The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Martin

Patterns of Distributed Systems

Unmesh Joshi

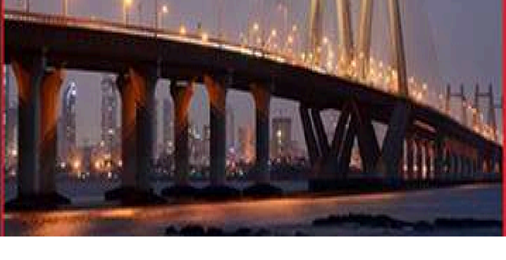
The Addison-Wesley Signature Series

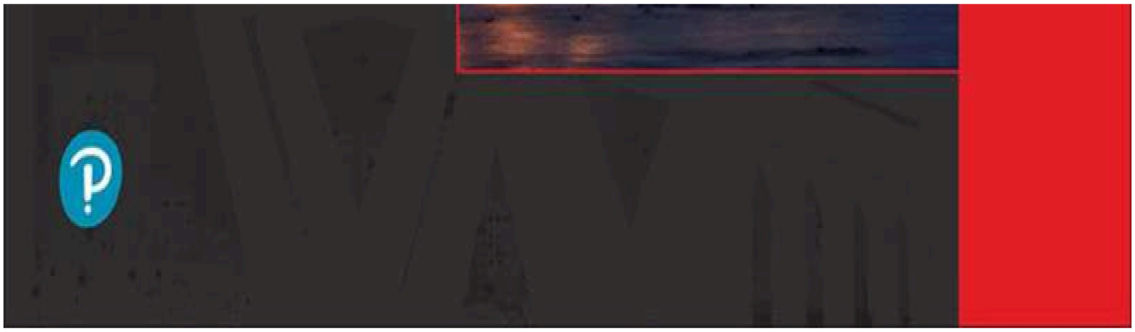
A Martin Fowler Signature Book

Martin

Patterns of Distributed Systems

Unmesh Joshi





Patterns of Distributed Systems

Unmesh Joshi

Addison-Wesley

Contents

Table of Contents

Part I: Narratives

Chapter 1. Why Distribute?

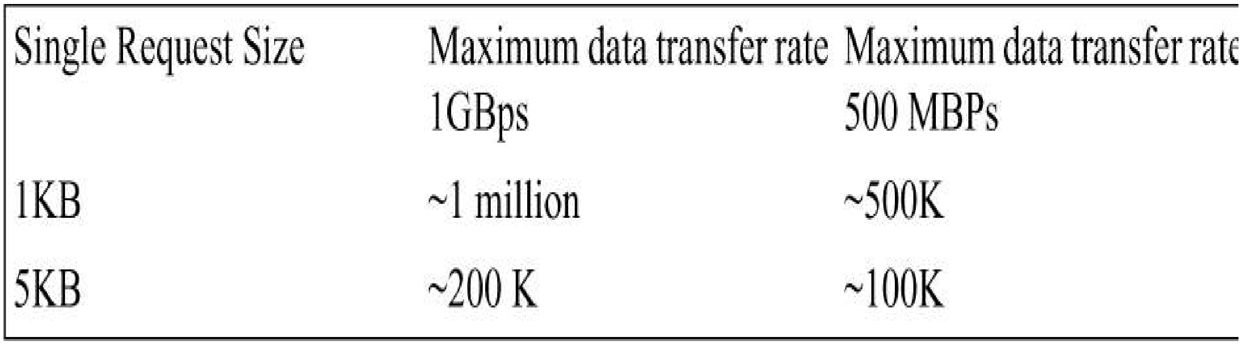
The four fundamental resources

We live in a digital world. Most of what we do is available over the network as a service. Be it ordering our favourate food or managing our finances. All these services run on some servers somewhere. These servers store data and do computations on that data handling user requests over the network. Servers typically wait for user requests, then they read the data stored on the disk into memory and process them using the CPU. CPU, Memory, Network and Disks are the four fundamental physical resources which any computation needs.

Consider a typical retail application exposed as a networked service. Users can add items to the shopping cart and buy them. Users also view their order, and can query their past orders. How many user requests will a single server be able to process? There are many factors based on the specific type of the application, but the upper bound will always be determined by the capacity of these four resources.

Lets start with the network. The network bandwidth decides the maximum limit on how much data can be transferred over the network at any given time. Consider a network bandwidth of 1Gbps. If the application is writing or reading 1KB records, the maximum number of requests that the network can support is 125000. If the records are 5KB in size, the number of requests which can be passed over the network is 25000.

The disks have a limit on the amount of data they can transfer. Following are some example numbers with typical disk bandwidths.



Single Request Size	Maximum data transfer rate 1GBps	Maximum data transfer rate 500 MBPs
1KB	~1 million	~500K
5KB	~200 K	~100K

This is a raw hardware limitation. But in practice, there is some software component which handles the writes and reads. The software component needs to handle issues like concurrent read/writes or transactions, which further limits the number of read/write requests which can be processed on a single server.

Disk and Network are the input/output devices. But to do any work with data read from these needs CPU cycles to be consumed. So with 100K requests if the CPU is at 100% of the capacity, any more requests will be waiting to be processed for their share of CPU.

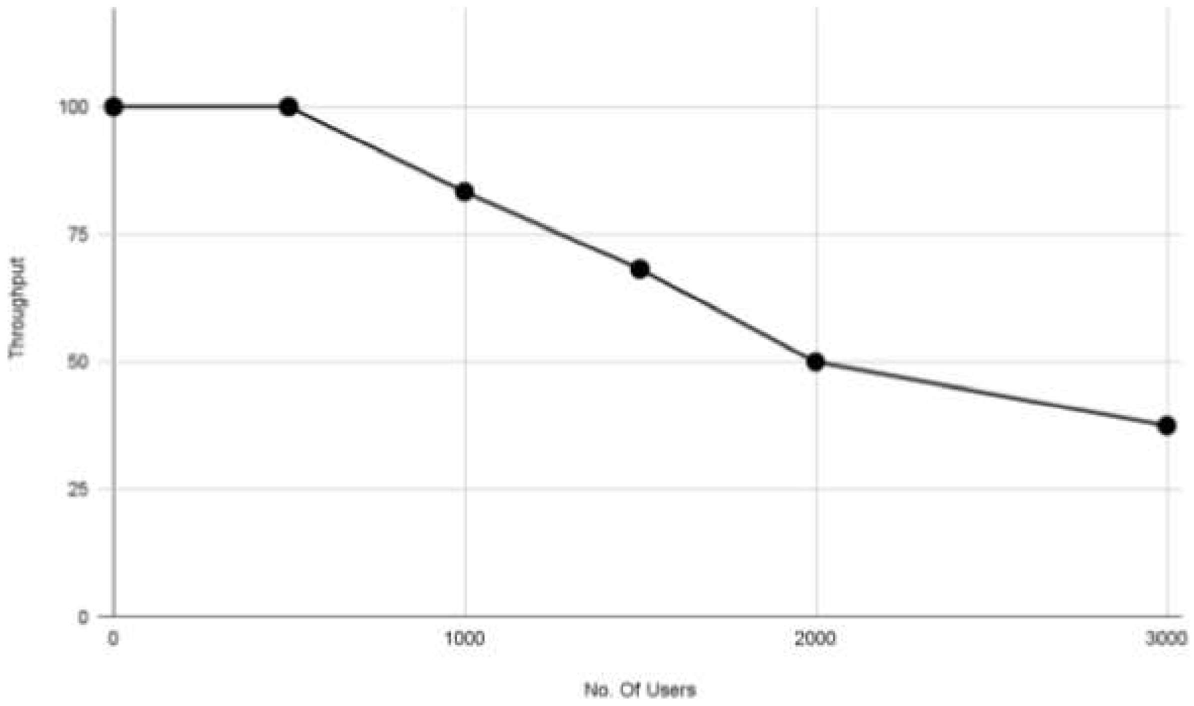
The fourth factor is that of memory. Servers load data in memory to process it. The data from requests over the network are loaded in memory to further process it. The data from storage is also loaded in memory to process it. Let’s say there is 1TB of storage, the memory will typically be in GBs, so at a time only part of the data can be loaded in memory. The whole purpose of different kinds of storage engines is to reduce the need to load all the data in the memory to process. Storage engines use different data structures and maintain indexes to quickly locate the specific data items on the disk and pick and load only those for processing. But depending on the type of the request, more data might be loaded in memory. For example, if all the users are searching for books they ordered in the last year, they will need to scan through more and more data, needing more memory. If the memory is full, again the requests need to wait for their share.

One common problem when these resources reach their physical limit is that of queuing. More requests will need to wait to be processed. This in effect has an adverse effect on the server’s ability to process user requests.

Queuing and its impact on system throughput

The disk, network, cpu and memory put upper bound on the number of requests which can be processed. If the number of requests the application needs to process goes above this upper bound, the requests start getting queued for their share of network, disk, cpu or memory. The queuing then increases the time it takes to process any requests.

The effect of reaching the resource limits is observed on overall throughput of the system as following.



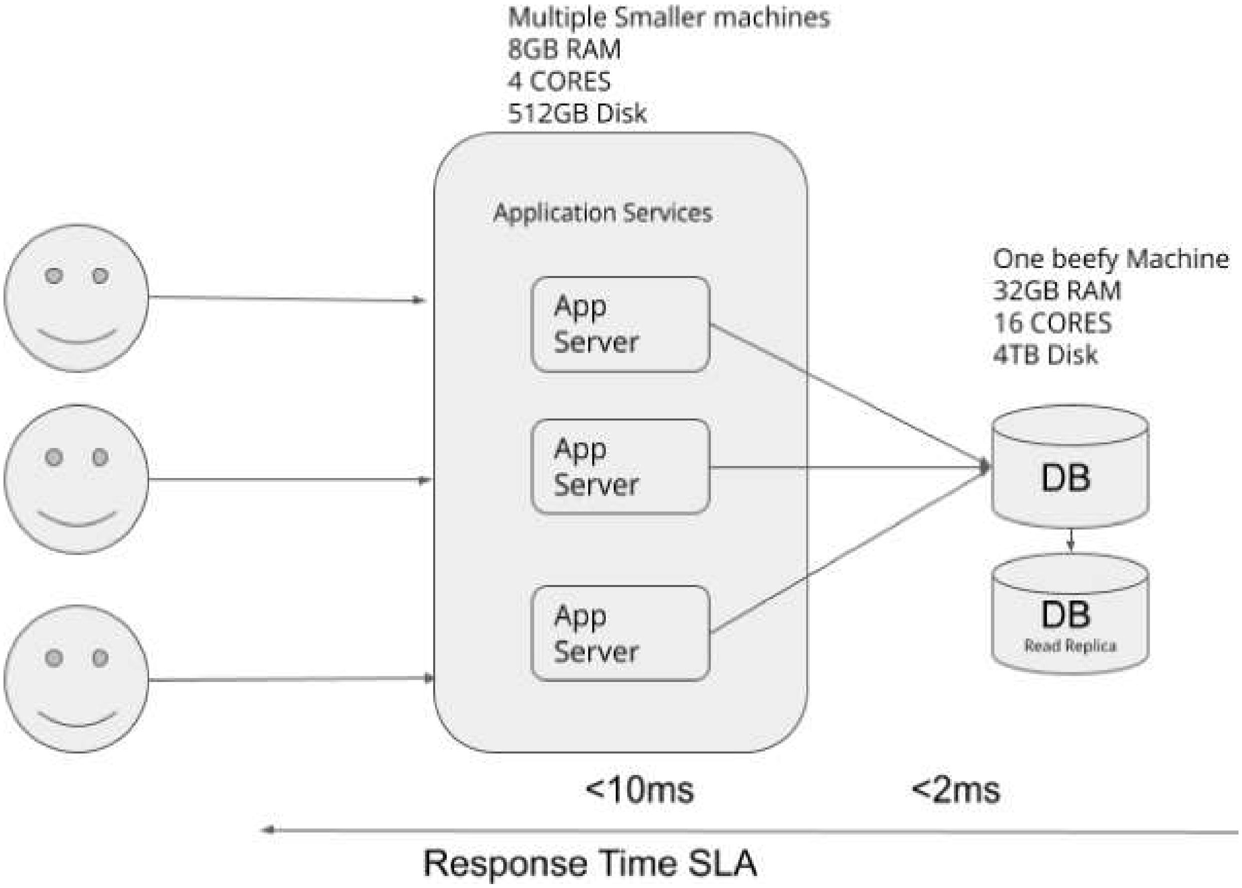
This is problematic to the end users. Because when they expect the system to serve more and more users, the system actually performance starts degrading.

The only way to make sure that the requests can be served properly is to divide and process them on multiple servers. This allows using physically separated CPUs, network, memory and disks for processing user requests. In the above example, the workload needs to be partitioned in such a way that each server serves about 500 requests.

Partitioning - Divide and Conquer

Separate business logic and data layer

One common way to divide the architectures is as following. The architecture has two parts, a stateless part exposing functionality to the end user. This can be a web application or more commonly a web api serving user facing applications. The second part, the stateful part, is managed by a database. When user load increases the stateless services are scaled horizontally. This allows serving more and more user requests connections. The business logic executed on the data is done separately on different servers



This architecture works fine provided two basic assumptions hold true.

- The database can serve a request from the stateless services in less than a few milliseconds.

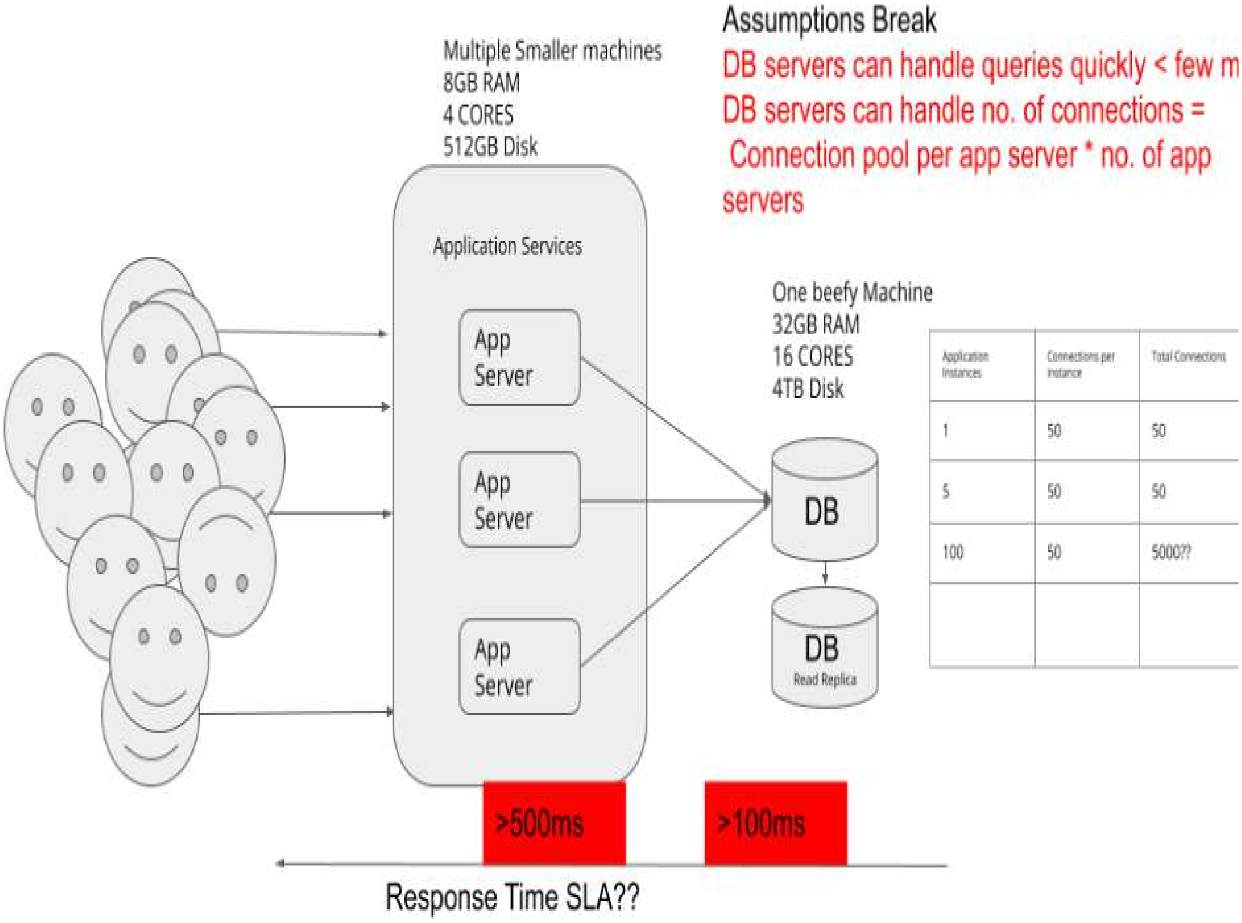
- The database can handle all the connections from multiple stateless service instances. This applications typically work around this constraint by adding caching layers to make sure that not all the request need to go to the database.

This architecture works very well, if most of the users can be served from caches put at different layers in the architecture. It makes sure that, out of all the requests, only a few requests need to reach the database layer. As nicely stated by Roy Fielding in his thesis on REST, "best application performance is obtained by not using the network". But caching does not work always. When most requests are writing data there is obviously no use of caching. With hyper personalized applications needing to always show latest information, the use of caching is limited. As more and more users start using the services, the assumptions start breaking down. It is caused by two reasons.

- The size of the data grows, from few terabytes to several hundred of terabytes to petabytes
- More and more people need to access and process that data

So the simple architecture shown above starts breaking down. The reasons it starts breaking down is again because of the physical limits of the four fundamental resources.

The impact on the classic architecture then looks as following:



Assumptions Break

DB servers can handle queries quickly < few m

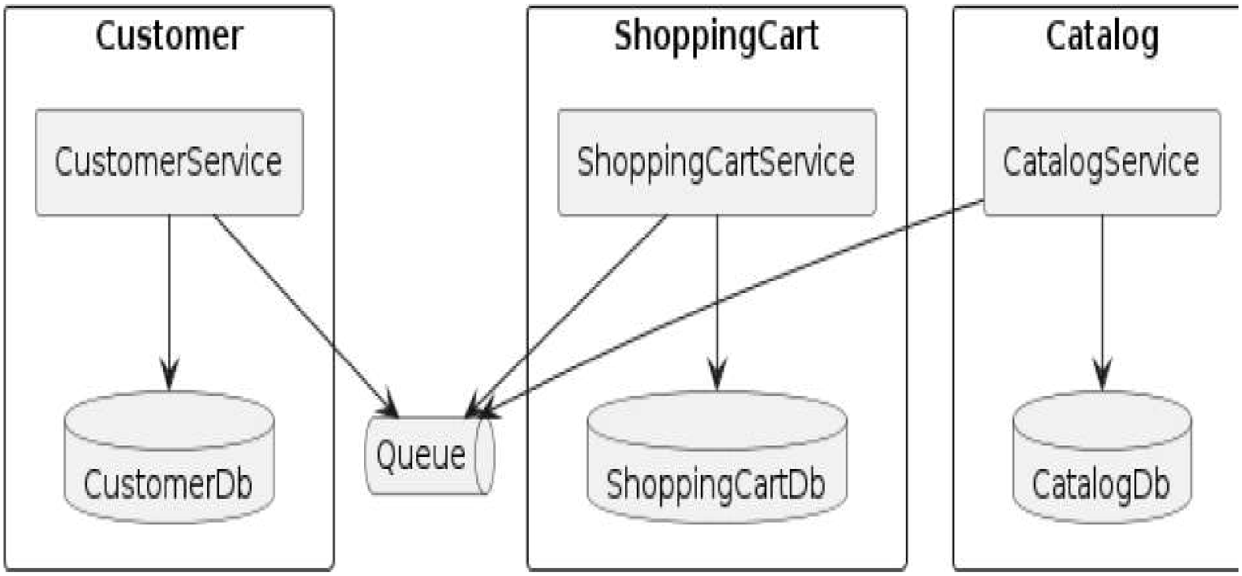
DB servers can handle no. of connections = Connection pool per app server * no. of app servers

Application Instances	Connections per instance	Total Connections
1	50	50
5	50	50
100	50	5000??

Partitioning by Domain

One way to work around these limits is to partition the application following the domain boundaries. The popular architectural style today is that of Microservices [bib-microservices] Microservices architecture encourages partitioning the architecture following the guidance from Domain Driven Design [bib-ddd].

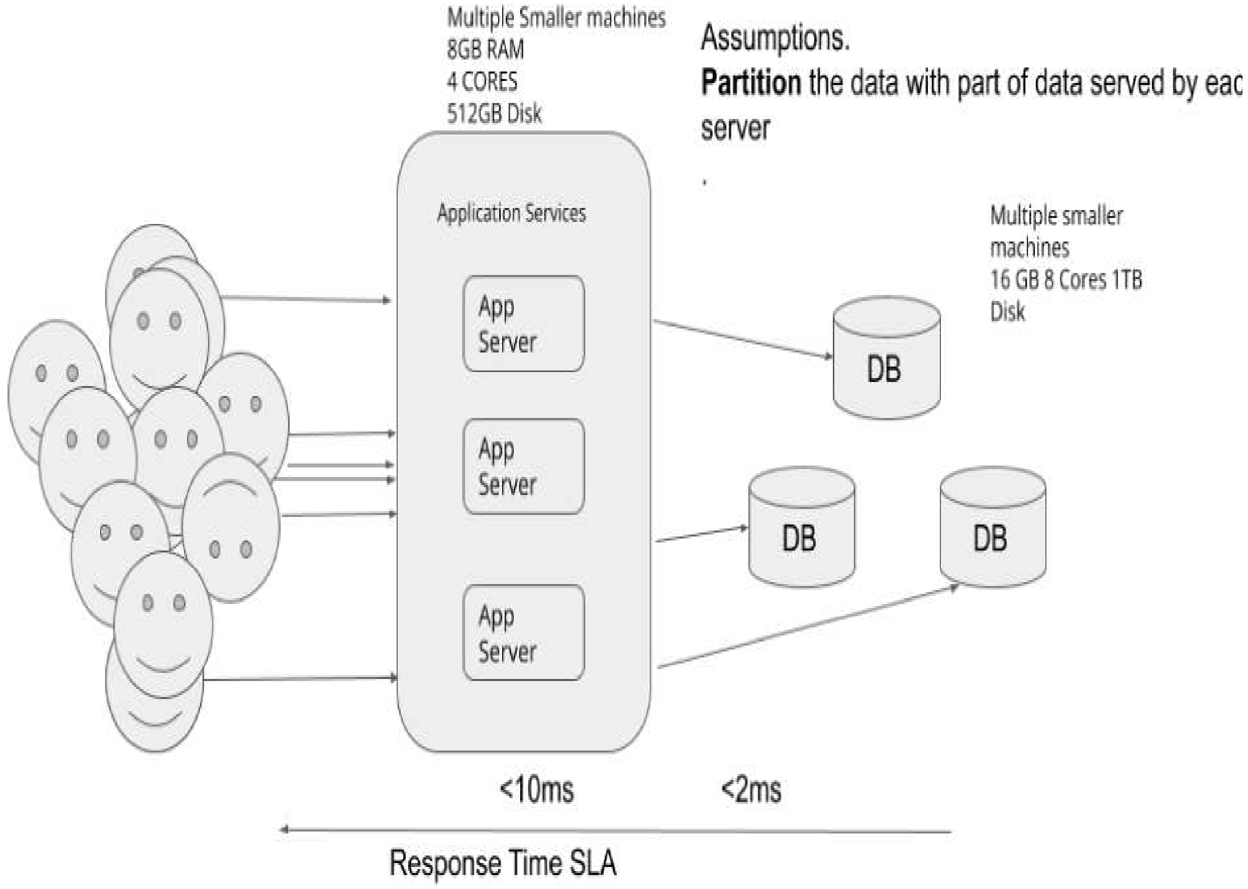
For example, a typical retail domain can have software systems created following the broader areas of interests as following.



While this architecture is also sometimes termed as a ‘distributed system’ it is not something that will be covered in this book. Even with this architecture, some shared infrastructure components or services which need to deal with lots of data faces similar issues as discussed in the previous section. That is one of the reasons for the popularity of products like Kafka [bib-kafka]. Same is true for some fundamental domain services like customer data management getting used by every other system in the organization. The size of data and number of requests these services need to process start showing similar symptoms. This is one of the reasons for popularity of distributed [nosql] [bib-nosql] databases like Cassandra [bib-cassandra] getting commonly used.

Partioning Data

One common reason for all the issues discussed so far is size of the data and number of requests needing to process that data. When software systems start facing issues because the physical limits are reached,the only way to make sure that the requests can be served properlyis to divide the data such that requests are processed on multiple servers. This allows using physically separated CPUs, network, memory and disks for processing requests on smaller portions of the data.

