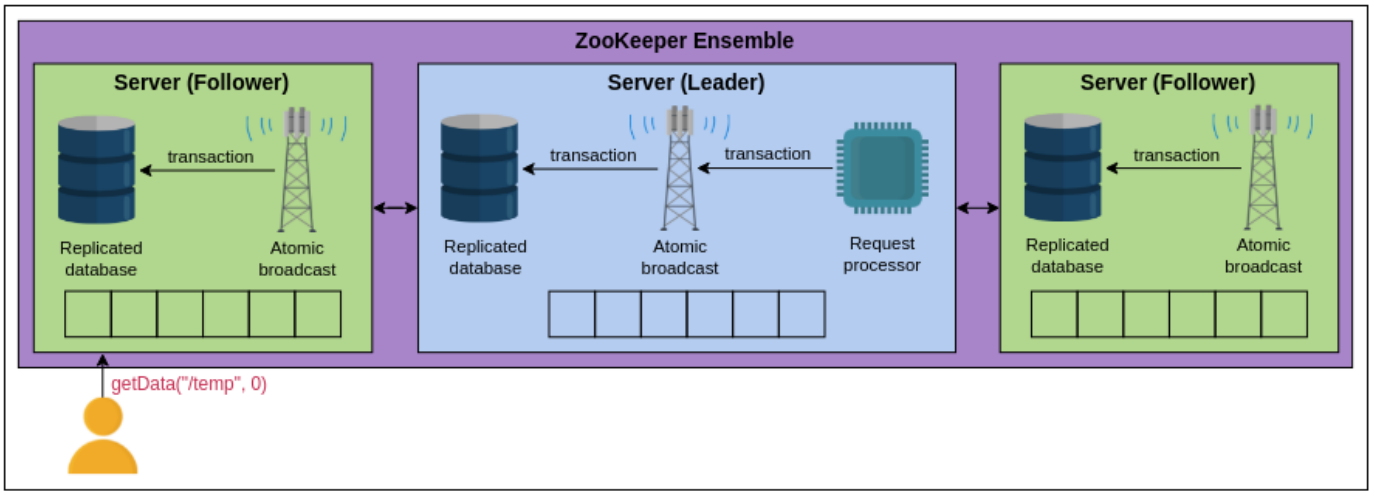
Reading is simple. We just need to go to the server with the `path`, and the server will use its in-memory replicated database to give us the desired data if it is available there. However, the write operation is not that simple. When the write request, `setData()`, is generated by the client, we perform the following four steps:

1. Write the data.
2. Notify the client(s) who have set the watches on that data.
3. Send the write request to the leader so it can be updated in the replicated database of all the servers in the case of a follower.
4. Data is replicated to all the connected servers in the ensemble.

The write requests are FIFO, and even read requests are FIFO when there is already a write request in execution. However, once the read operation is in progress, multiple readers can read in parallel on the same data. The read and write operations are not allowed to be executed in parallel on the same data. The

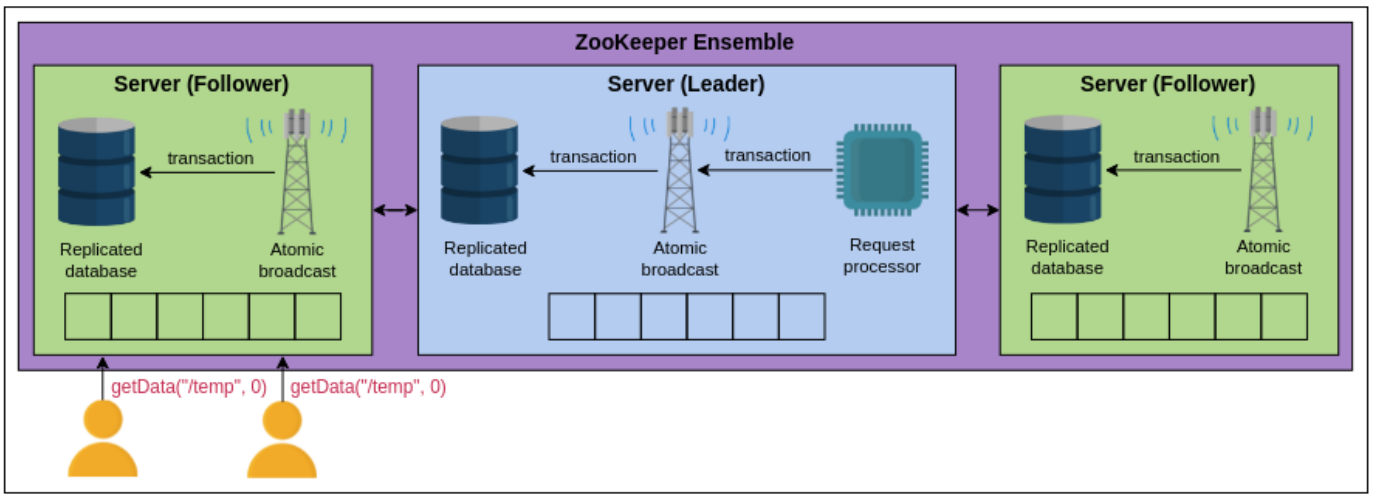
client who has set the watch on the znode is notified by the server with whom it is connected.