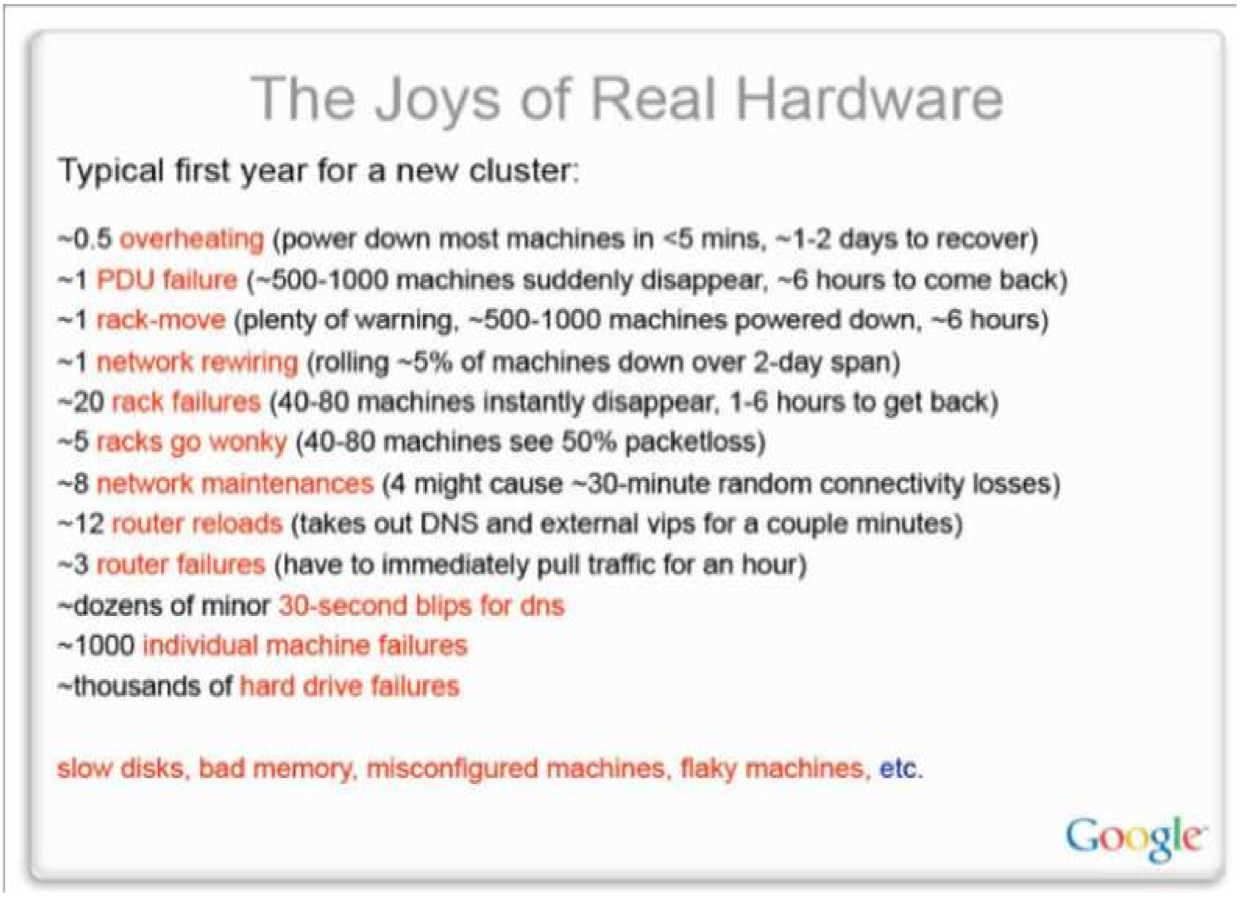


Failures become a very important concern, when we are dealing with data. A separate instance can not be as easily creates on a random server. It needs a lot of care to make sure consistency the servers start in a correct state. Most of what is discussed in this book is about this kind of systems.

A look at Failures

When there are multiple machines comprising of multiple disk drives network iterconnects, processors and memory units, the probability of something failing becomes an important concern. To understand why this is the case, consider an example of probability of a hard disk failing. If a disk disk might fail once in 1000 days, the probability of it failing on any given day is 1/1000. Probably not a big concern. But if we have 1000 disks, the probability of any one of that going down on a given day is 1. That means every day any one hard disk might fail. If the partitioned data is getting served from that disk, it will be unavailable until the disk is recovered.

To get some idea of the kind of failures, take a look at following details from Jeaf Dean’s talk [bib-jeaf-dean-google-talk] delivered in 2009 have some interesting failure numbers observed in Google’s data centers. Even if the numbers are from 2009, they are good representative numbers.



The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

So just distributing data across cluster nodes is often not enough. The failures need to be masked as well. From end user’s perspective the system as a whole should remain functional, even if part of it is facing some failures.

Replication - Masking failures

To make sure that the data is available even when different kinds of failures happen, the data is replicated on multiple machines. In case of a failure clients can connect to a server which holds a copy of the data. But this puts the responsibility of masking these failures on the software thats handling user requests. The software needs to make sure that it detects failures, makes sure that inconsistencies are not visible to the users. To be able to successfully mask these failures, it is important to understand what errors a software system experiences because of various hardware failures.

Common failures experienced by software systems

There are four common problems which the software systems experience and needs to mask from the users of the system.

Process Crash

A software process can crash at any time because of various reasons. It can crash because of some hardware failure. But can crash for reasons like unhandled exceptions in the code. In the containerized or cloud environments, the monitoring software can restart a process.

Network Delay

TCP/IP network protocol is asynchronous by nature. It does not give any guaranteed upper bound on delay in delivery of the messages. This creates a challenge for software processes commnicating over TCP/IP. They need to figure out how much to wait for responses from other processes. If they do not get response in time, whether they should retry or consider the other process as failed is the question to answer.

Process Pause

A process execution can pause at any given point in time. People familiar with garbage collecting languages like Java are familiar with the GC pause. In the worst case scenario, this pause can be as long as tens of sends. Other processes then need to figure out if this process has failed. The tricker problem happens when the paused process comes out of the pause and start sending messages to other processes, what should other processes do? If they had marked the paused process as failed, should they ignore the messages or process them?

Unsynchronized Clocks

The clocks in machines typically use a quartz crystal. The clock ticks are governed by the oscillation of this quartz crystal. The oscillation frequency can change because of temprature changes, or vibrations and cause the clock on a given machine to either go slower or faster. This causes clocks on two different machines to have totally different times. When processes need to order messages or figure out which data is saved ahead of other, they can not then rely on the system timestamp.

Defining the term "Distributed Systems"

We can now define the term distributed systems as used in this book. The software systems which store data, runs as multiple processes across multiple servers to coordinate and have an agreement on the state of that data. There are various kinds of distributed systems. But the ones we will focus on, have following characteristics.

The failure assumption here is that of ‘fail-stop’, meaning that when the processes fail, they stop functioning and possibly restart. But they will never process requests or respond in arbitary manner. The failures are not ‘byzantine failures’.

- They run on multiple processes possibly across physically distant locations
- They manage data, so are inherently stateful systems
- They communicate by message passing
- They tolerate partial failures. So from end users perspective, the system is functional even if some of these processes fail.

The patterns approach

There are many nice books, like the book by Nancy Lynch [bib-distrib-algorithms-nancy-lynch] which discuss the theory of distributed systems. There are few good books like Designing Data Intensive Application [bib-intensive-data-book] which describe implementation of most mainstream systems used today. These books are excellent in their own right, but practicing professionals need some way to get an intuitive understanding of these systems, which is detailed and specific enough to be able to understand real code, but generic enough to be applicable to broad range of systems. Patterns approach provides a nice way out.

[patterns] [bib-patterns] is an approach introduced by an architect Christopher Alexander in his book the patterns language. The approach was adopted in the software world and popularized by the book famously known by the name Gang Of Four [bib-gang_of_four] book.

Patterns by their very nature provide a way to describe specific problems which the software systems face and a concrete enough solution structure to be able to show real code. Patterns have names. Having good names with specific enough code level details is very powerful.

One of the key requirements for finding the patterns is to study actual code bases. To find and document patterns in this book, code base of various systems was studied. Kafka [bib-kafka], Cassandra [bib-cassandra], MongoDB [bib-mongodb], [pulsar] [bib-pulsar] , etcd [bib-etcd], Zookeeper [bib-zookeeper], CockroachDB [bib-cockroachdb], YugabyteDB [bib-yugabyte], Akka [bib-akka], JGroups [bib-jgroups] to name a few.

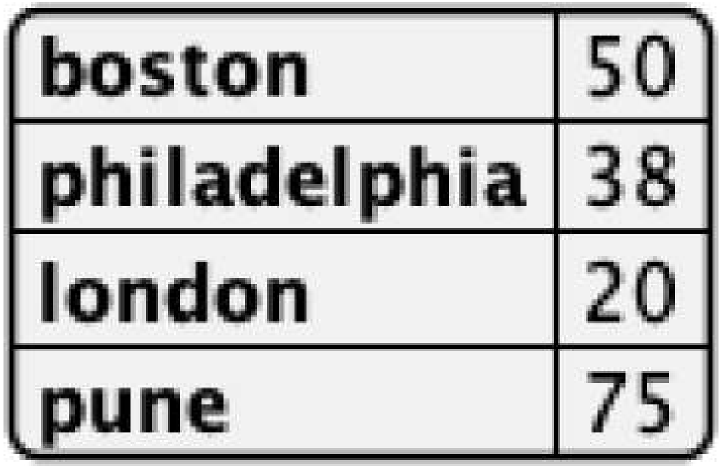
The next chapter takes a tour of most of the patterns and shows how they link together.

Chapter 2. Overview of the Patterns

by Unmesh Joshi and Martin Fowler

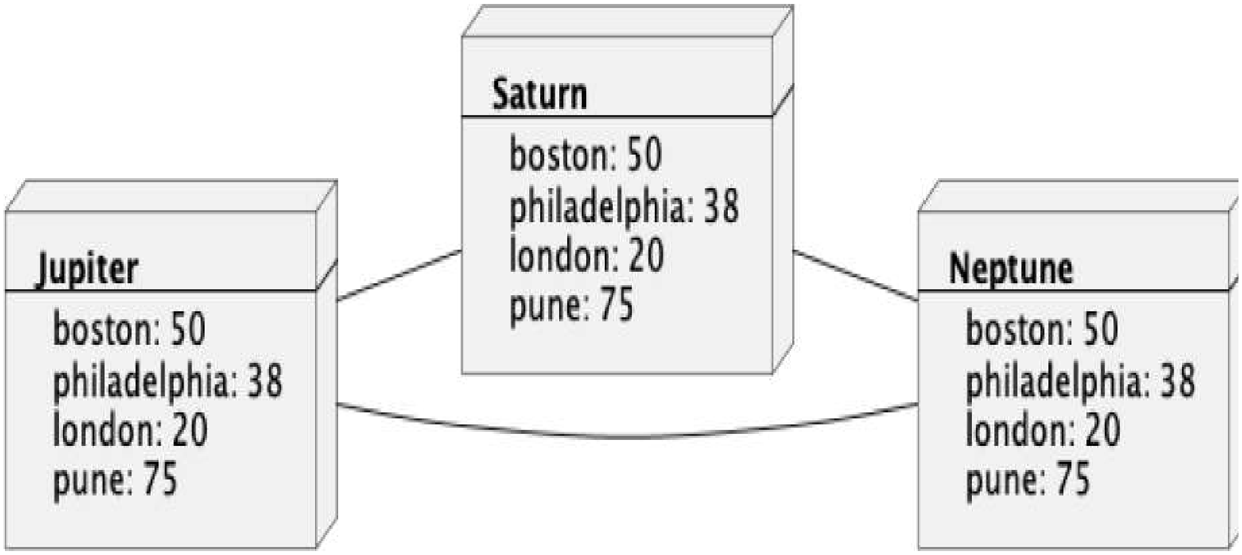
As discussed in the last chapter, distributing data means at least one of two things: partitioning and replication. To start our journey through the patterns in this book, we’ll focus on replication first

Imagine a very minimal data record, that captures how many widgets we have in three locations.



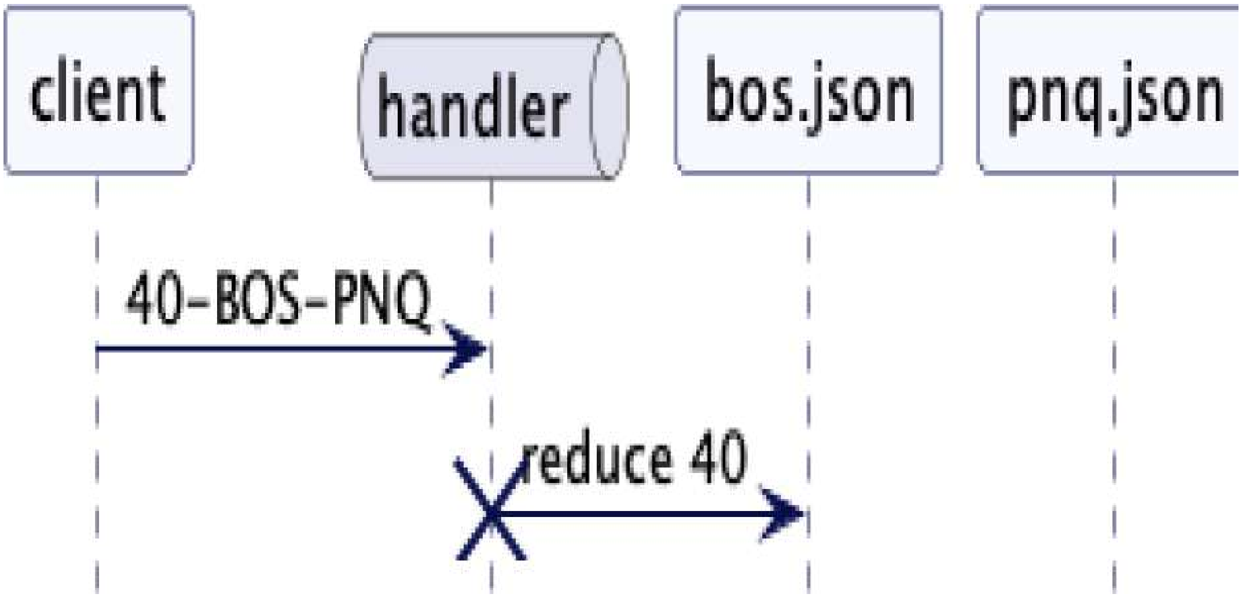
boston	50
philadelphia	38
london	20
pune	75

We replicate it on three nodes: Jupiter, Saturn, and Neptune

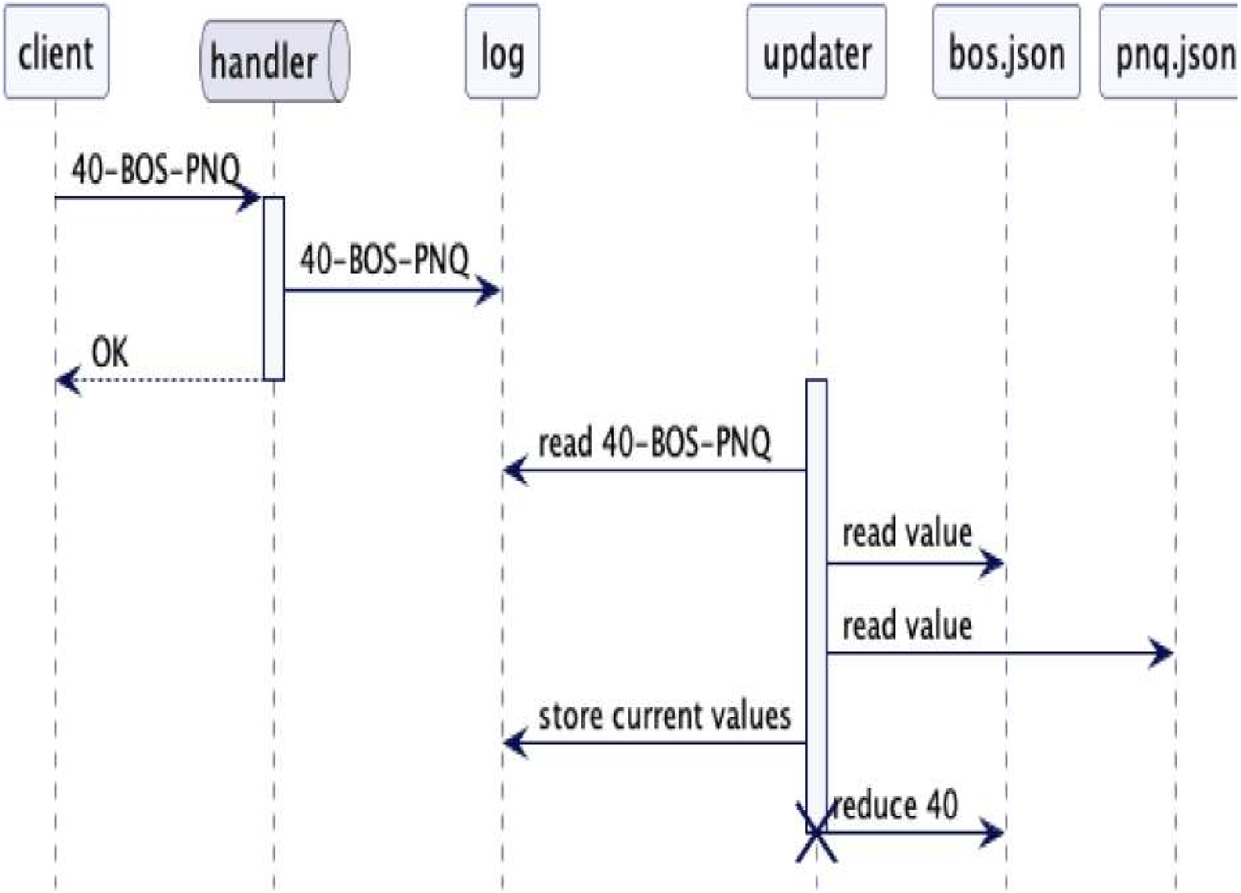


Keeping data resilient on a single server

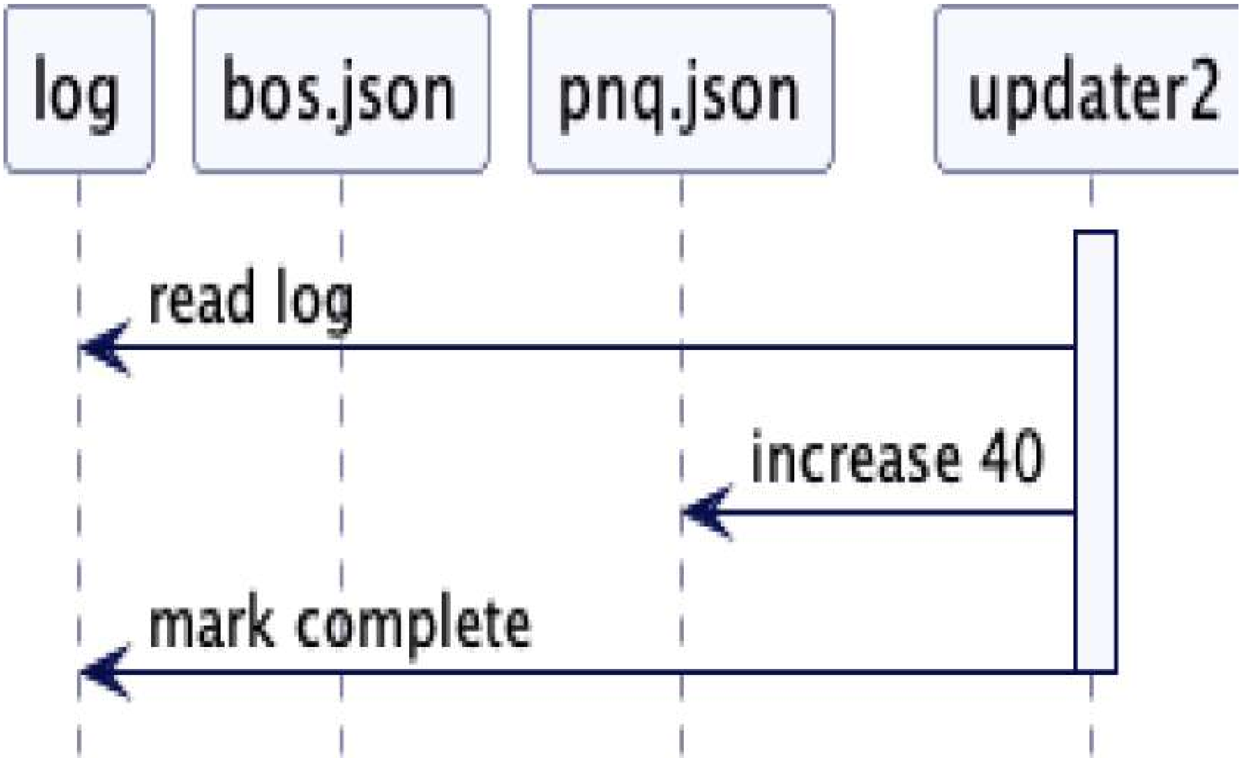
The first area of potential inconsistency appears with no distribution at all. Consider a case where the data for Boston, London, and Pune are held on different files. In this case performing a transfer of 40 widgets means changing bos.json to reduce its count to 10 and changing pnq.json to increase its count to 115. But what happens should Neptune crash after changing Boston's file but before updating Pune's? In that case we would have inconsistent data, destroying 40 widgets.



An effective solution to this is *Write-Ahead Log*. With this, the message handler first writes all the information about the required update to a log file. This is a single write, so is simple to ensure it’s done atomically. Once the write is done, the handler can acknowledge to its caller that it has handled the request. Then the handler, or other component, can read the log entry and carry out the updates to the underlying files.



Should Neptune crash after updating Boston, the log should contain enough information for Neptune to figure out what happened when it restarts and restore the data to a consistent state. (In this case it would store the previous values in the log before any updates are made to the data file.)

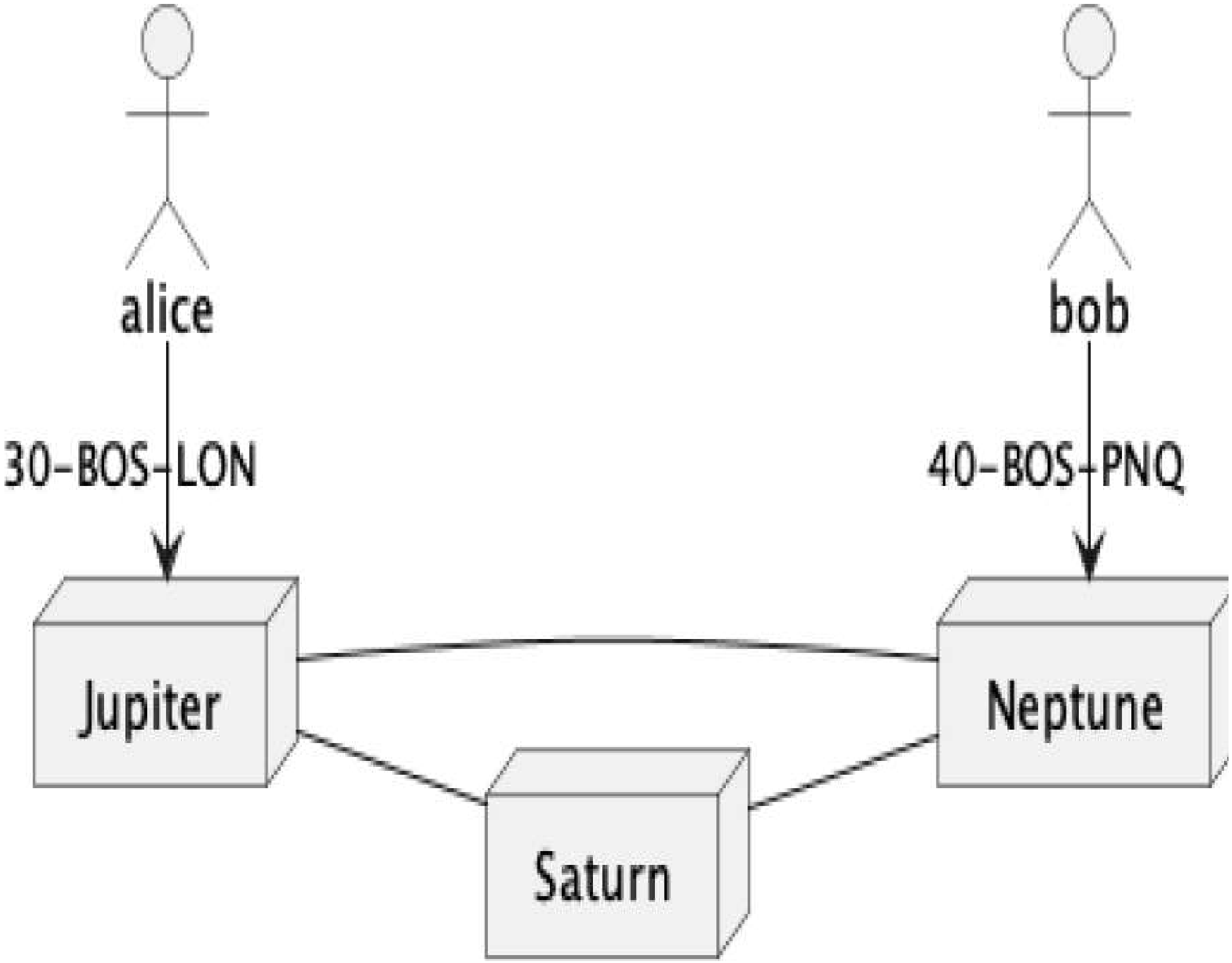


The log gives us resilience because, given a known prior state, the linear sequence of changes determines the state after the log is executed. This property is important for resilience here, but as we'll see, it's also very valuable for replication too. If multiple nodes start at the same state, and they all play the same log entries, we know they will end up at the same state too.