By default, ZooKeeper provides an asynchronous transmission of data, which can lead to inconsistent or stale reads. Read operation being fast comes with the disadvantage of showing stale data which is not being yet updated. Not all applications require asynchronous transmission, and ZooKeeper allows us to synchronize the transmission by using the `sync()` method. This `sync()` method ensures that all the operations on the path are updated before `sync()`, and this update is global. This can be done by putting the `sync()` method in the `enQueue()` method of the queue, in both the leader and follower executing it. The leader will ensure that all the followers synchronize so that all the connected clients get the updated data.



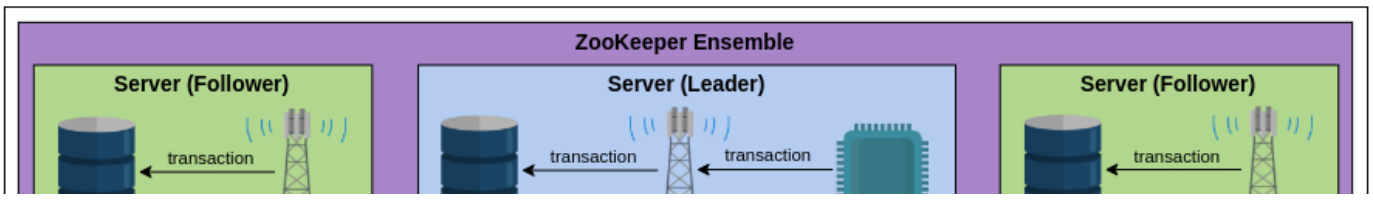
A client comes to a follower with the read request

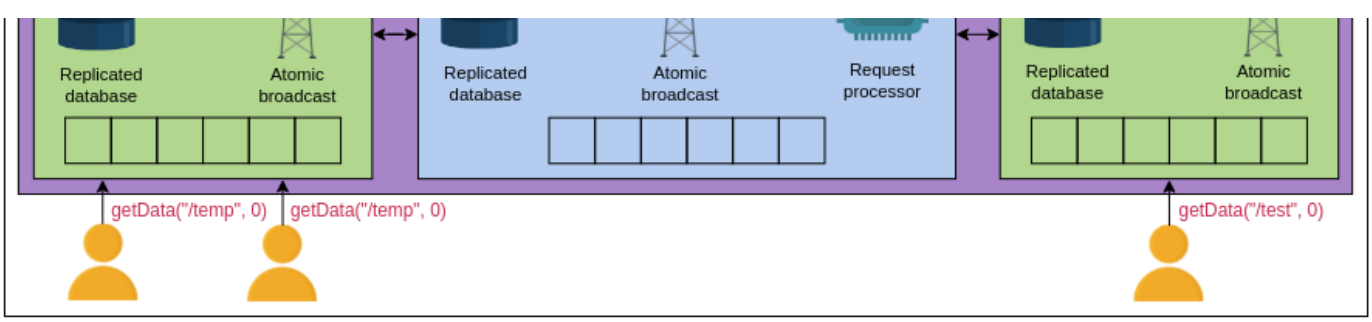
1 of 7



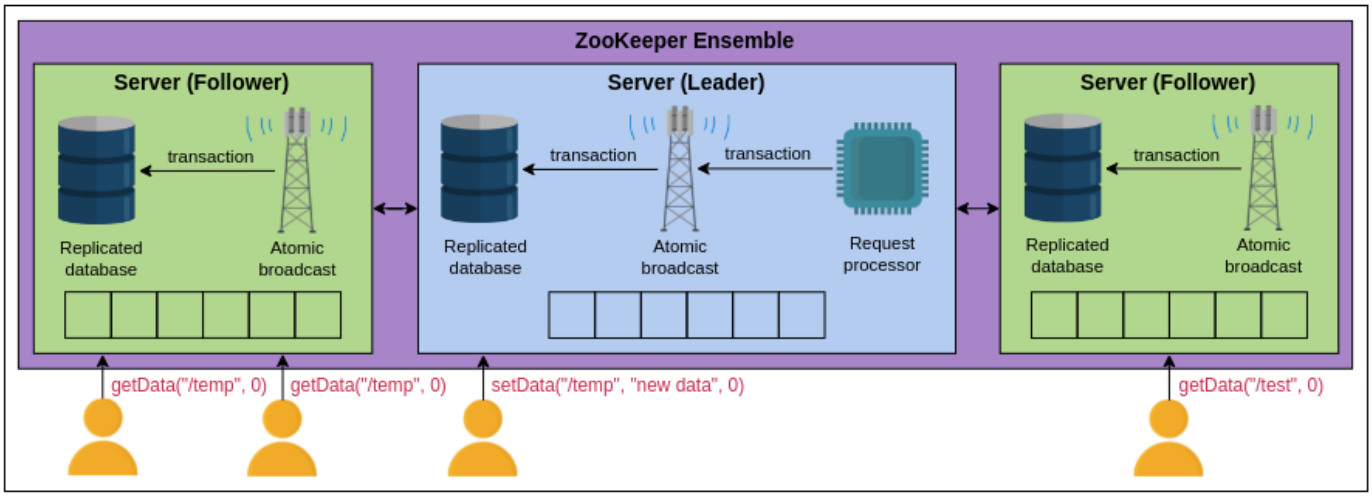
Another client comes to the same follower with a read request on the same znode, which will be executed in parallel

2 of 7

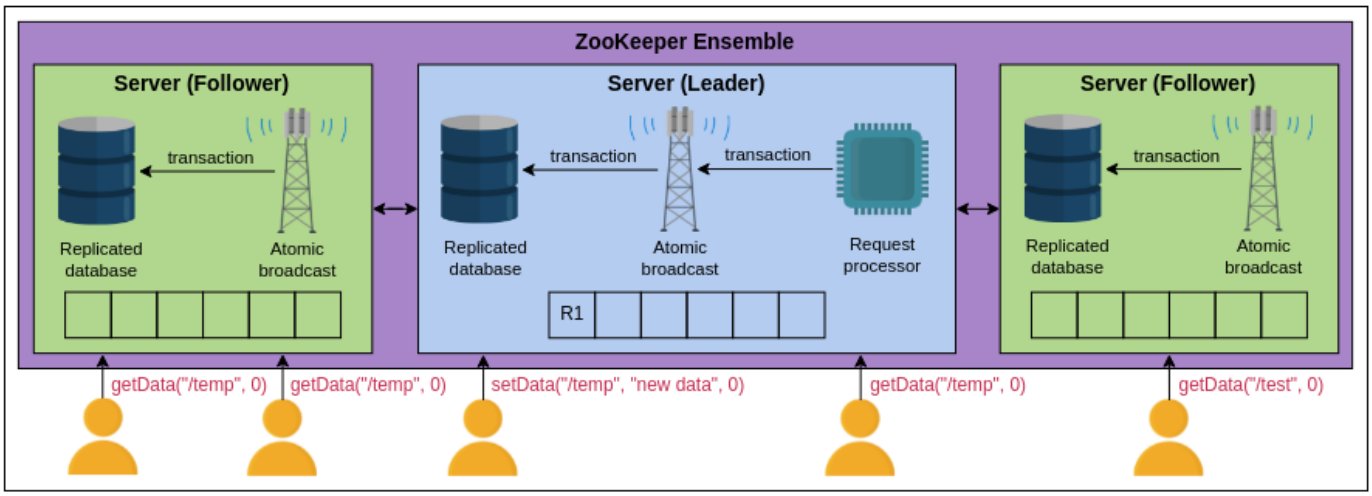




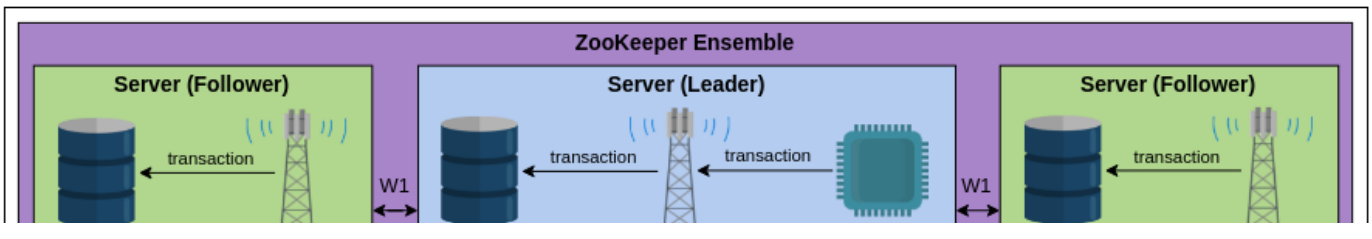
Another client comes to a different follower with the read request on a different zone