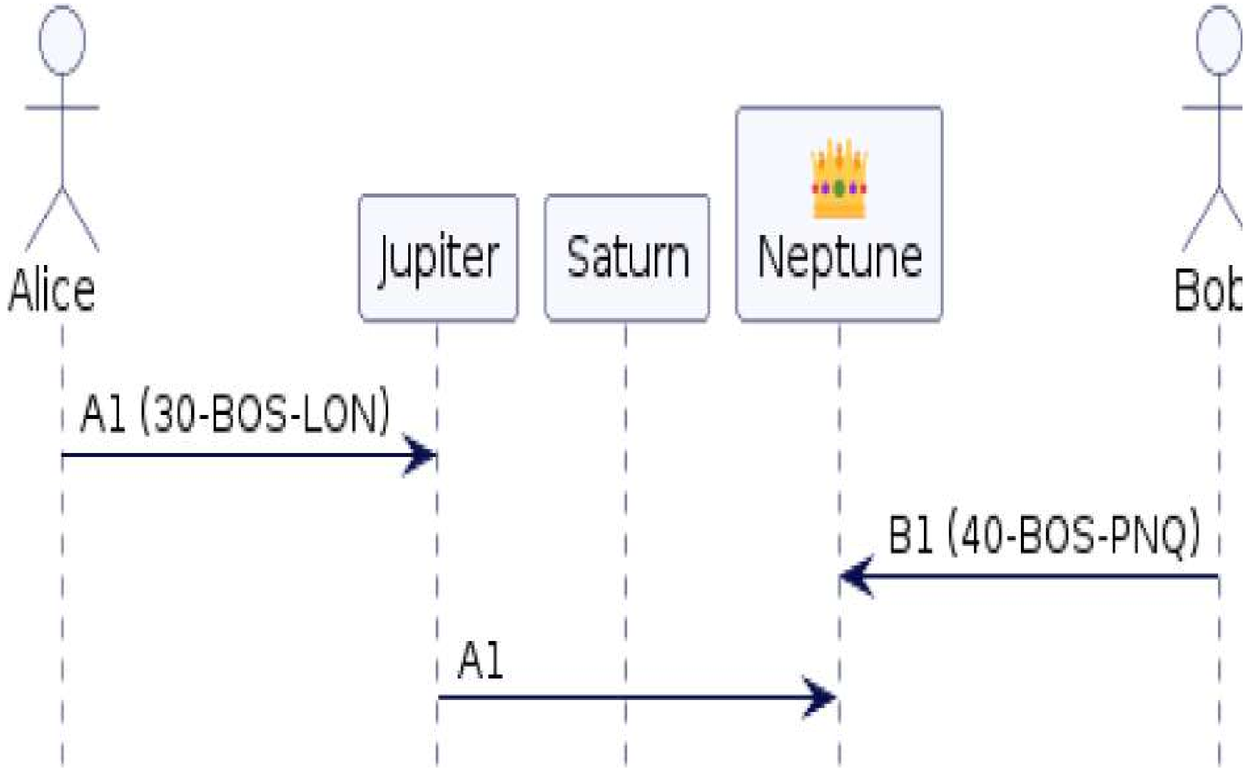
Most programmers don't use a Write-Ahead Log explicitly like this, but use one implicitly all the time, since databases use a Write-Ahead Log to implement transactions.

Competing Updates

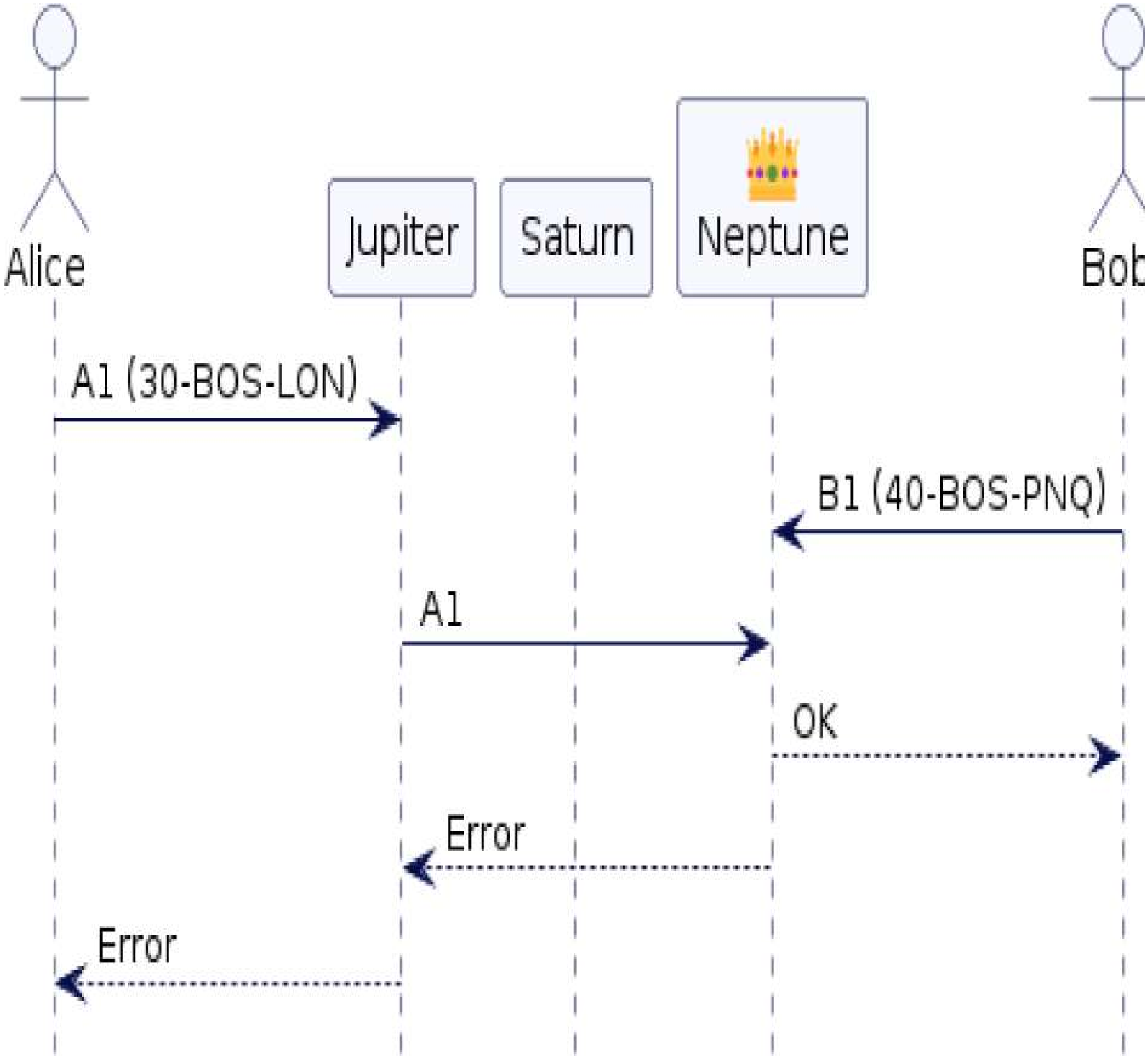
Lets now consider if two different users, Alice and Bob are connecting to two different cluster nodes to execute their requests. Alice wants to move 30 widgets from Boston to London, while Bob wants to move 40 widgets from Boston to Pune.



How should the cluster resolve this? We can’t have any node just decide to do an update because we’d quickly run into inconsistentcy hell as we try to figure out how to get boston to store antimatter widgets. One of the most straightforward approaches is *Leader and Followers*, where one of the nodes is marked as the leader, and the others are considered followers. In this situation the leader handles all updates, and broadcasts those updates to the followers. Let’s say Neptune is the leader in this cluster, then Jupiter will forward Alice’s A1 request to Neptune.

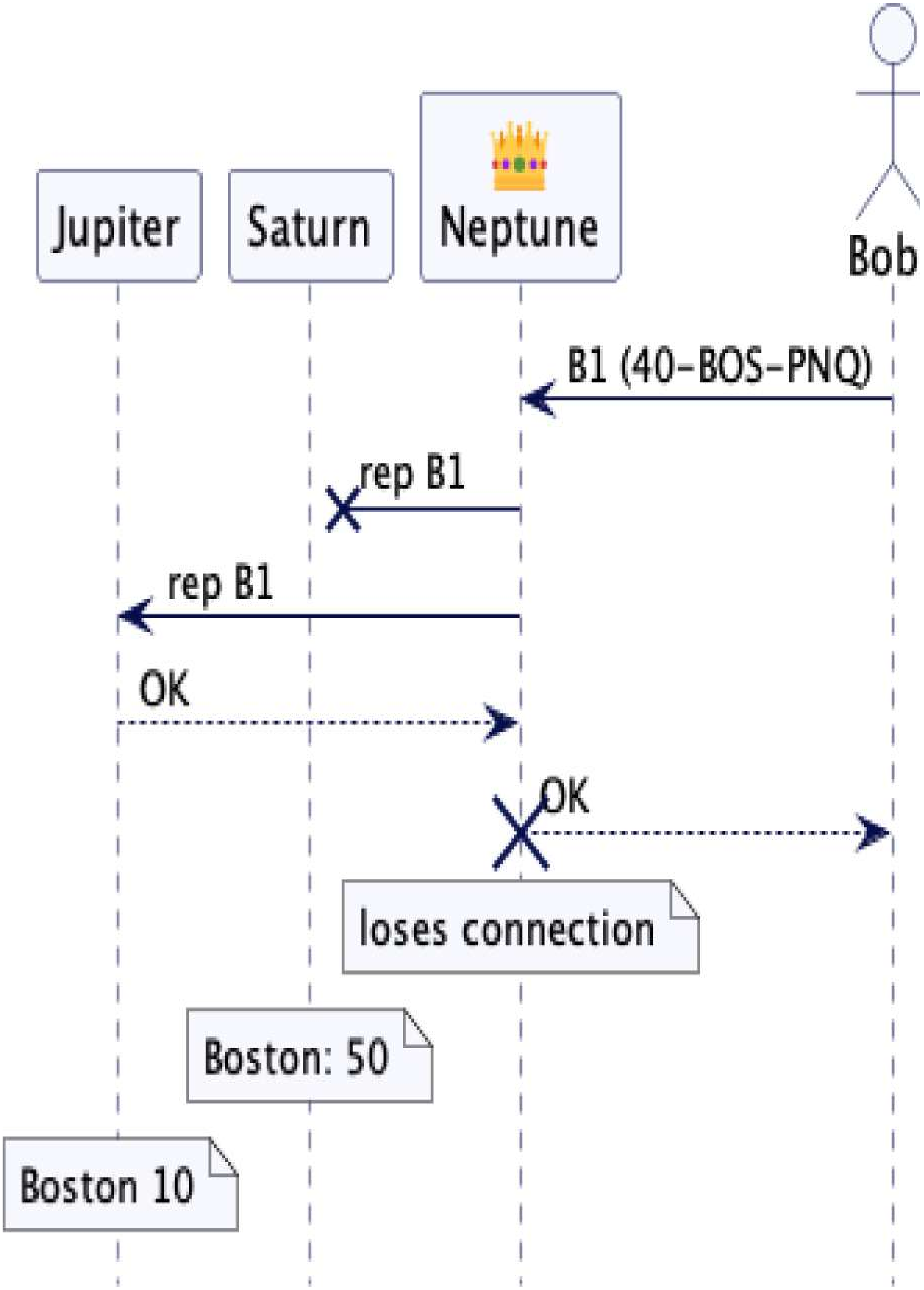


Neptune now gets both update requests, so it has the sole discretion as to how to deal with them. It can process the first one it receives (Bob's B1) and reject A1



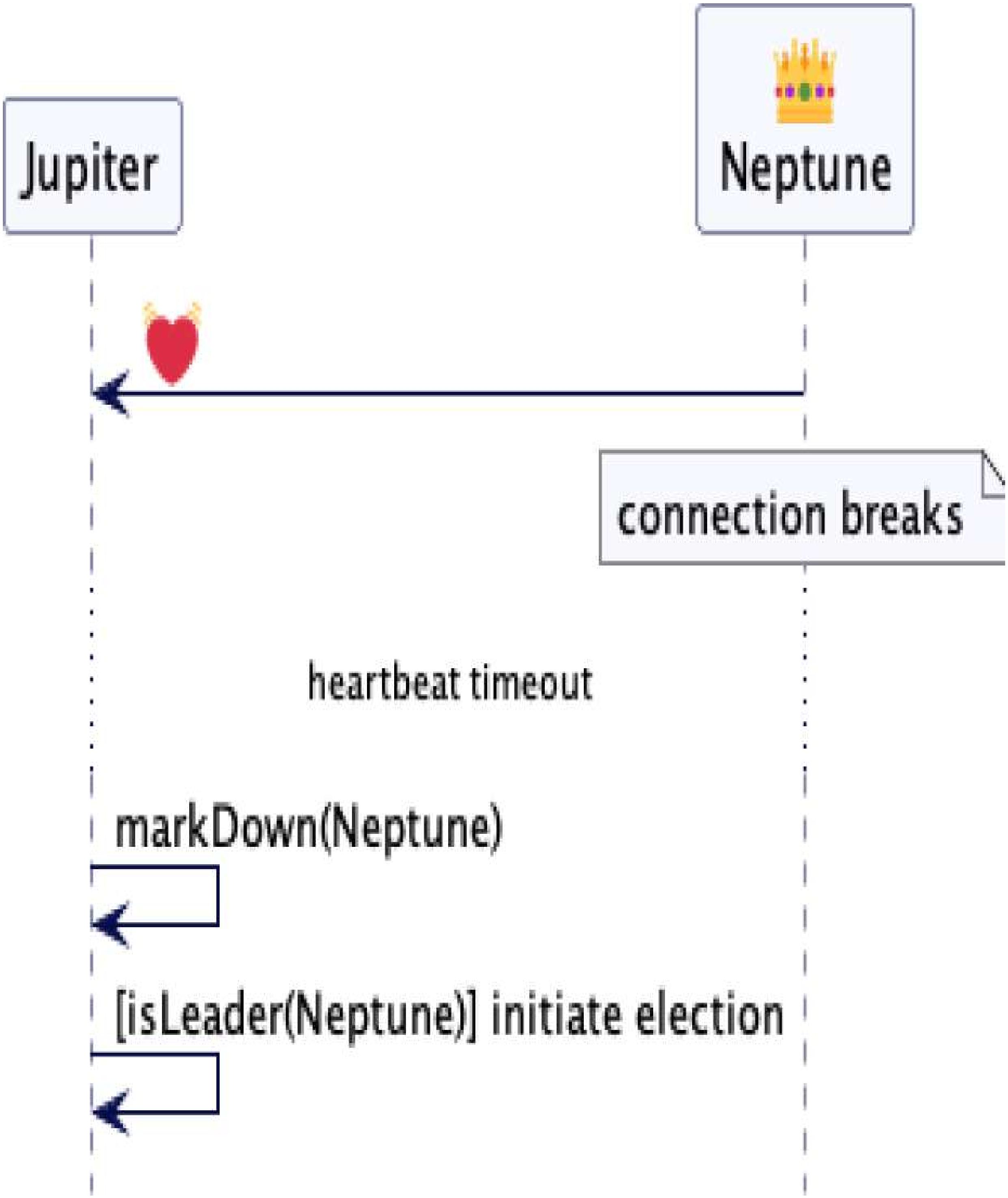
Dealing with the leader failing

That's what happens most of the time, when all goes well. But the point of getting a distributed system to work is what happens when things don't go well. Here's a different case, Neptune receives B1, sends out its replication messages. But is unable to contact Saturn. It could replicate only to Jupiter. At this point it loses all connectivity with the other two nodes. This leaves Jupiter and Saturn connected together, but disconnected from their leader.

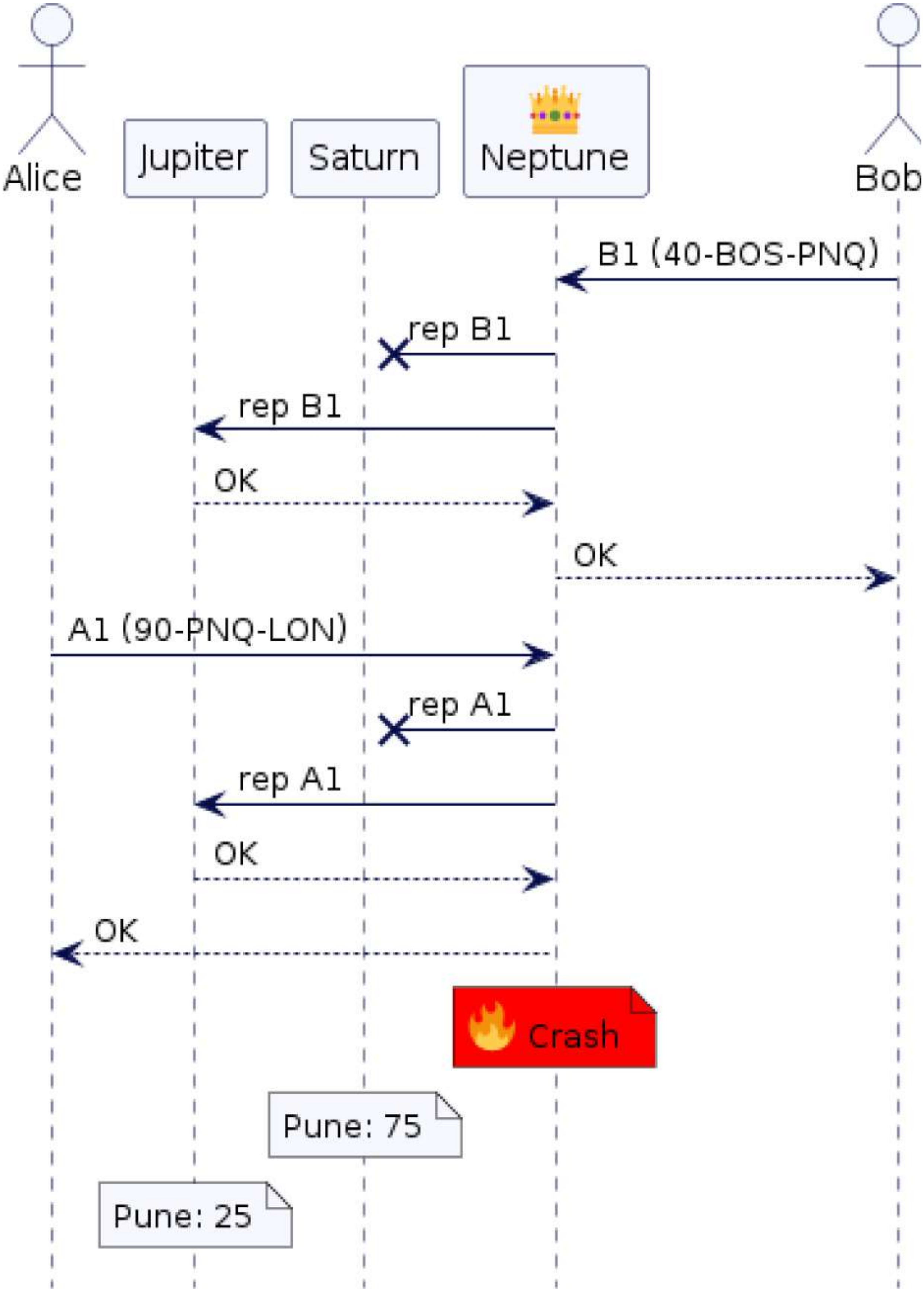

Jupiter
Saturn
Neptune
Bob
B1 (40-BOS-PNQ)
rep B1
rep B1
OK
OK
loses connection
Boston: 50
Boston 10

So now what do these nodes do? For a start, how do they even find out what’s broken? Neptune can’t send Jupiter and Saturn a message saying the connection is broken... because the connection is broken. Nodes need a way to find out when connections to their colleagues break, they do this with a *HeartBeat*. Or more strictly they this with the absence of a heartbeat.

A heartbeat is a regular message sent between nodes, just to indicate they are alive and communicating. If Saturn doesn’t receive a heartbeat from Neptune for a period of time, Saturn marks Neptune as down. Since Neptune is the leader, Saturn now calls for an election for a new leader.

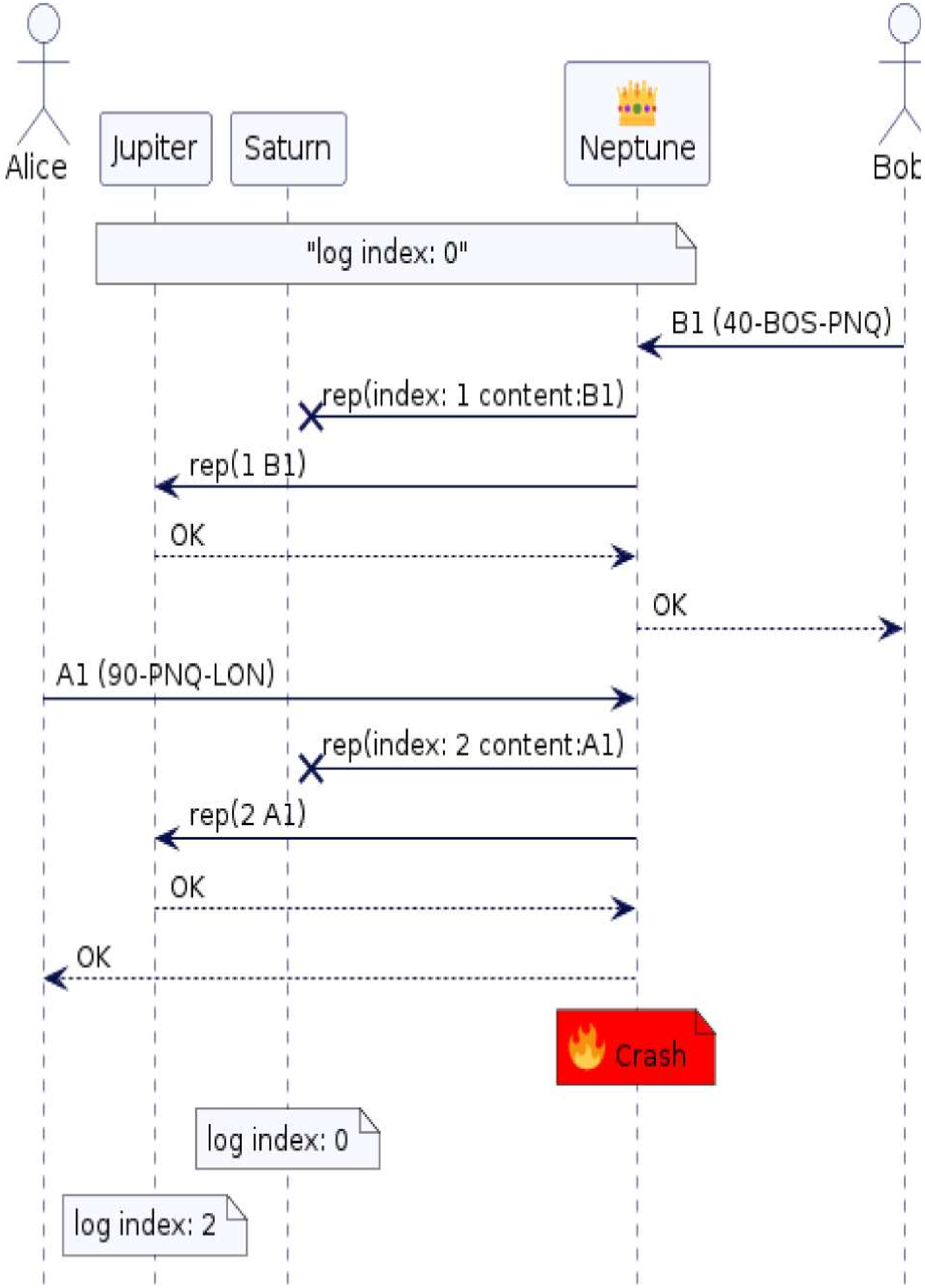


The heartbeat gives us a way to know that Neptune has disconnected, so now we can turn the problem of how to deal with Bob’s request. We need to ensure that once Neptune has confirmed the update to Bob, even if Neptune crashes, the followers can elect a new leader with B1 applied to their data. But we also need to deal with more complication than that, as Neptune may have received multiple messages. Consider the case where both there are messages from both Alice (A1) and Bob (B1) handled by Neptune. Neptune successfully replicates them both with Jupiter but is unable to contact Saturn before it crashes


Alice
Jupiter
Saturn
Neptune
Bob
B1 (40-BOS-PNQ)
rep B1
rep B1
OK
OK
A1 (90-PNQ-LON)
rep A1
rep A1
OK
OK
Crash
Pune: 75
Pune: 25

In this case how do Jupiter and Saturn deal with the fact that they have different states?

The answer is essentially the same as discussed earlier for resilience on a single node. If Neptune writes changes into a *Write-Ahead Log* and treats replication as copying those log entries to its followers, then its followers will be able to figure out what the correct state is by examining the log entries.

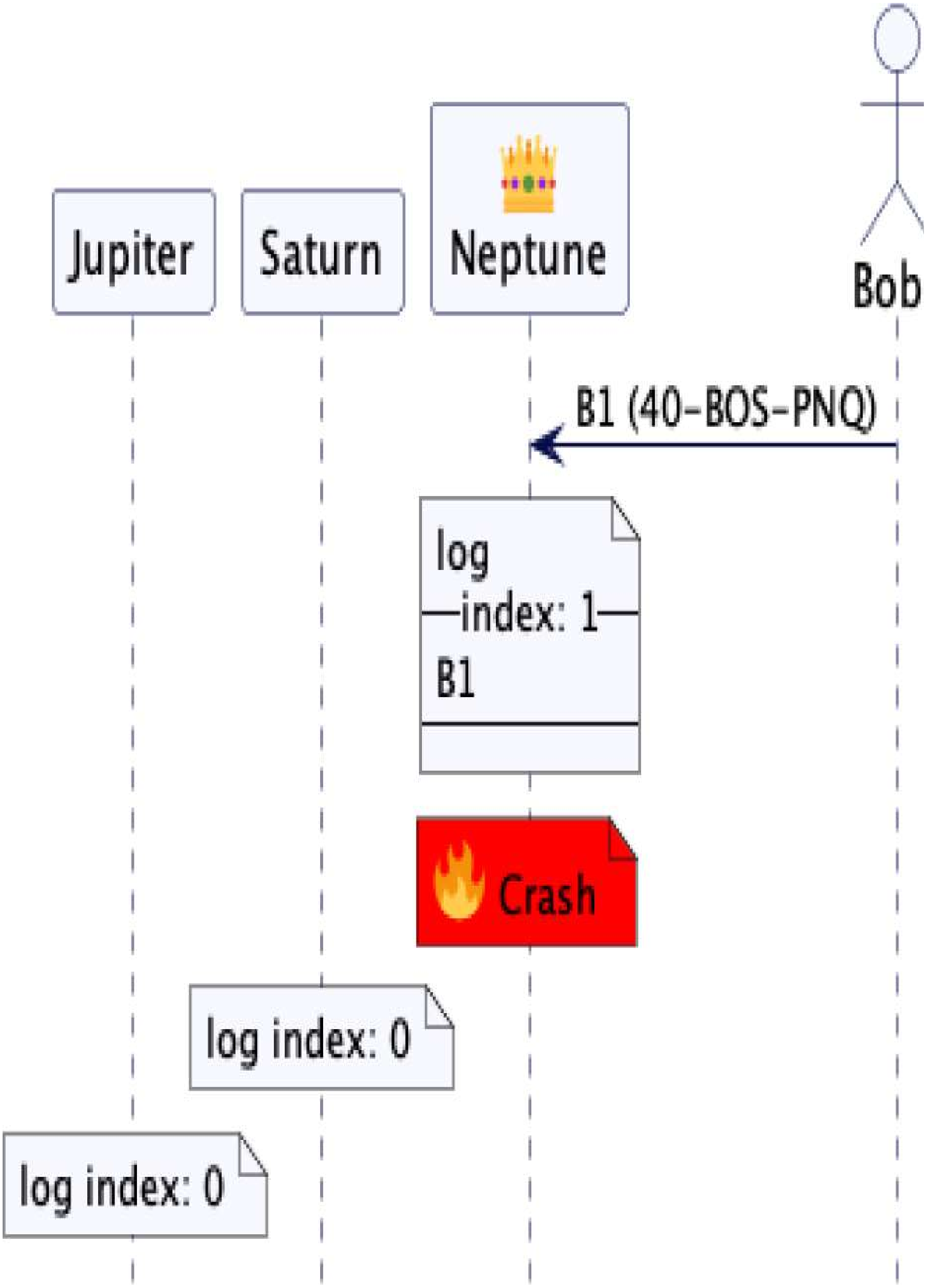

Alice
Jupiter
Saturn
Neptune
Bob
"log index: 0"
B1 (40-BOS-PNQ)
rep(index: 1 content:B1)
rep(1 B1)
OK
OK
A1 (90-PNQ-LON)
rep(index: 2 content:A1)
rep(2 A1)
OK
OK
🔥 Crash
log index: 0
log index: 2

When Jupiter and Saturn elect a new leader, they can easily tell that Jupiter’s log has later index entries, and Saturn can easily apply those log entries to itself to gain a consistent state with Jupiter.

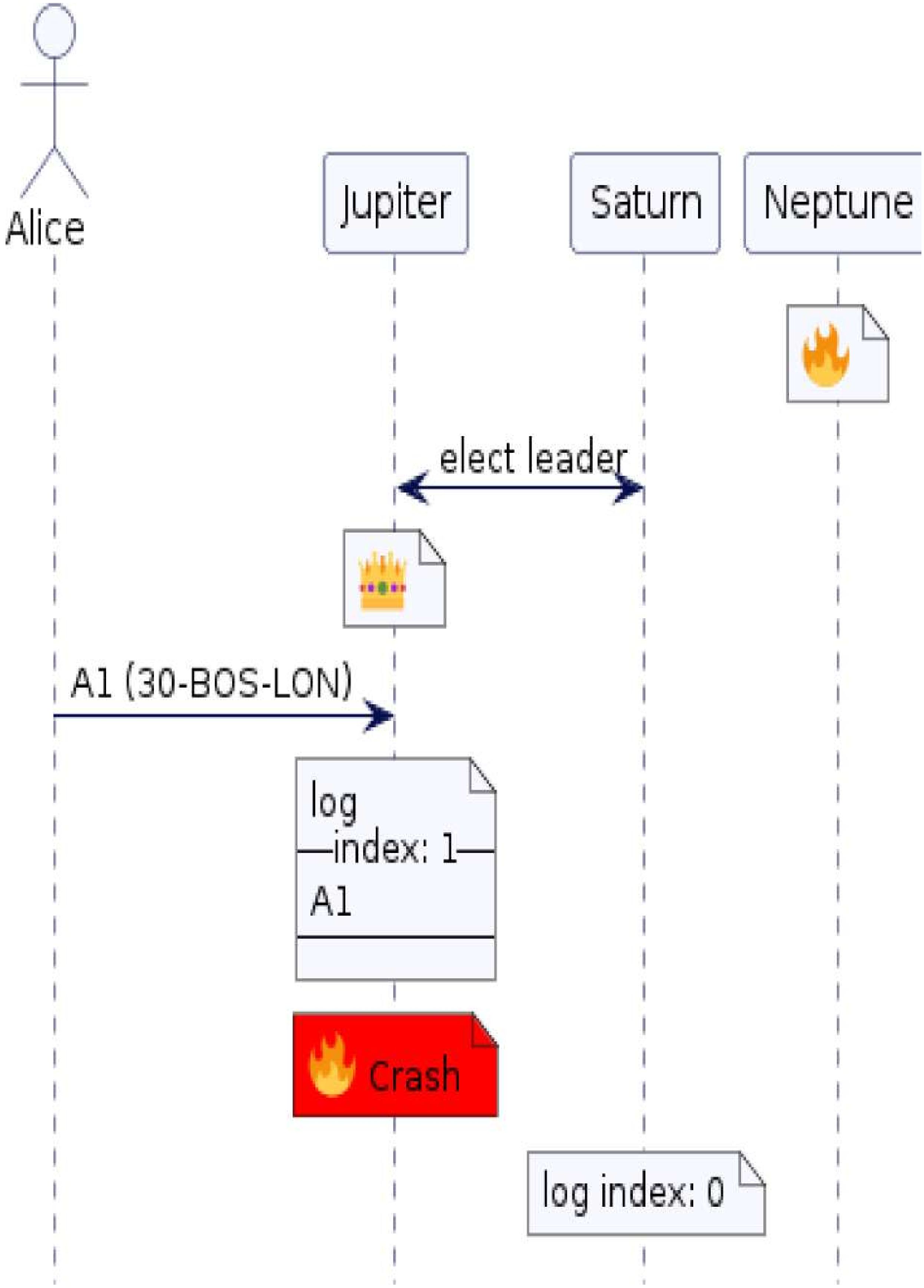
This is also why Neptune can reply to Bob that the update was accepted, even though it hadn’t heard back from Saturn. As long as a *Quorum*, that is a majority, of the nodes in the cluster have successfully replicated the log messages, then Neptune can be sure that the cluster will maintain consistency even if the leader disconnects.

Multiple failures need a *Generation Clock*

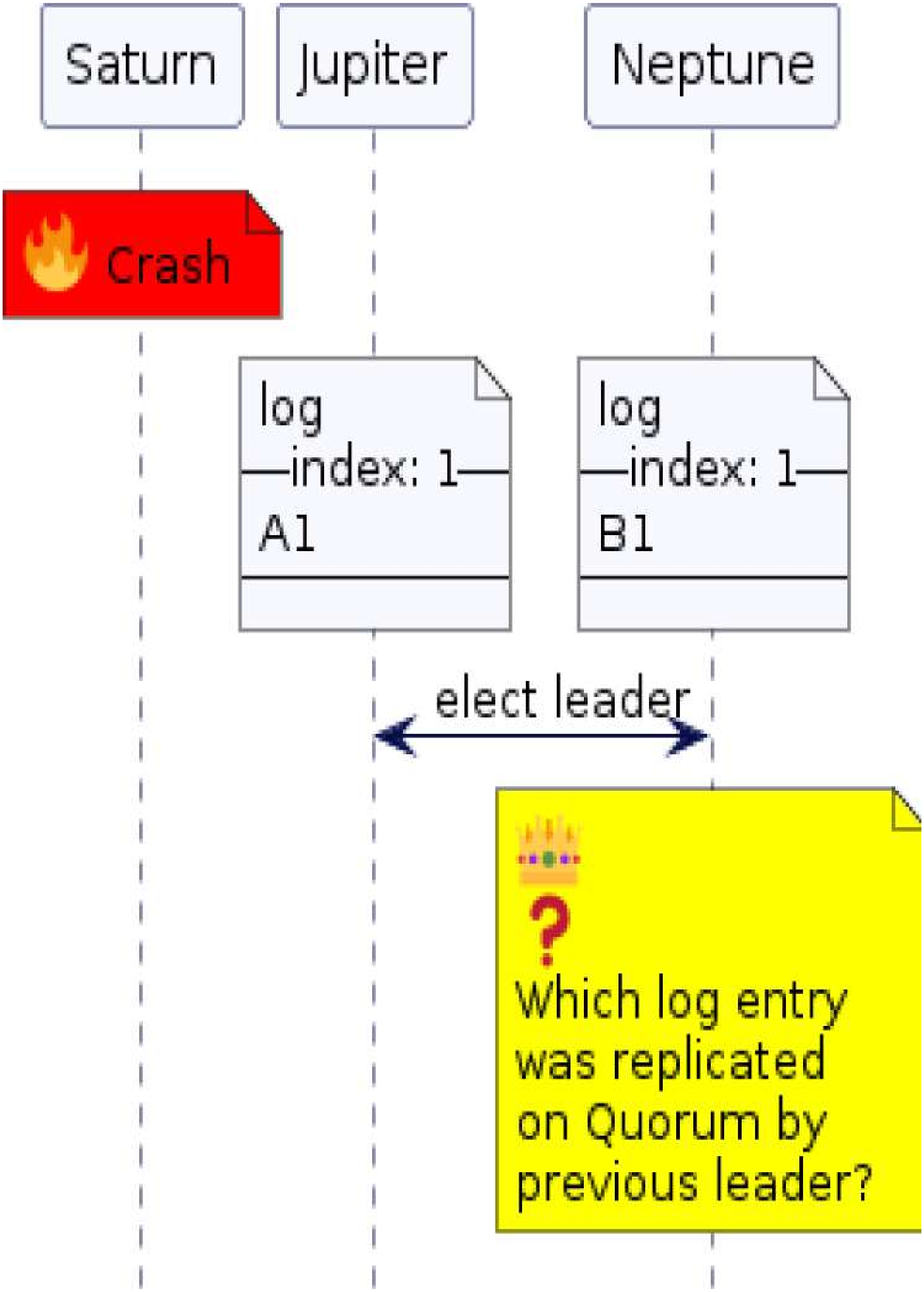
We assumed here that Jupiter and Saturn can figure out whose log is most up to date. But things can get trickier. Lets say Neptune accepted a request from Bob to move 40 widgets from Boston to London but failed before replicating it.


Jupiter
Saturn
Neptune
Bob
B1 (40-BOS-PNQ)
log
index: 1
B1
🔥 Crash
log index: 0
log index: 0

Jupiter is elected as a new leader, and accepts a request from Alice to move 30 widgets from Boston to Pune. But it also crashes before replicating the request to other nodes.

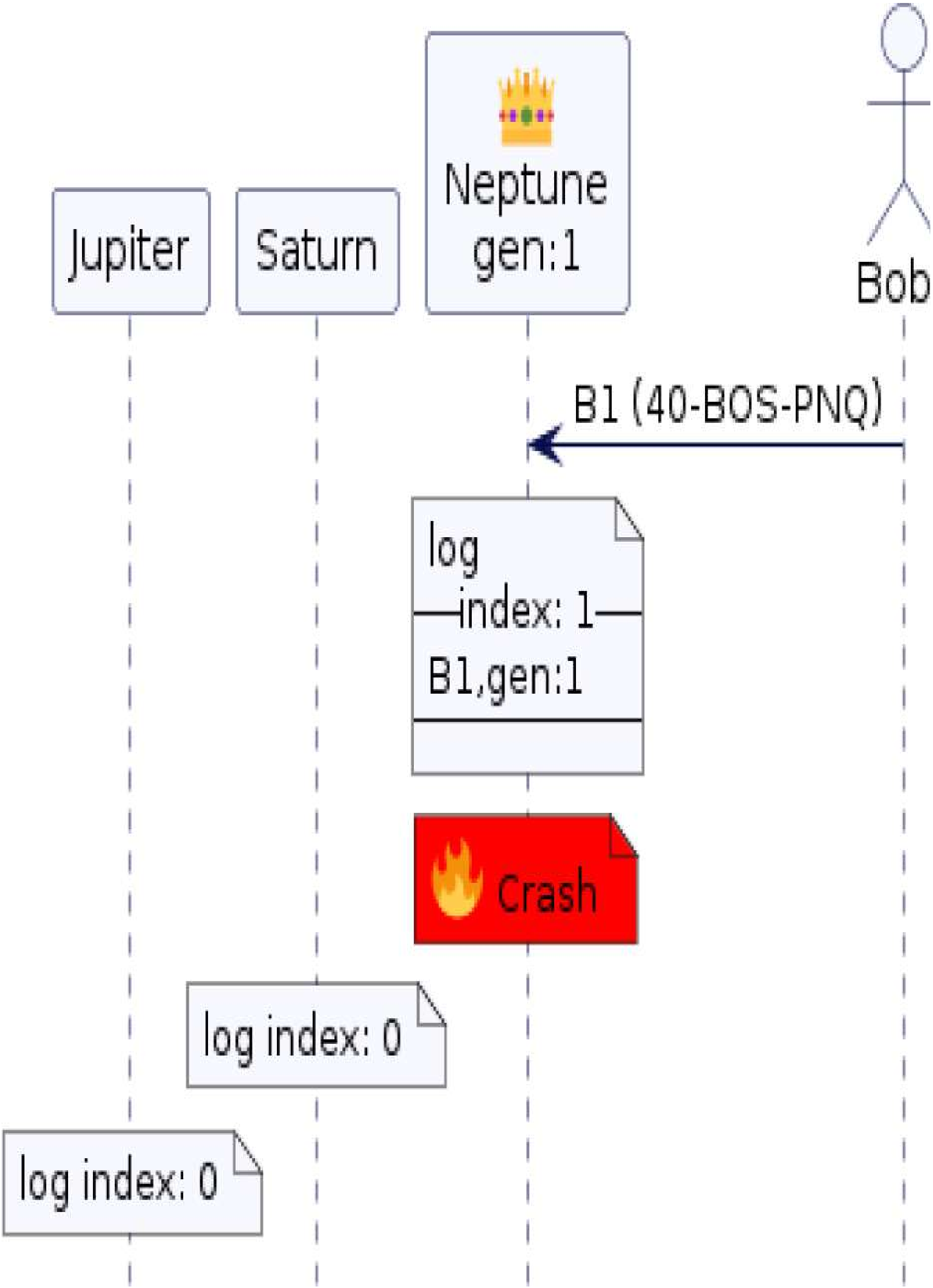

Alice
Jupiter
Saturn
Neptune
elect leader
A1 (30-BOS-LON)
log
index: 1
A1
Crash
log index: 0

In a while, Neptune and Jupiter come back, but before they can talk Saturn crashes. Neptune is elected as a leader. Neptune checks with itself and Jupiter for the log entries. It will see two separate requests at index 1, the one from Bob which it had accepted and the one from Alice that Jupiter has accepted. Neptune can't tell which one it should pick.

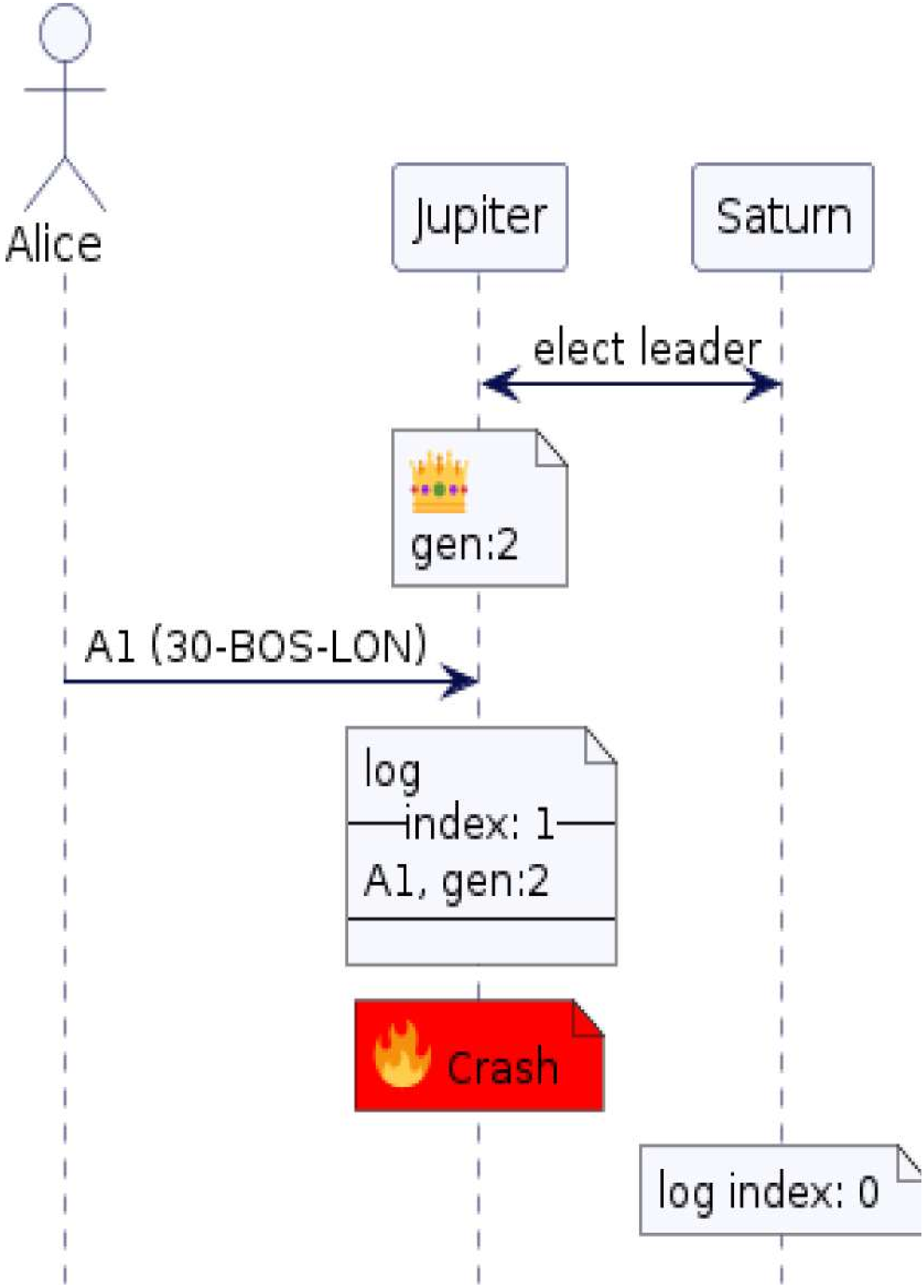

Saturn
Jupiter
Neptune
🔥 Crash
log
index: 1
A1
log
index: 1
B1
elect leader
👑
❓
Which log entry
was replicated
on Quorum by
previous leader?

To solve this kind of situation, we use a Generation Clock. This is a number that increments with each leadership election and a key requirement of *Leader and Followers*.

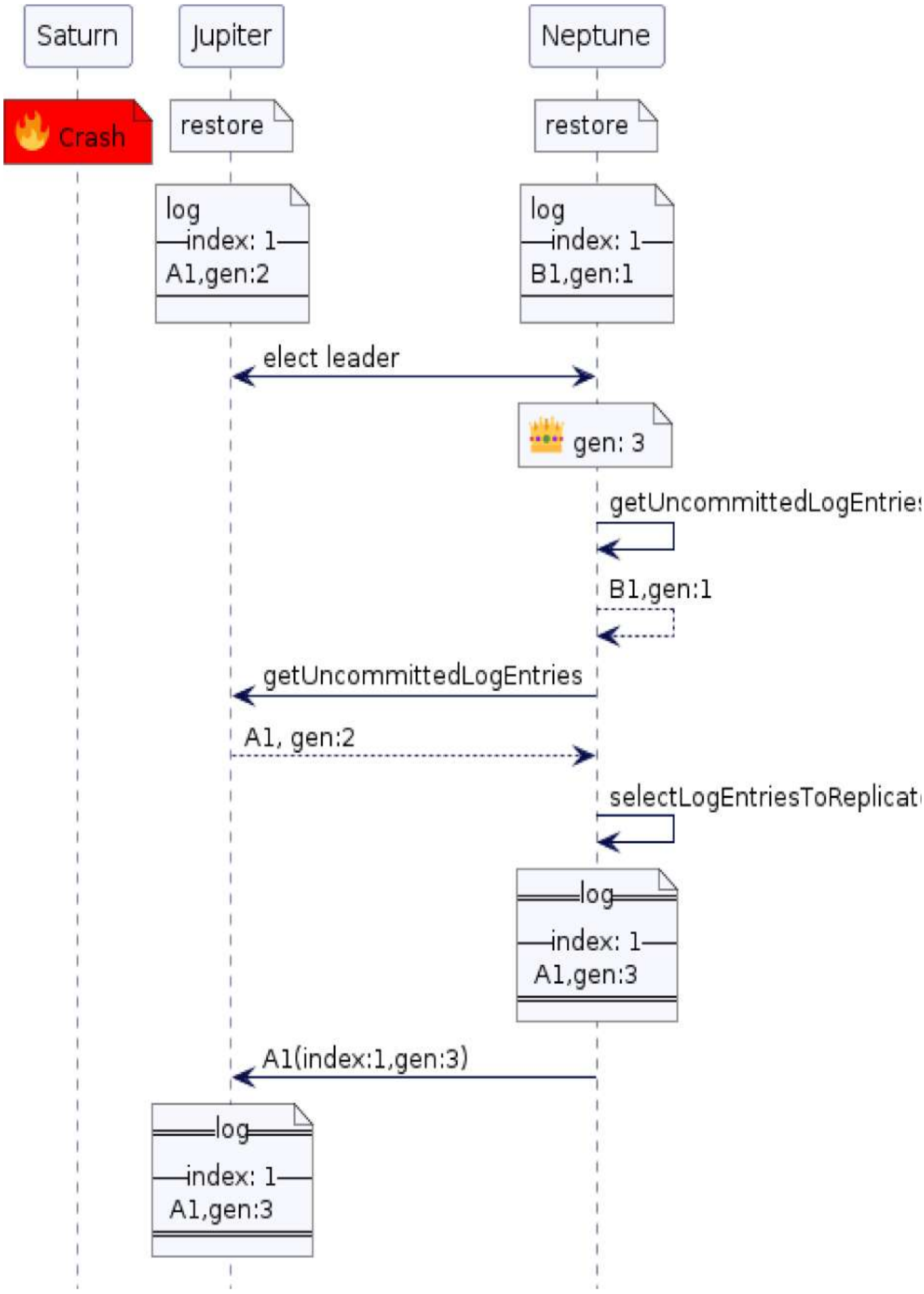
So looking at the previous scenario again, Neptune was leader for generation 1. It adds Bob’s entry in its log marking it with its generation.


Jupiter
Saturn
👑
Neptune
gen:1
Bob
B1 (40-BOS-PNQ)
log
index: 1
B1,gen:1
🔥 Crash
log index: 0
log index: 0

When Jupiter was elected as a leader, it incremented the generation to 2. So when it adds Alice’s entry to its log, its marked for generation 2.