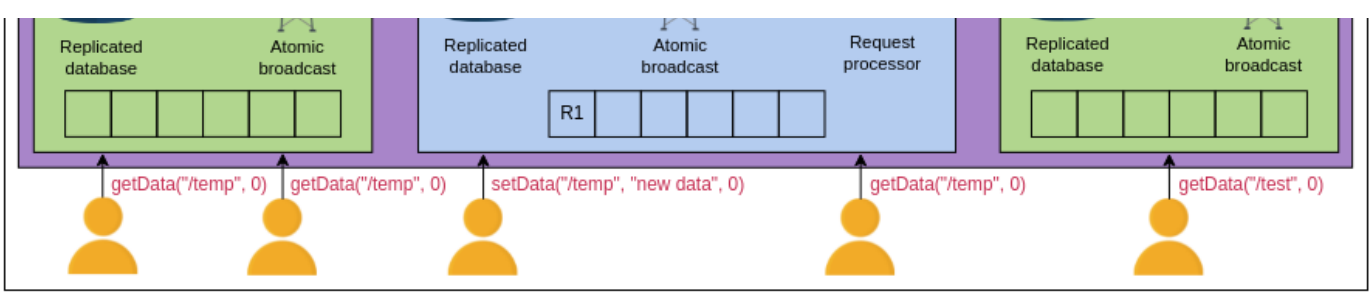


A client comes to a leader with the write request (for /temp) on the data that two clients are currently reading at one follower

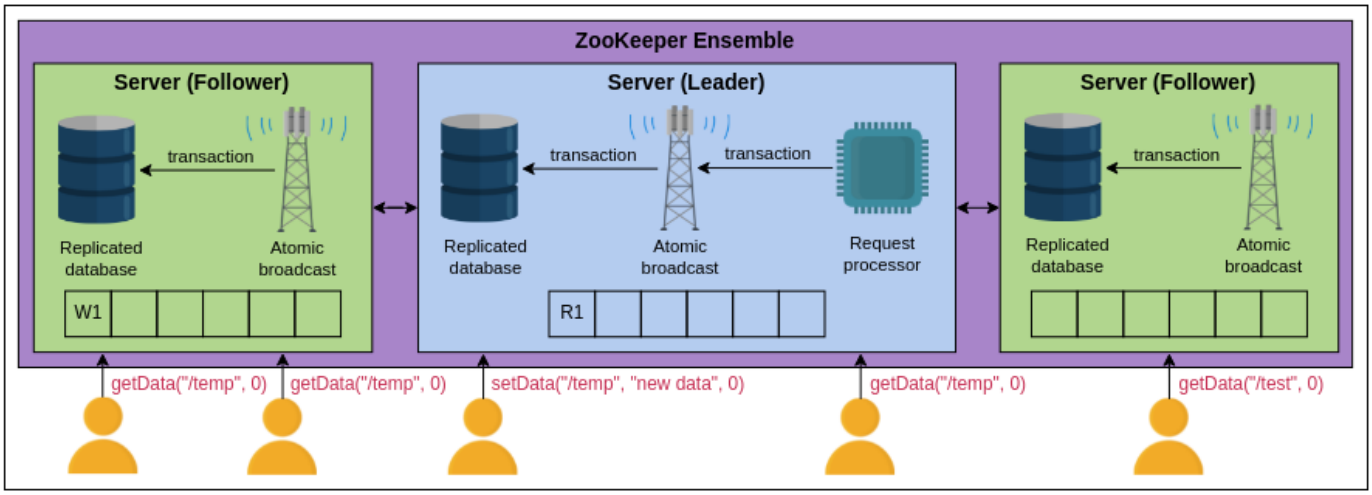


A client comes to the leader with the read request on the same znode, and the request goes into the queue as the write operation is currently in progress





The write request is broadcasted to the followers



The follower who has two clients reading the data on the same znode for which znode the write request came will put the request in the queue, and the other follower will execute it

ZooKeeper servers execute the client's write requests in the FIFO order. Each response to a read request has a `zxid` generated by the server. Even when there is no activity between the client and the server, the client sends the last served `zxid` to that server. In case of disconnectivity from that server, the client is connected to another one. Using the last `zxid`, ZooKeeper ensures that the client gets the same view as the previous server. If the new server is not at the same `zxid` as the client, then it doesn't establish a session with it unless both are not in the same state. ZooKeeper guarantees that the client will be connected to a server with the same recent view.

We have used timeouts to detect whether the client is still connected to one of the servers. If none of the servers receive any response from the client during the session timeout, the leader decides it is a failure. The client library sends heartbeat messages after $\frac{1}{3}$ of the session timeout s, where s is in milliseconds. If the client doesn't get a response from the server in the subsequent $\frac{1}{3}$ (which is $\frac{1}{3} + \frac{1}{3} = \frac{2}{3}$ by now) of the session timeout s, the client connects with another server before the session is expired.

In this lesson, we learned about the design of the ZooKeeper in detail, such as the client API, server, and its interactions. In the next lesson, we'll discuss the primitives of ZooKeeper, which are implemented using the client API.