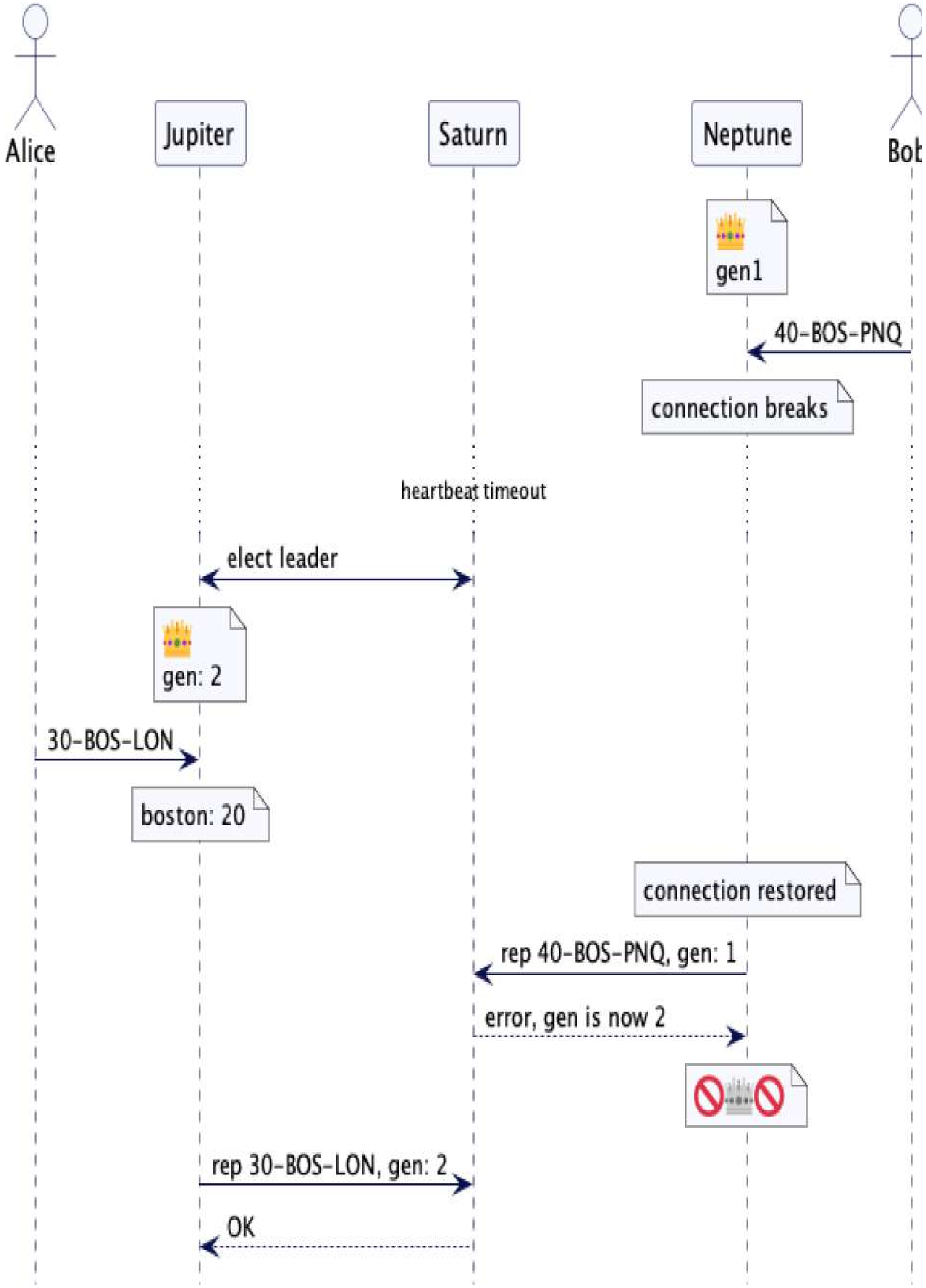

Alice
Jupiter
Saturn
elect leader
gen:2
A1 (30-BOS-LON)
log
index: 1
A1, gen:2
Crash
log index: 0

Now when Neptune is again elected as a leader, it will be for generation 3. Before it starts serving the client requests, it checks the logs of all the available nodes for entries which are not replicated on the *Quorum*. We call these entries as ‘uncommitted’, as they are not yet applied to data. We will see how each node figures out which entries are incompletely replicated in a while. But once the leader knows about these entries, it completes the replication for those entries. In case of conflict it safely pick up the entry with higher generation.


Saturn
Jupiter
Neptune
Crash
restore
restore
log
index: 1
A1,gen:2
log
index: 1
B1,gen:1
elect leader
gen: 3
getUncommittedLogEntrie
B1,gen:1
getUncommittedLogEntries
A1, gen:2
selectLogEntriesToReplicat
log
index: 1
A1,gen:3
A1(index:1,gen:3)
log
index: 1
A1,gen:3

After selecting the entry with the latest generation, Neptune overwrites the uncommitted entry in its own log with its current generation number and replicates with Jupiter.

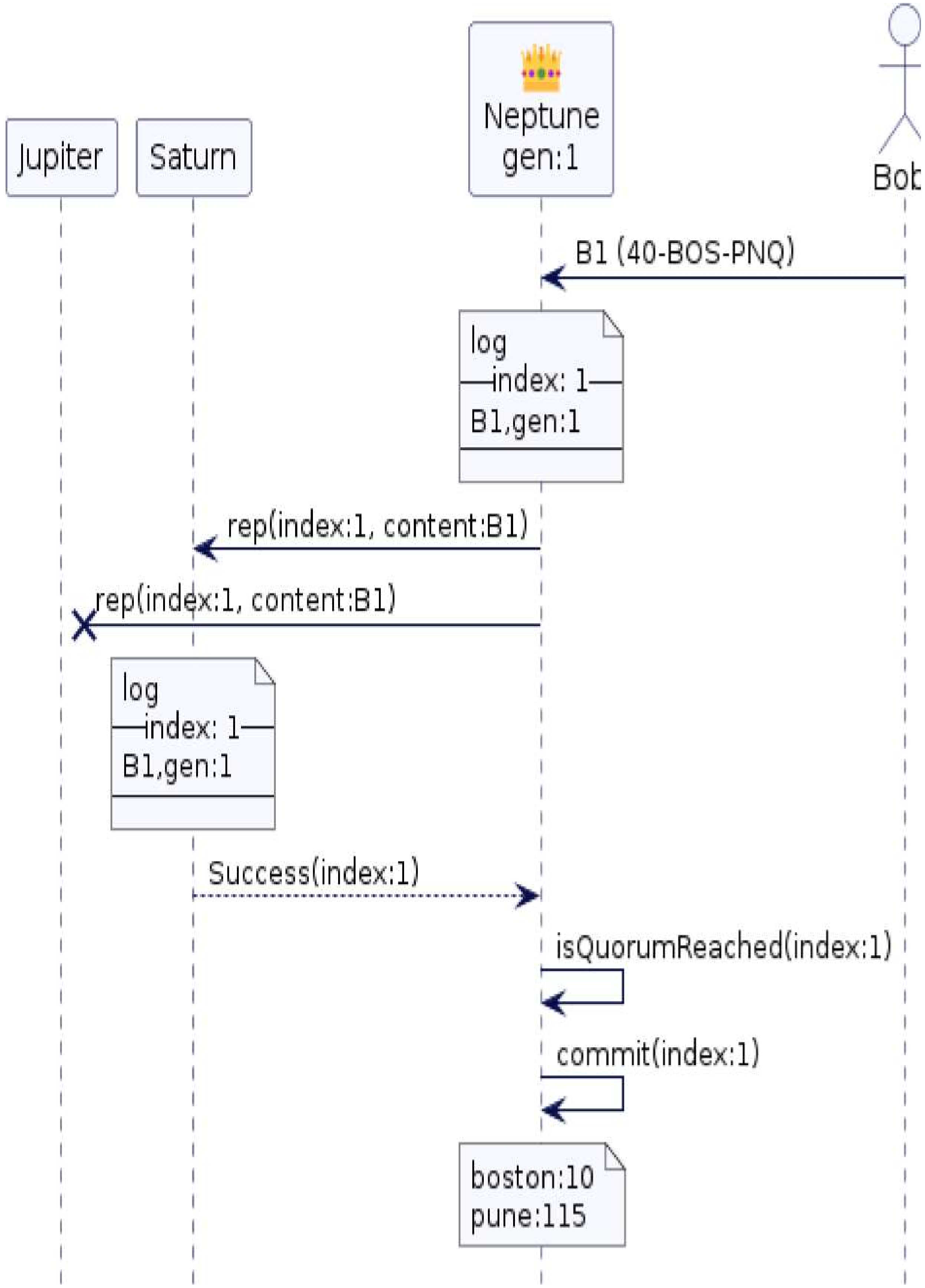
Every node tracks the latest generation it knows of the leader. This is helpful in another problem that might occur. When Jupiter became leader, the previous leader, Neptune might not have crashed, but just temporarily disconnected. It might come back online, and send the requests to Jupiter and Saturn. If Jupiter and Saturn have elected a new leader and accepted requests from Alice, what should they do when they suddenly start getting requests from Neptune? Generation Clock is useful in this case as well. Every request is sent to cluster nodes, along with the generation clock. So every node can always choose the requests with the higher generation and reject the ones with the lower generation.


Alice
Jupiter
Saturn
Neptune
Bob
gen1
40-BOS-PNQ
connection breaks
heartbeat timeout
elect leader
gen: 2
30-BOS-LON
boston: 20
connection restored
rep 40-BOS-PNQ, gen: 1
error, gen is now 2
rep 30-BOS-LON, gen: 2
OK

Log entries cannot be committed until they are accepted by a *Quorum*

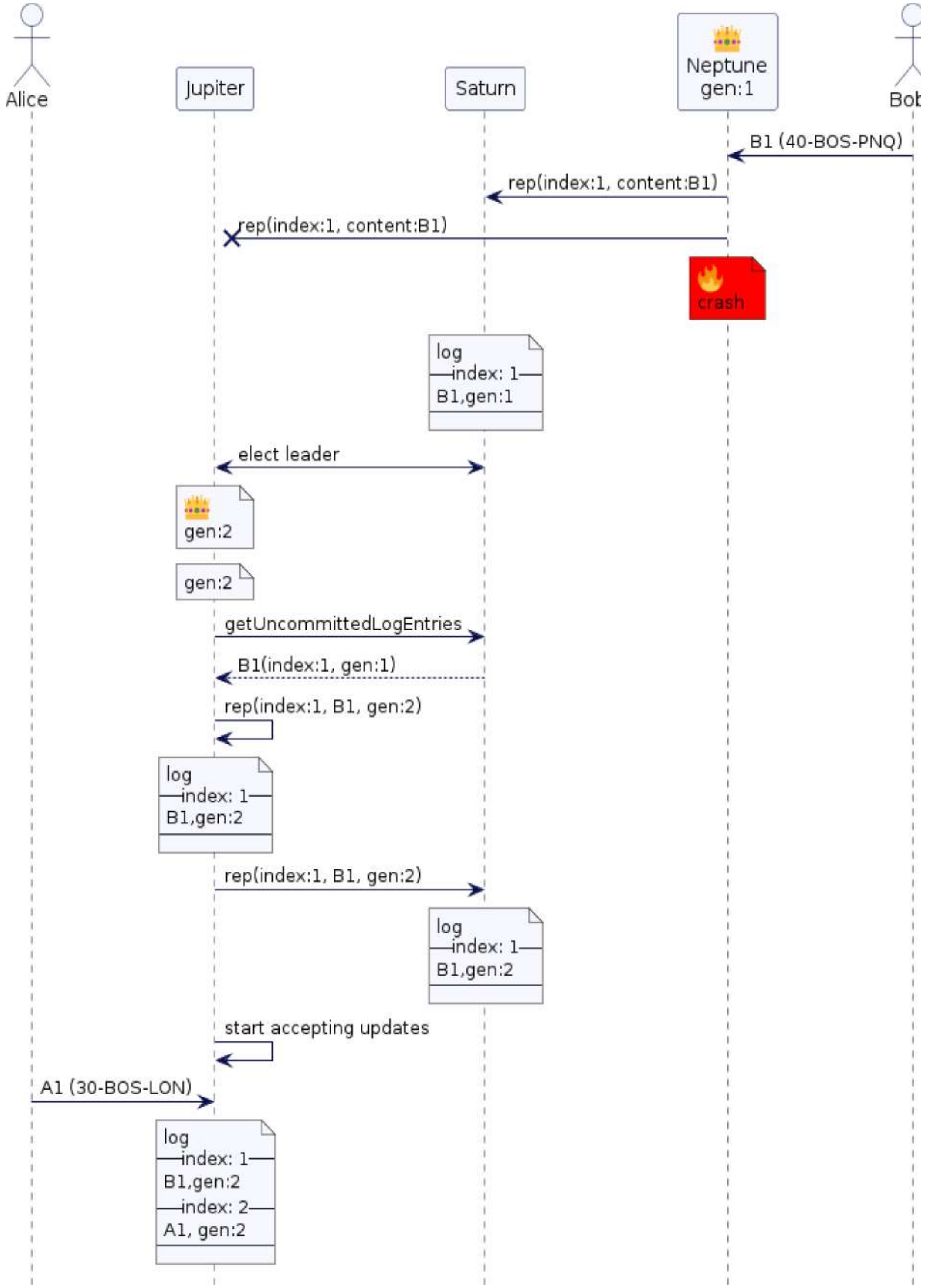
As seen above, entries like B1, can be overwritten if they haven't been successfully replicated to a Quorum of nodes in the cluster. So the leader can not apply the request to its data store after just appending to it's own log, it has to wait until it gets enough acknowledgments from other nodes first. When an update is added to a local log it is *uncommitted*, until the leader has had replies from a Quorum of other nodes, at which point it becomes *committed*. In the case of the example above, Neptune cannot commit B1 until it hears that at least one other node has accepted it, which point that other node, plus Neptune itself, makes two out of three nodes - a majority and thus a Quorum.

When Neptune, the leader, receives an update, either from a user (Bob) directly or via a follower, it adds the uncommitted update to its log and then sends replication messages to the other nodes. Once Saturn (for example) replies, that means two nodes have accepted the update (Neptune and Saturn), this is 2 out of 3 nodes, which is the majority and thus a Quorum. At that point Neptune can commit the update.


Jupiter
Saturn
Neptune
gen:1
Bob
B1 (40-BOS-PNQ)
log
index: 1
B1,gen:1
rep(index:1, content:B1)
rep(index:1, content:B1)
log
index: 1
B1,gen:1
Success(index:1)
isQuorumReached(index:1)
commit(index:1)
boston:10
pune:115

The importance of the Quorum is that it applies to decision by the cluster. Should a node fail, any leadership election must involve a Quorum of nodes. Since any committed updates have also been sent to a Quorum of nodes, we can be sure that committed updates will be visible during the election.

If Neptune receives Bob’s update (B1), replicates, gets an acknowledgment from Saturn, and then crashes, Saturn still has a copy of B1. If the nodes then elect Jupiter as the leader, Jupiter must apply any uncommitted updates, that is B1, before it can start accepting new ones.


Alice
Jupiter
Saturn
Neptune
gen:1
Bob
B1 (40-BOS-PNQ)
rep(index:1, content:B1)
rep(index:1, content:B1)
crash
log
index: 1
B1,gen:1
elect leader
gen:2
gen:2
getUncommittedLogEntries
B1(index:1, gen:1)
rep(index:1, B1, gen:2)
log
index: 1
B1,gen:2
rep(index:1, B1, gen:2)
log
index: 1
B1,gen:2
start accepting updates
A1 (30-BOS-LON)
log
index: 1
B1,gen:2
index: 2
A1, gen:2

When the log is large, moving the log across nodes for leader election can be costly. So leader-election is often optimized to elect the leader which has most up-to-date log.

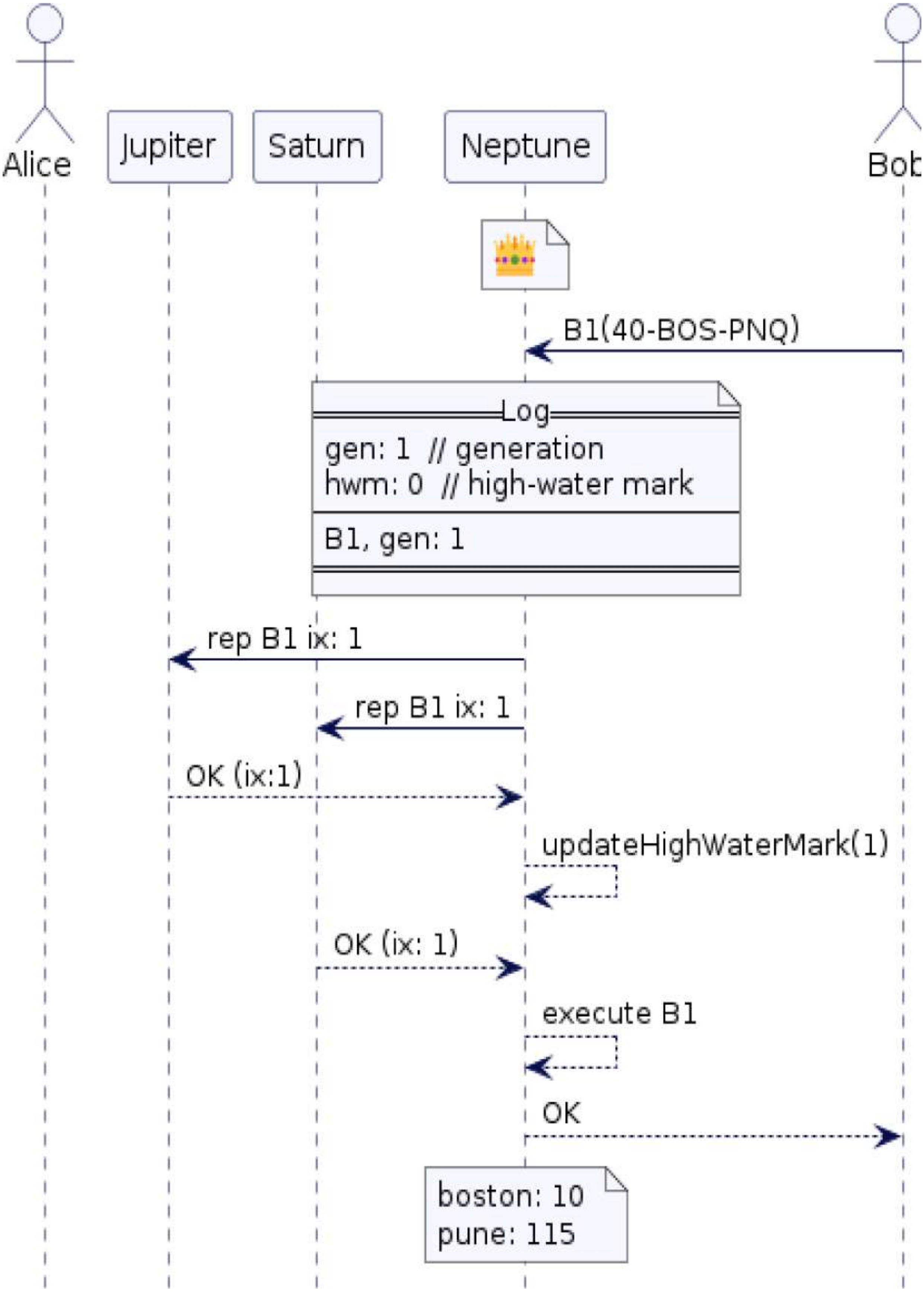
The most commonly used algorithm for *Replicated Log*, Raft [bib-raft], optimizes this by electing the leader with the most up-to-date log. In the above example this would elect Saturn as the leader.

Followers commit based on a *High-Water Mark*

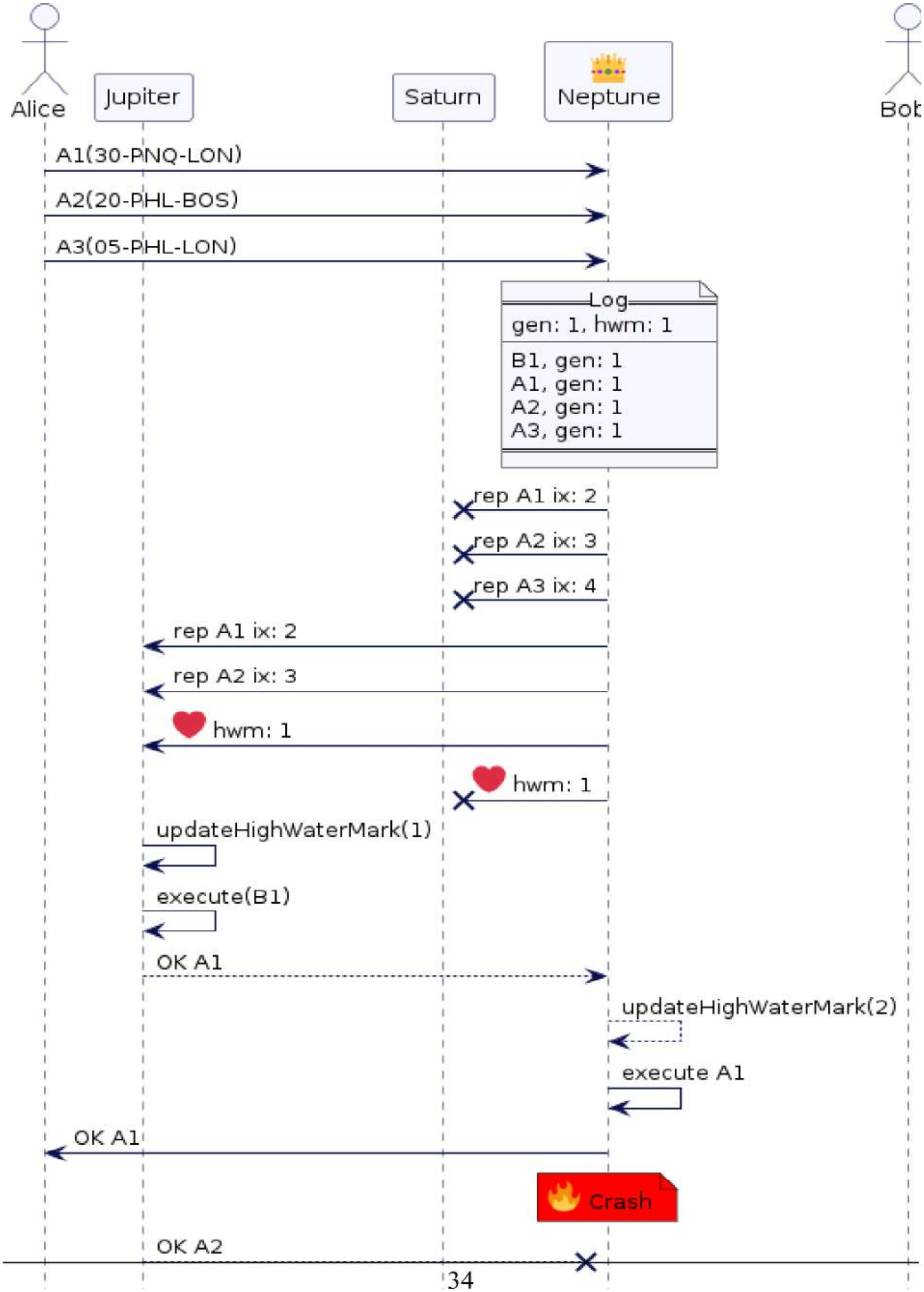
As we’ve seen, leaders commit when they get acknowledgments from a *Quorum*, but when do followers commit their log entries? In the three node example we’ve been using, it’s obvious. Since we know the leader must have added the log entry before it replicates, any node knows that it can commit since it and the leader form a Quorum. But that isn’t true for larger clusters, in a five node cluster a single follower and a leader is only 2 of 5.

A High-Water Mark solves this conundrum. Simply put, the High-Water Mark is maintained by the leader and is equal to the index of the latest update to be committed. The leader then adds the High-Water Mark to its *HeartBeat*. Whenever a follower receives a HeartBeat, it knows can commit all its log entries up to the High-Water Mark.

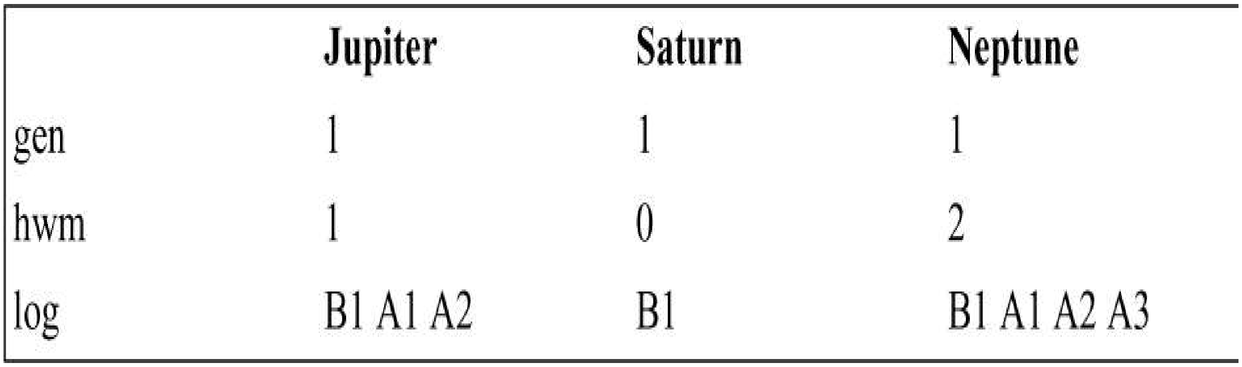
Let’s look at an example of this. Bob sends a request (B1) to Neptune. Neptune replicates the request to Jupiter and Saturn. Jupiter acknowledges first, allowing Neptune to increase its High-Water Mark to 1, execute the update against its data store and return success to Bob. Saturn’s acknowledgment is late, and since it’s not higher than the High-Water Mark, Neptune takes no action on it.


Alice
Jupiter
Saturn
Neptune
Bob
👑
B1(40-BOS-PNQ)
Log
gen: 1 // generation
hwm: 0 // high-water mark
B1, gen: 1
rep B1 ix: 1
rep B1 ix: 1
OK (ix:1)
updateHighWaterMark(1)
OK (ix: 1)
execute B1
OK
boston: 10
pune: 115

Neptune now gets three requests from Alice (A1, A2, and A3). Neptune puts all of these into its log and starts sending replication messages. The link between Neptune and Saturn, however, gets tangled and Saturn doesn’t get them. After the first two, Neptune concidentally sends out heartbeats, which alerts followers to update their High-Water Mark. Jupiter acknowledges A1, allowing Neptune to update its High-Water Mark to 2, execute the update and notify Alice. But then Neptune crashes before it’s able to replicate A3.

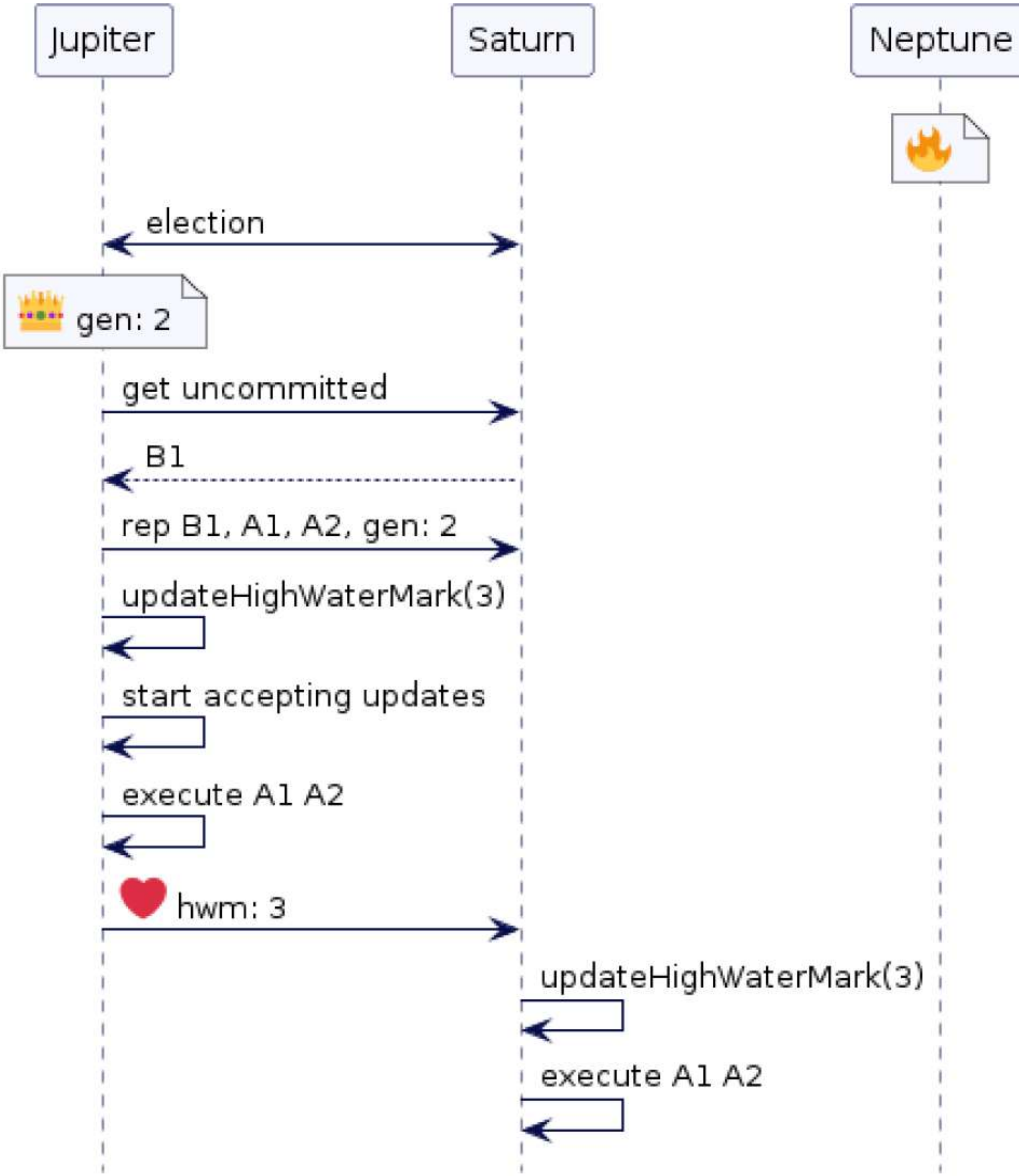

Alice
Jupiter
Saturn
Neptune
Bob
A1(30-PNQ-LON)
A2(20-PHL-BOS)
A3(05-PHL-LON)
Log
gen: 1, hwm: 1
B1, gen: 1
A1, gen: 1
A2, gen: 1
A3, gen: 1
rep A1 ix: 2
rep A2 ix: 3
rep A3 ix: 4
rep A1 ix: 2
rep A2 ix: 3
hwm: 1
hwm: 1
updateHighWaterMark(1)
execute(B1)
OK A1
updateHighWaterMark(2)
execute A1
OK A1
Crash
OK A2

At this point, here are the states of the nodes.



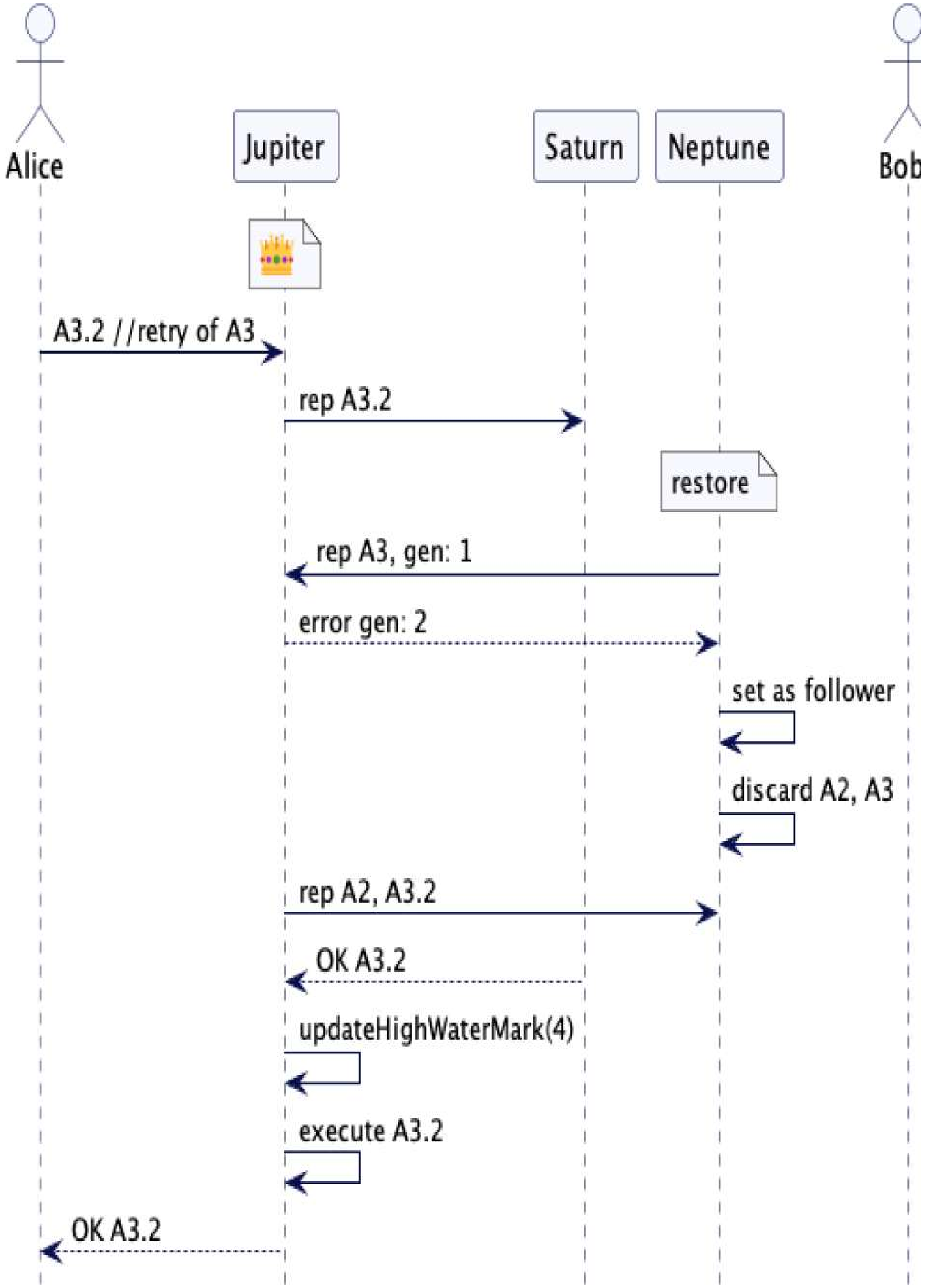
	Jupiter	Saturn	Neptune
gen	1	1	1
hwm	1	0	2
log	B1 A1 A2	B1	B1 A1 A2 A3

Jupiter and Saturn fail to get HeartBeat from Neptune and thus hold an election for a new leader. Jupiter wins and gathers log entries. In doing this it accepts that A2 reached Quorum and sets its High-Water Mark to 3. Jupiter replicates its log to Saturn, and when Saturn gets a HeartBeat with High-Water Mark of 3 it’s able to update its High-Water Mark and execute the updates against its store.



	Jupiter	**Saturn**	**Neptune**
gen	2	2	1
hwm	3	3	2
log	B1 A1 A2	B1 A1 A2	B1 A1 A2 A3

At this point Alice times out of her A3 request and resends it (A3.2), which routes to Jupiter as the new leader. Just as this happens, Neptune starts back up again. Neptune tries to replicate A3, and is told that there’s a new generation of leader, so Neptune now accepts that it’s a follower of Jupiter, discarding its log down to its High-Water Mark. Jupiter sends replication messages for A2 and A3.2. Once Jupiter gets an acknowledgment for A3.2, it can update its High-Water Mark, execute the update, and respond to Alice.


Alice
Jupiter
Saturn
Neptune
Bob
👑
A3.2 //retry of A3
rep A3.2
restore
rep A3, gen: 1
error gen: 2
set as follower
discard A2, A3
rep A2, A3.2
OK A3.2
updateHighWaterMark(4)
execute A3.2
OK A3.2

Saturn and Neptune will update their states on the next HeartBeat from Jupiter.

Leaders use a series of queues to remain responsive to many clients

A leader has to handle a lot of requests from many clients. Each request takes a fair bit of processing. This processing happens in multiple stages. Requests need to be parsed to understand the request and its payload. Updates need to persisted to a *Write-Ahead Log*, which means a write to a durable store, and in this context "durable" means "slow". Requests may also be acknowledgments from followers for a replication request, for these the leader needs to find the request, check to see if it’s reached *Quorum* and if so, update the *High-Water Mark*.

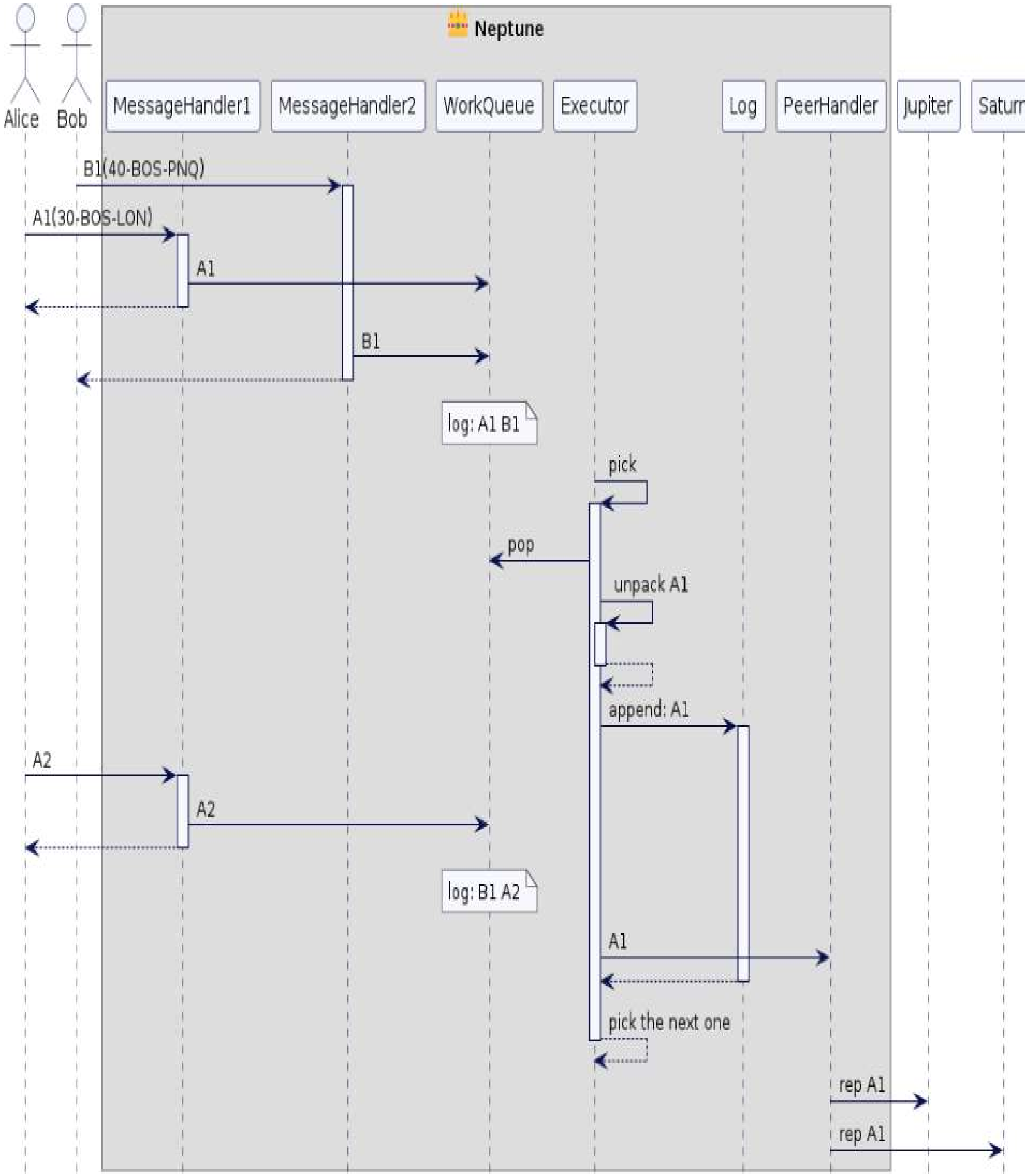
We need to ensure that that these operations don’t run into problems with multiple threads trying to update the same data at once. Each entry on the Write-Ahead Log needs to be written and processed in full before we start to write another, but we don’t want clients to wait for other clients to finish their work. But at the same time we do not want other processing stages to be blocked while all of this is going on.

For these reasons, we use a *Singular Update Queue*. Most programming languages these days will have some form of in-memory queue object, which handles requests from multiple threads. Singular Update Queue build on this by allowing client threads to write simple entries onto such an in-memory queue. A separate processing thread takes entries from this work queue, and carries out the processing we discussed above. This way the system remains responsive to clients, but also keeps the processing of requests in a saner, single-threaded world.

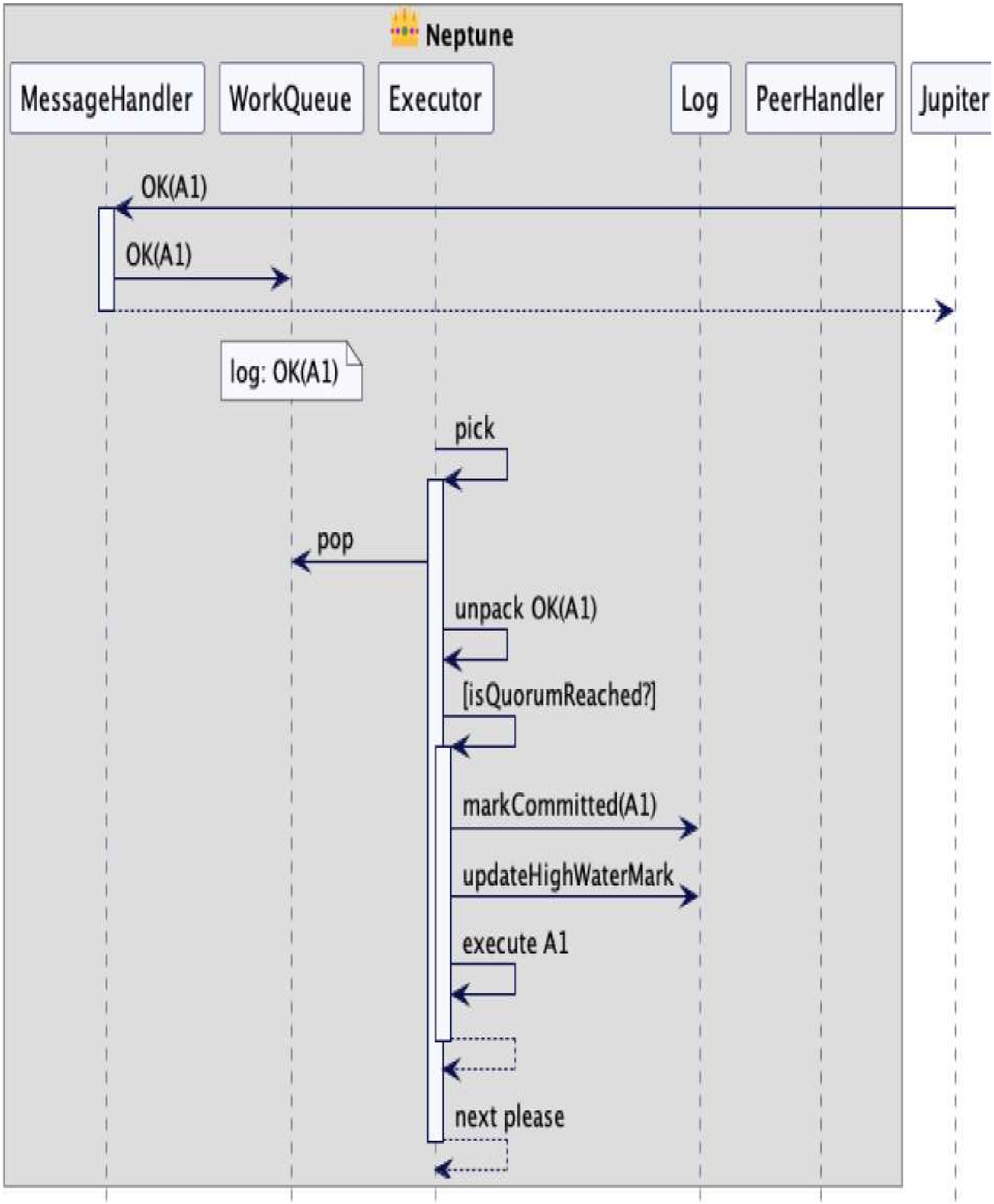
Some programming languages like Go programming langauage [bib-go-lang], have first class support for this mechanism with channels and go-routines.

If Alice and Bob both send messages (A1 and B1) to Neptune, they will be handled by different message handling threads on Neptune. Each of these put the mostly-raw message onto the work queue. The thread handling the

Replicated Log, works independently, popping from the head of the queue, unpacking the details, adding to the log, and sending for replication.

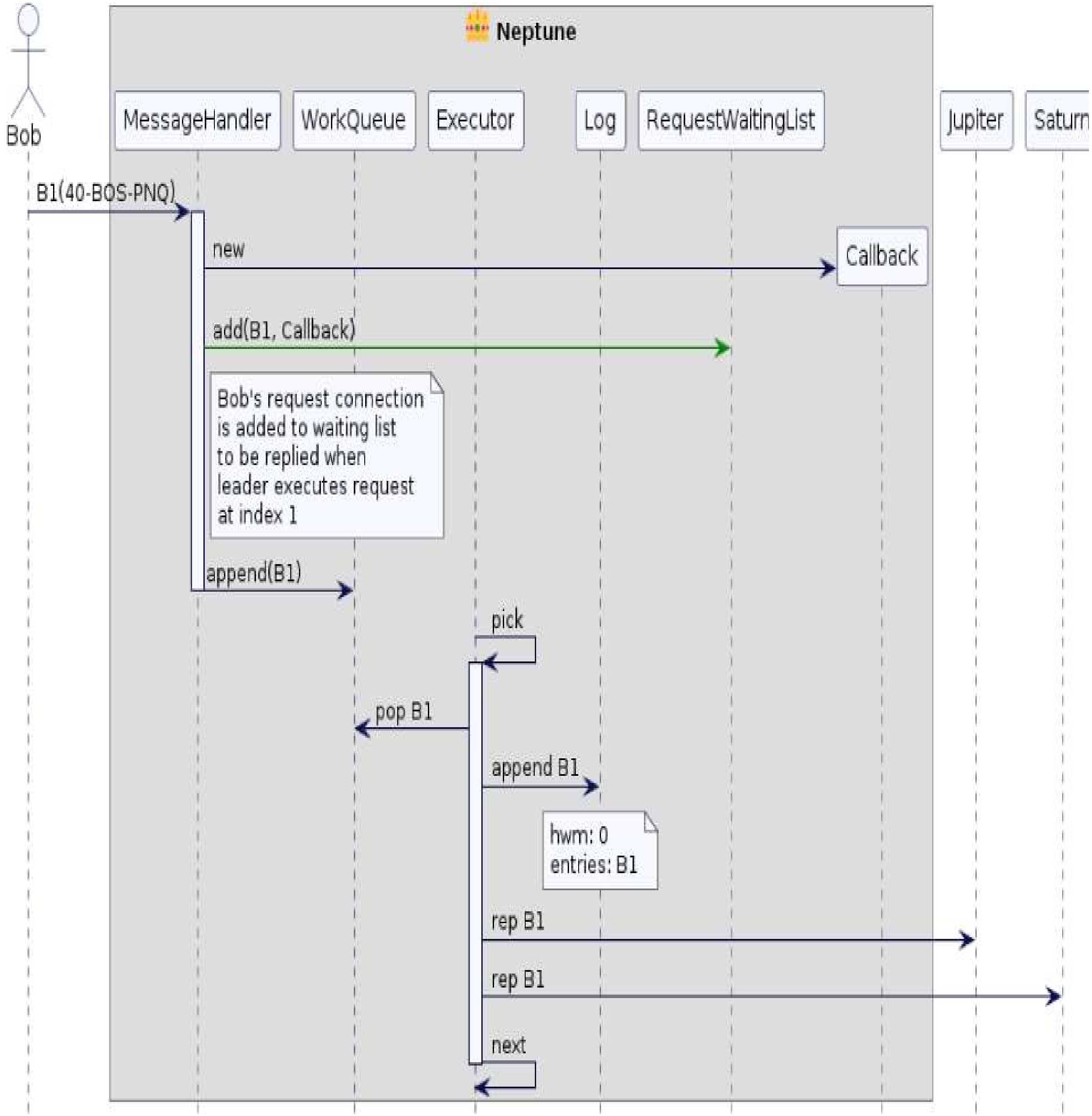


When Jupiter acknowledges the replication, its response is handled by a message handler, which just puts the raw message on the work queue. The processing thread picks this message, check Quorum, marks the log entry as committed, and updates the High-Water Mark.

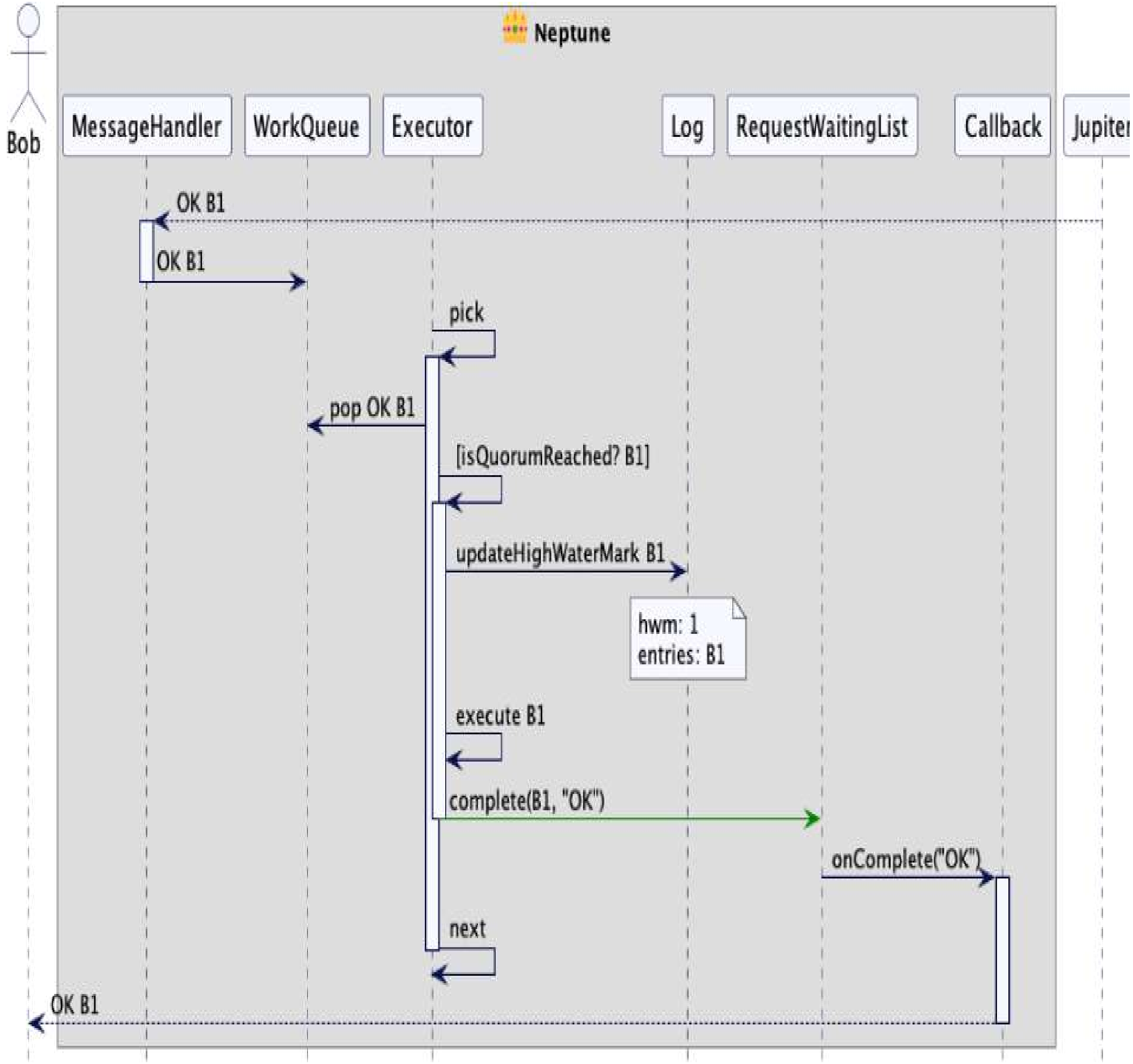


Given this approach, clients receive an immediate acknowlegement that their request has been received, but need additional mechanisms to ensure they know when a request has been committed and acted on by the cluster. If nothing else, they need to know if they need to retry a request if it doesn't go through. In a cluster like this, that update can only be confirmed once a Quorum of nodes accepts the update. But we don't want to leave either the client or the message handler blocked while all this goes on. The client has other things to do and the message handler can possibly deal with more requests while the cluster replicates and reaches Quorum. So instead of blocking, we use a *Request Waiting List* to track waiting requests and respond to clients when the requests are actually executed before the request is put on the work queue.

When the leader receives the request, it adds a callback to Request Waiting List that will contain the behavior of how to notify Bob when the request succeeds (or fails).



When Jupiter acknowledges the update, the executor notifies the Request Waiting List, which invokes the callback, notifying Bob of the success of the request.

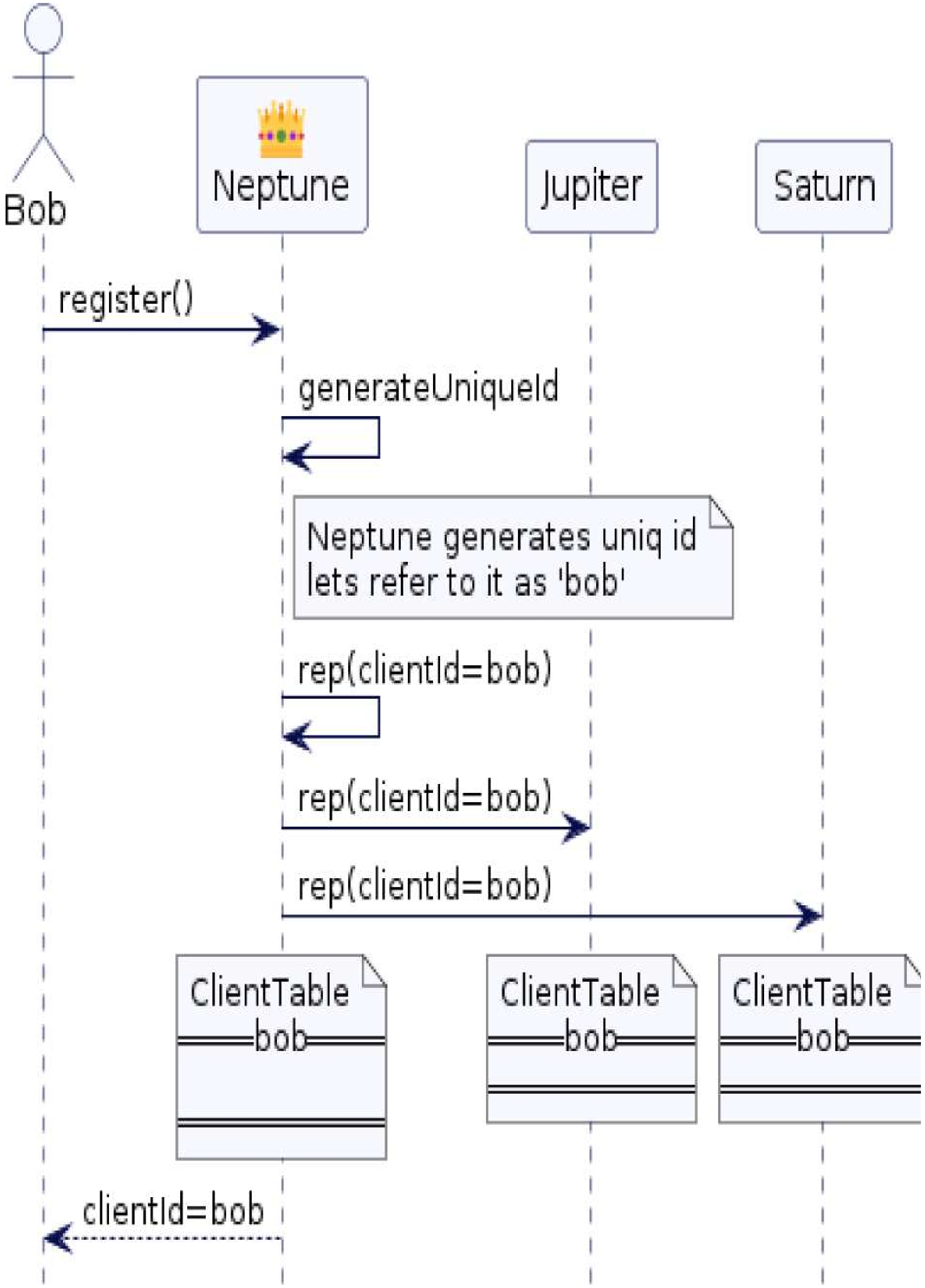


But what if the leader fails before it sends an acknowledgment back to the client? The client then doesn't know whether the cluster still managed to commit the request or if the leader's failure lost the request. In those circumstances, the client needs to retry its request, but this leads to a second problem, as we don't want to make the same transfer twice.

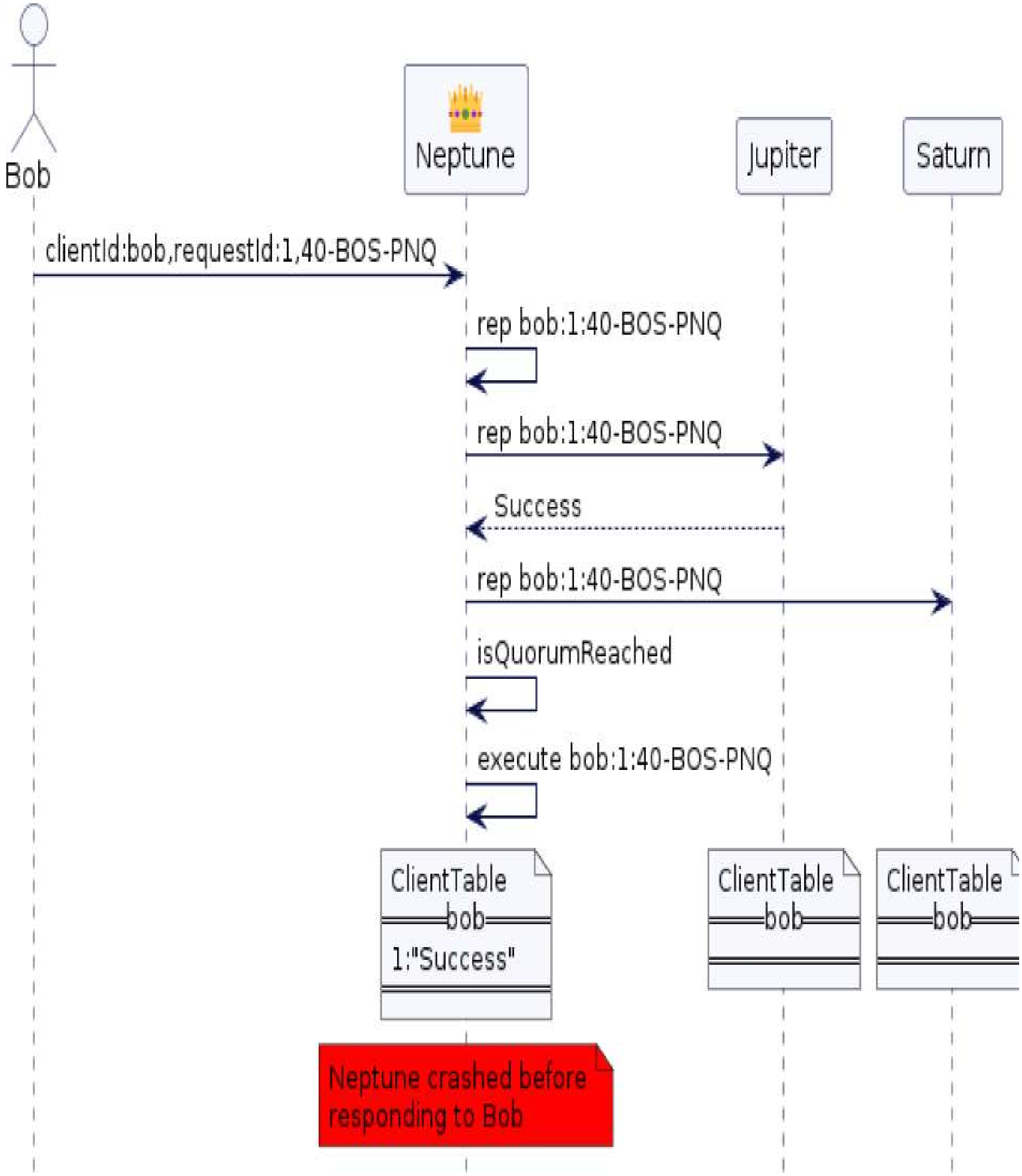
To avoid executing a retried requests again, cluster nodes ensure they are an *Idempotent Receiver*. (An idempotent operation is one that can be executed multiple times with the same effect as if it were implemented once. Adding 1 to a variable isn't idempotent, but setting a variable to a value is.)

To implement idempotency, each client registers itself with the leader before sending any requests. The client registration is also replicated across all the replicas similar to any other request. The registered client assigns a unique number to each request. The server can then use client id and unique request number to store the responses of the executed requests. This mapping is used, when a client repeats a request. Instead of executing the request again, the server returns the stored response. That way, even an non-idempotent request, such as transfer 40 widgets, can be handled in an idempotent way.

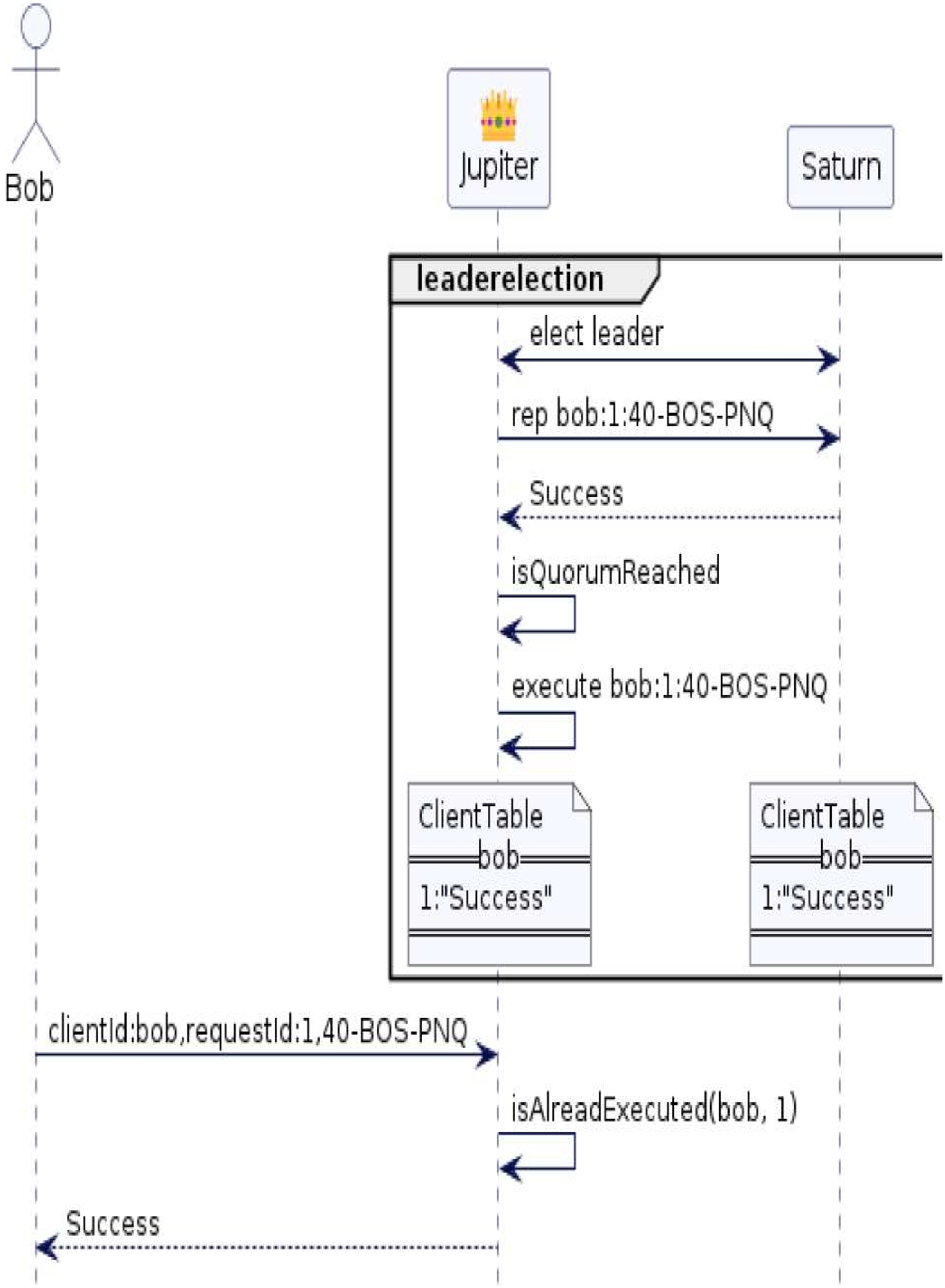
So from our example, Bob registers itself before starting to send any requests. The registration request is replicated in the Replicated Log and a unique client id is returned to Bob. Each cluster node maintains a table with entries for client ids.


Bob
Neptune
Jupiter
Saturn
register()
generateUniqueId
Neptune generates uniq id
lets refer to it as 'bob'
rep(clientId=bob)
rep(clientId=bob)
rep(clientId=bob)
ClientTable
bob
ClientTable
bob
ClientTable
bob
clientId=bob

Bob now uses client id, "bob" to send the requests. It also assigns a unique number to each request. Here it sends request 40-BOS-PNQ with request number 1. Whenever the request is executed, the response is stored in the client table.

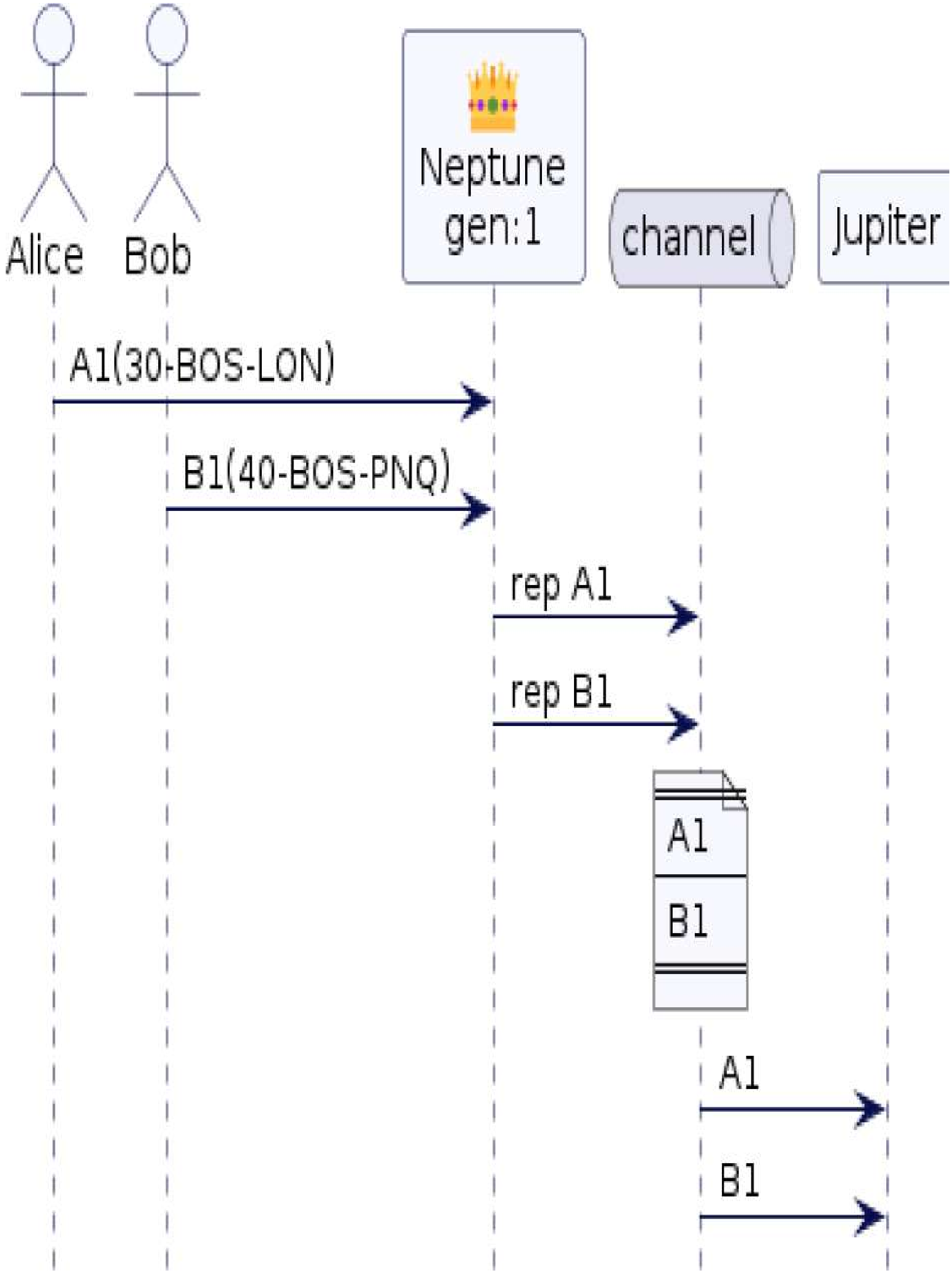


Neptune fails, before sending the response to Bob. Jupiter and Saturn now run a leader election as discussed in the previous section. Jupiter will execute pending log entries. Once it executes the request 40-BOS-PNQ from bob, it makes an entry in the client table. At this point, Bob has not received the response, so he retries 40-BOS-PNQ again, with Jupiter. But because Jupiter has already executed the request numbered 1 from bob, it will return the response already stored. This way, the retried request from bob is not executed again.


Bob
Jupiter
Saturn
leaderelection
elect leader
rep bob:1:40-BOS-PNQ
Success
isQuorumReached
execute bob:1:40-BOS-PNQ
ClientTable
bob
1:"Success"
ClientTable
bob
1:"Success"
clientId:bob,requestId:1,40-BOS-PNQ
isAlreadExecuted(bob, 1)
Success

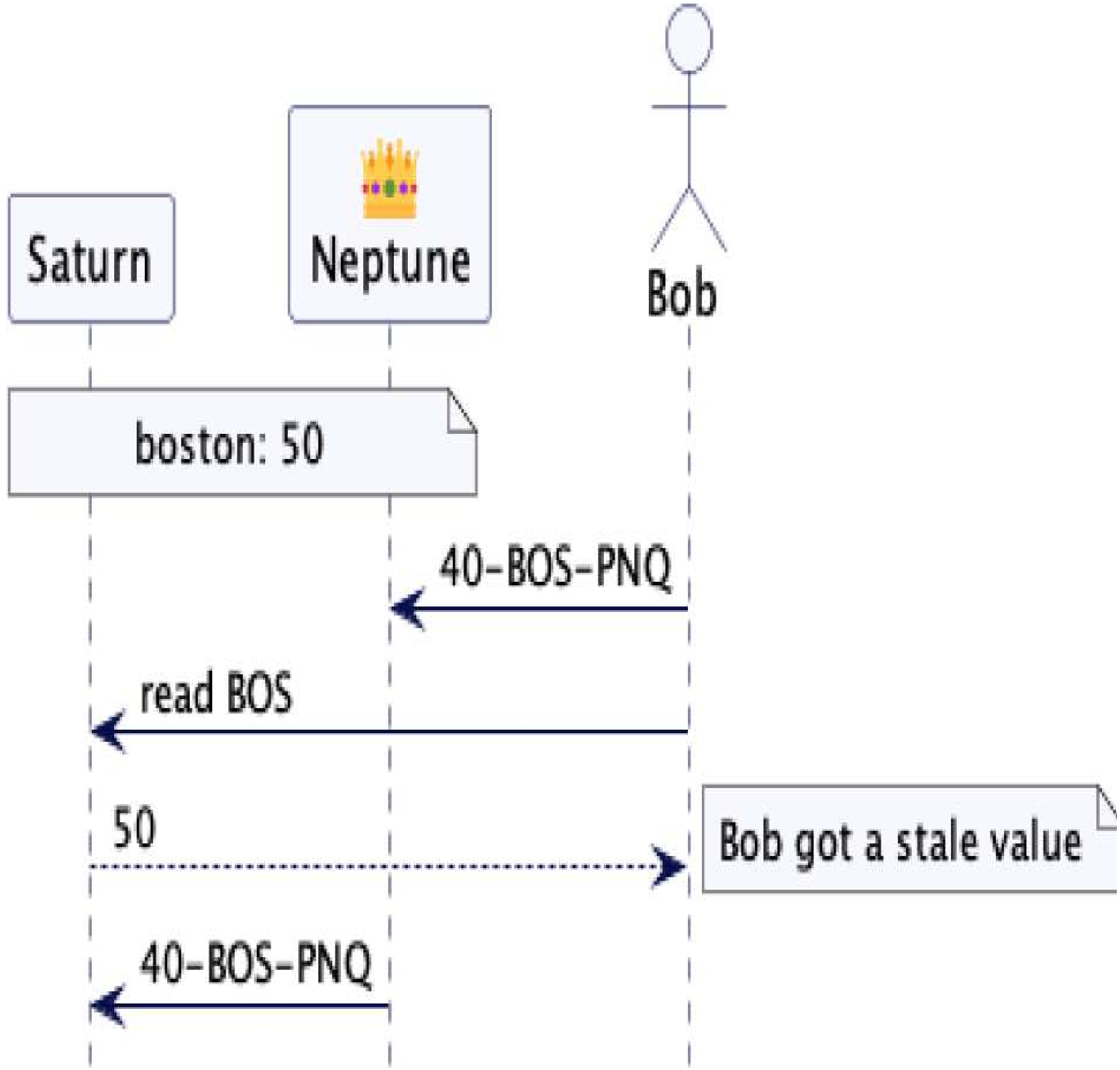
Given we are replicating an ordered log, nodes have to be careful that they maintain the order of the entries when there is no guarantee that they will receive messages in the right order. After all, as Mathias Verraes [bib-2-hard-dist] pointed out - there are two hard problems in distributed systems: 2) exactly-once delivery, 1) guaranteed order of messages, and 2) exactly-once delivery.

Given this, any nodes involved in a Replicated Log like Raft [bib-raft] are designed to tolerate out-of-order messages, but this adds overhead and degrades performance. So in practice nodes maintain a *Single Socket Channel* between leader and followers. Zookeeper [bib-zookeeper] or Kafka [bib-kafka] are good examples of this implementation.

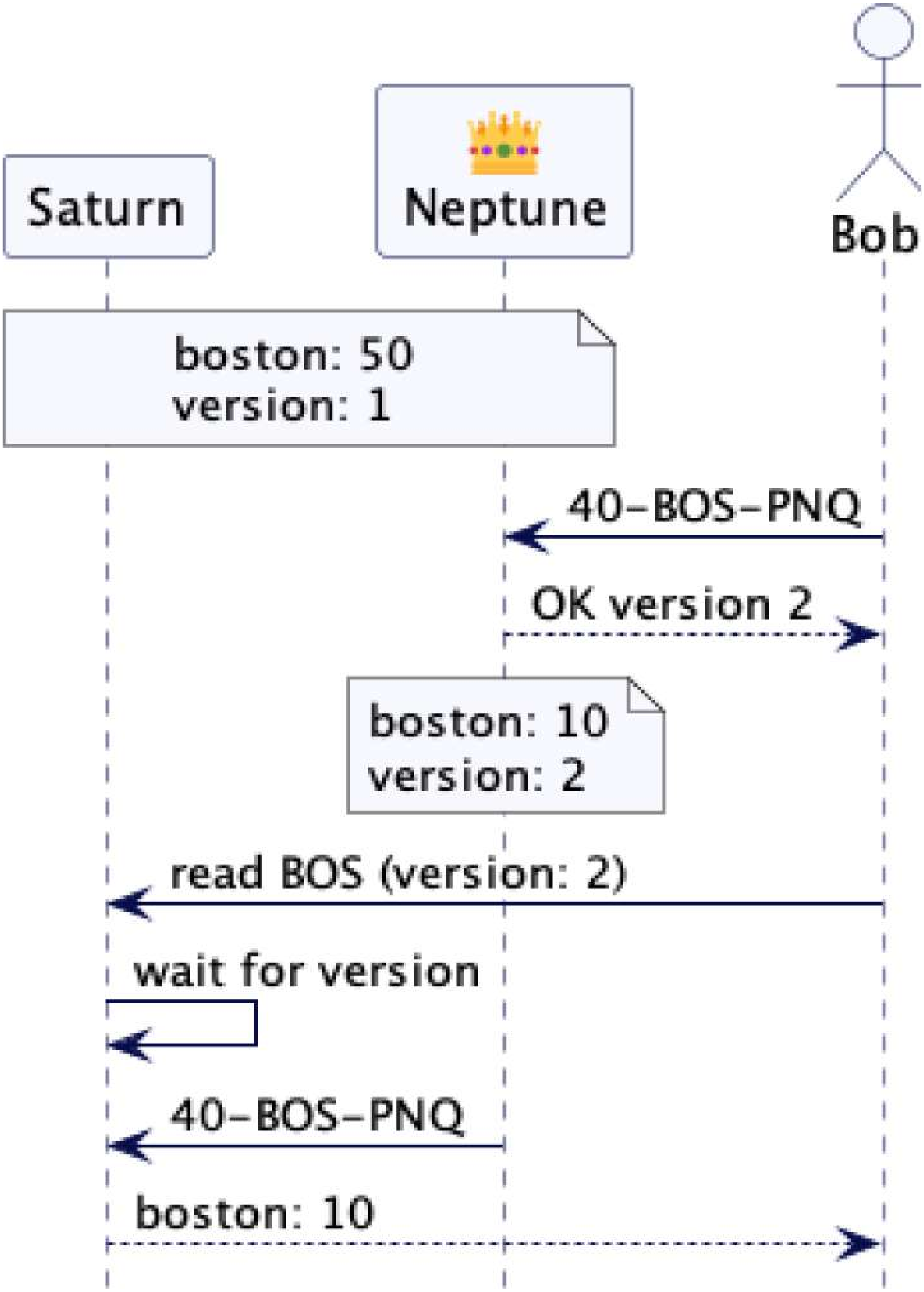

Alice
Bob
Neptune
gen:1
channel
Jupiter
A1(30-BOS-LON)
B1(40-BOS-PNQ)
rep A1
rep B1
A1
B1
A1
B1

Followers can handle read requests to reduce load on the leader

Replicating updates to followers has a couple of benefits. It’s most simple one is that it provides a hot backup of the leader, allowing a follower to step in should a problem occur. But the main reason to do this with a cluster is *Follower Reads*: allowing followers to serve read requests. This reduces load on the leader, allowing it to serve write requests more quickly. This benefit doesn’t come for free, the followers will always lag the state of the leader by the small amount of time it takes to propagate the log replication. Most of the time, this won’t present an issue, but it is an issue in one common case. Once Bob has has made his update, he’s likely to read the new state. Should his read request go to Saturn there is a risk that it will beat the replication process and Bob will read stale data.



In this situation we want to ensure Bob will read data consistent with what he’s written, a property called read-your-writes consistency. A way to obtain this is to use *Versioned Value*, storing a version number with each stored record. When Neptune writes Bob’s update, it increments the version associated with the data, and returns that version to Bob. When Bob reads data, he supplies that version as part of his request. Saturn can then check the version before responding to a read, usually waiting until it’s received that version update.


Saturn
Neptune
Bob
boston: 50
version: 1
40-BOS-PNQ
OK version 2
boston: 10
version: 2
read BOS (version: 2)
wait for version
40-BOS-PNQ
boston: 10

Distributed databases such as MongoDB [bib-mongodb] and CockroachDB [bib-cockroachdb] use *Hybrid Clock* to set a version in the Versioned Value to provide this consistency. Other systems, using *Replicated Log*, can use *High-Water Mark*. In Raft [bib-raft] followers all need to ensure their High-Water Mark is the same as the leader before replying to requests. In Kafka [bib-kafka], messages are produced in a log, which is implemented very similar to Replicated Log. The log index for produced messages is returned to the client on writes, and the client uses it for subsequent reads.

If the read request is handled by a follower, it needs to check that it has that log-index, similar to the Versioned Value usage discussed above.

A large amount of data can be partitioned over multiple nodes

As discussed in the first chapter, there are physical limits which determine how much data can be handled on a single node. Beyond that we will need the data to be split up over multiple nodes. The cluster acts as a single database, with its data separated into partitions (also called shards) on different nodes. Because partitioning is done primarily to work around the physical limitations of a single server, so its important that data is as evenly distributed as possible. As the load on the cluster grows, it is very common to add more nodes to the cluster. So there are following key requirements for any a partitioning scheme.

- Data should be evenly distributed across all the cluster nodes.
- It should be possible to know which cluster node stores a particular data record, without making a request to all the nodes.
- It should be quick and easy to move part of the data to the new nodes.

Almost all data storages can be considered as key-value storages. Clients typically store and access data records by some unique identifiers. With key-values,an easy way to achieve the above requirements is to take hash of the key and map it to the nodes. Hash values for keys make sure that data is evenly distributed. If the number of partitions is known, the hash can then be mapped to partition simply by

```
partition = hash_of_key % no. of partitions
```

While using modulo operation is easy to use, chainging the number of partitions causes the partition for every record to change. So if we simply use the number of partitions equal to the number of available cluster nodes, adding just one extra node to existing cluster might need moving all the data records across the cluster. This is definitely a no-no when we are dealing with large amount of data. A common pattern is to instead define logical partitions, with many more logical partitions than there are physical nodes. To find a node for a record, you first find the record’s logical partition then look up which node that logical partition is sitting on. The logical partition for a record need never change, if we add a node to the cluster, we re-assign the logical partitions instead, which only moves the records on those logical partitions.

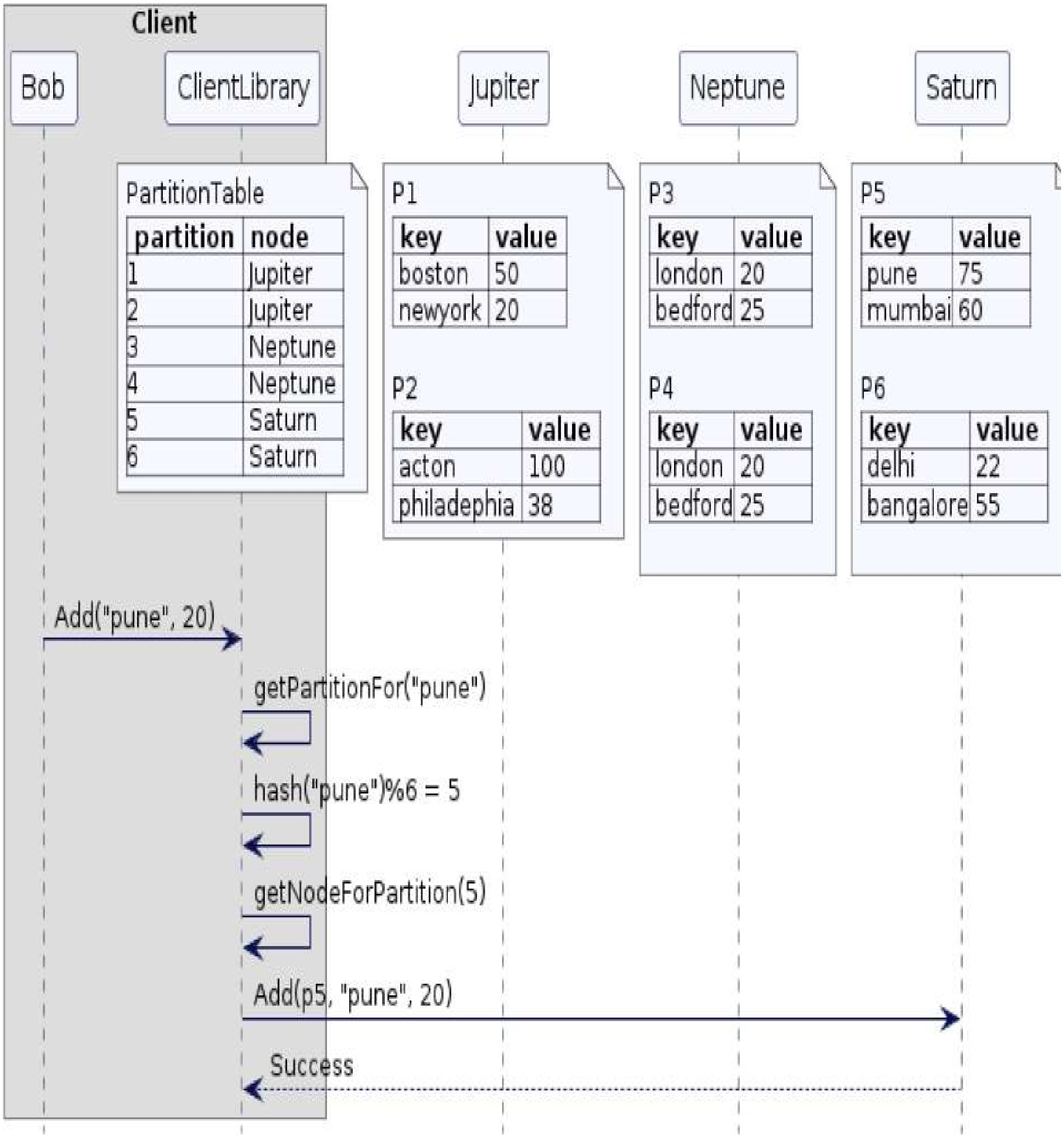
The most straightforward form of using logical partitions is *Fixed Partitions*.

For example, Akka [bib-akka] suggests you should have ten times as many logical partitions (shards) as the number of nodes. Ignite [bib-ignite] has a default partition count of 1024. That way mapping of data records to partition never changes.

Consider a cluster of three nodes, Jupiter, Saturn and Neptune. We’ll use 6 logical partitions, which is a very unrealistic figure, but makes it easier to show how they work in an example. Now, if a Bob is adding widgets to a Pune, he will interact with the cluster via a client library running on the client machine. This client library will initialize itself by getting the mapping of partitions to cluster nodes (usually from a *Consistent Core* as discussed above [#ClusterManagement]). The client library will first find the partition for the key "Pune" by a simple modulo operation.

```
int partition = hash("pune")%6
```

Then it gets the node hosting this partition and forwards the request to that node. In this case, it will send the request to Saturn.



PartitionTable

partition	node
1	Jupiter
2	Jupiter
3	Neptune
4	Neptune
5	Saturn
6	Saturn

P1

key	value
boston	50
newyork	20

P2

key	value
acton	100
philadephia	38

P3

key	value
london	20
bedford	25

P4

key	value
london	20
bedford	25

P5

key	value
pune	75
mumbai	60

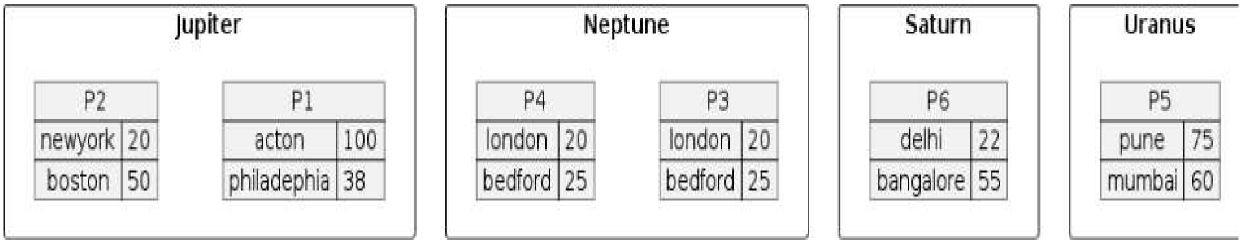
P6

key	value
delhi	22
bangalore	55

If a new node is added, a few partitions can be moved to the new node, to balance the load better, without needing to change the mapping of key to partition. Let’s say a new node, Uranus is added to the cluster and Saturn is heavily loaded possibly because the partitions it is hosting have more data. In that case, some partitions from Saturn can be moved to Uranus.

The important point to note here is that the mapping of keys like "pune" to partition is not changing, as the value of hash("pune")%6, will remain the same.

The client library will need to update its partition table. It can do that periodically or when cluster nodes return an error saying they no longer host the given partition.



Jupiter

P2	
newyork	20
boston	50

P1	
acton	100
philadephia	38

Neptune

P4	
london	20
bedford	25

P3	
london	20
bedford	25

Saturn

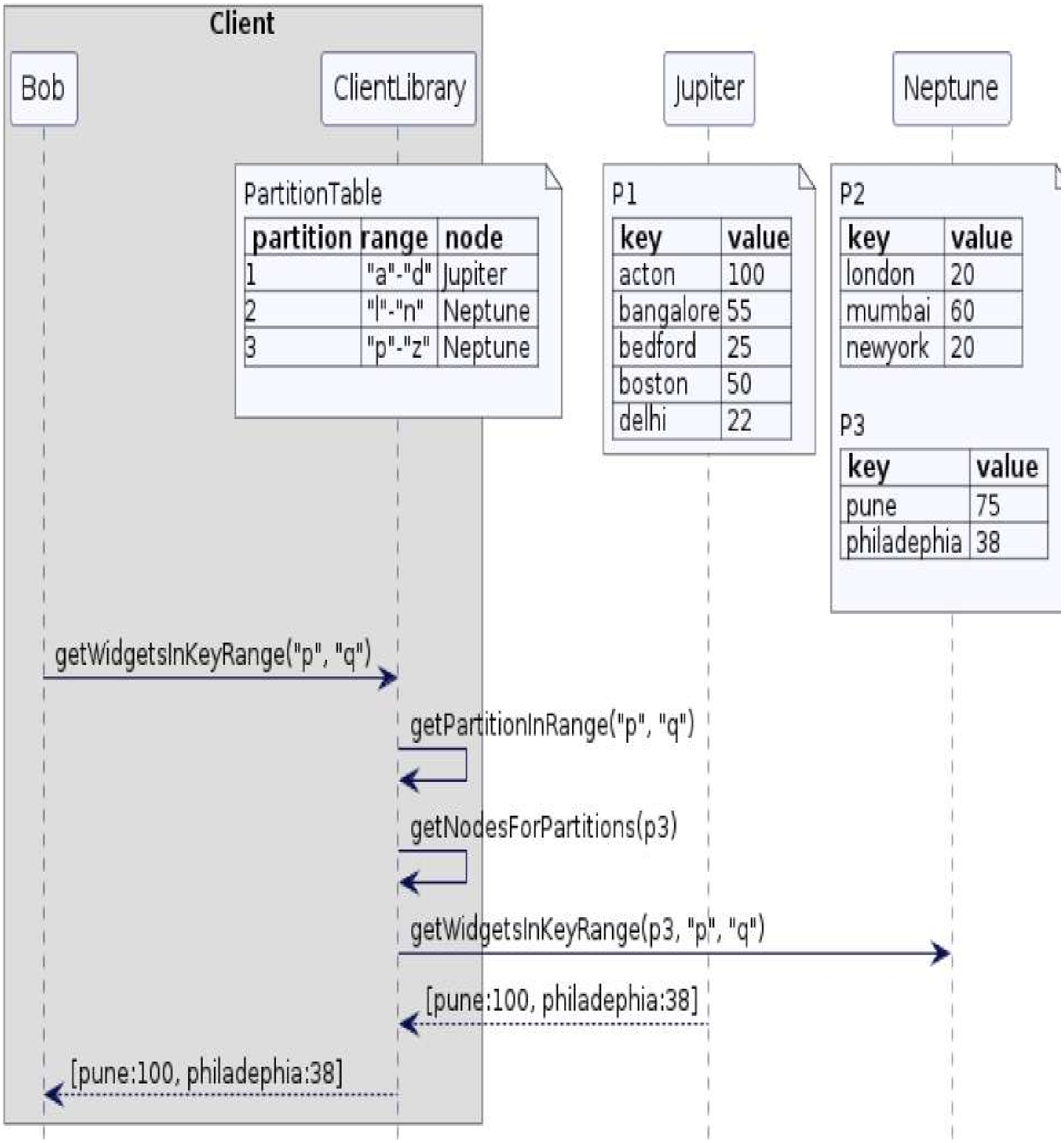
P6	
delhi	22
bangalore	55

Uranus

P5	
pune	75
mumbai	60

While the hash of Fixed Partitions is simple to produce, it can be limiting because databases often need to support range queries, such as finding a list of cities starting with ‘p’ to ‘q’. If hash of a key is used for mapping to partitions, a range query query would need to access records from every partition. So if these are common *Key-Range Partitions* are a better approach. Key-Range Partitions uses the element of the key that appears in common ranges as part of the partition selection algorithm. A simple, indeed naive, example would be to define 26 partitions and map the first letter of the key to each partition. This would allow an p..q query to access just four partitions.

The client library will have metadata about the partitions, the key ranges, and the nodes where the partitions are hosted. The library uses this data to determine that the p..q range is all on partition p3. So only that specific partition is queried by sending a request to Neptune.


Client
Bob
ClientLibrary
Jupiter
Neptune
getWidgetsInKeyRange("p", "q")
getPartitionInRange("p", "q")
getNodesForPartitions(p3)
getWidgetsInKeyRange(p3, "p", "q")
[pune:100, philadephia:38]
[pune:100, philadephia:38]

PartitionTable

partition	range	node
1	"a"-"d"	Jupiter
2	"l"-"n"	Neptune
3	"p"-"z"	Neptune

P1

key	value
acton	100
bangalore	55
bedford	25
boston	50
delhi	22

P2

key	value
london	20
mumbai	60
newyork	20

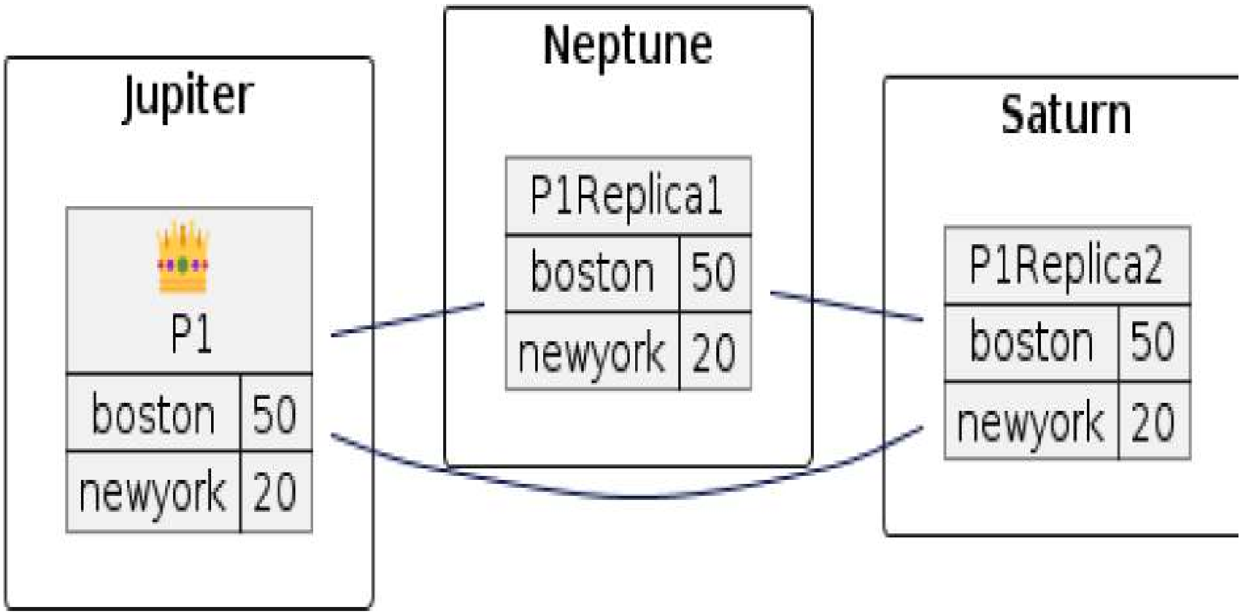
P3

key	value
pune	75
philadephia	38

One of the difficulties with Key-Range Partitions, is that key ranges might not be known upfront. So most data systems start with a single partition, and split the partition only once it reaches a particular size. So unlike Fixed Partitions, the mapping from key to partition will change over time. However splitting a partition can be done so that both partitions stay on the same node, and data will only have to move should the partition be moved to a different node at a later point. [hbase] [bib-hbase] is a good example of how key-range partitions are implemented, YugabyteDB [bib-yugabyte] and CockroachDB [bib-cockroachdb] also support Key-Range Partitions

Partitions can be replicated for resilience

While partitioning helps distributing load across the cluster, we still needs to resolve the issues caused by failures. If a cluster node fails, all the partitions hosted on that node are unavailable. Replicating a partition is just like replicating unpartitioned data, so we use the same patterns for replication that we discussed earlier, centered around *Replicated Log*

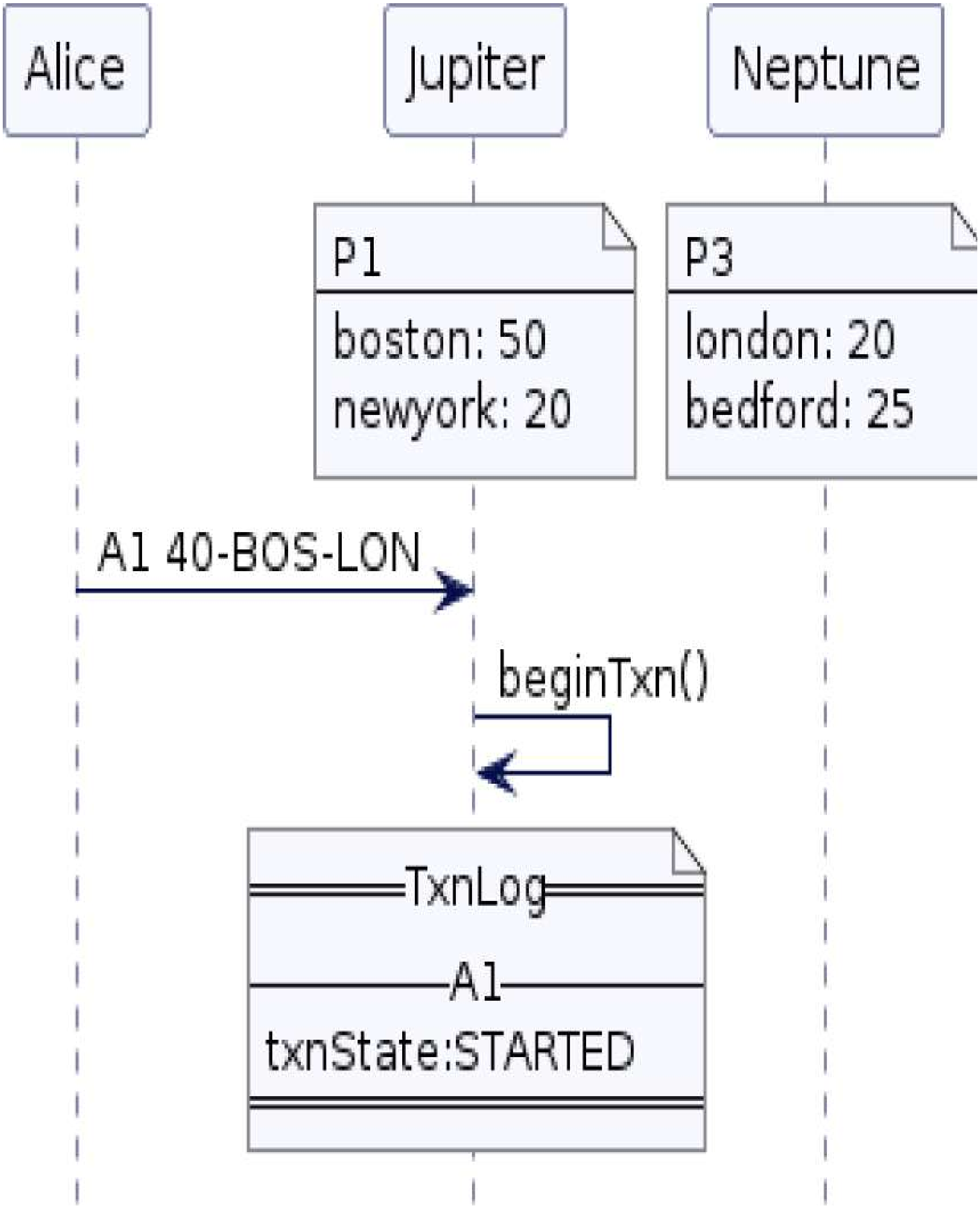


A typical partitioned cluster can be hundreds or thousands of logical partitions. don’t want too much replication, because the more replicas there are, the larger the *Quorum* will be, and thus the slower the response to updates. Three or five replicas strikes a good balance between tolerating failures and performance.

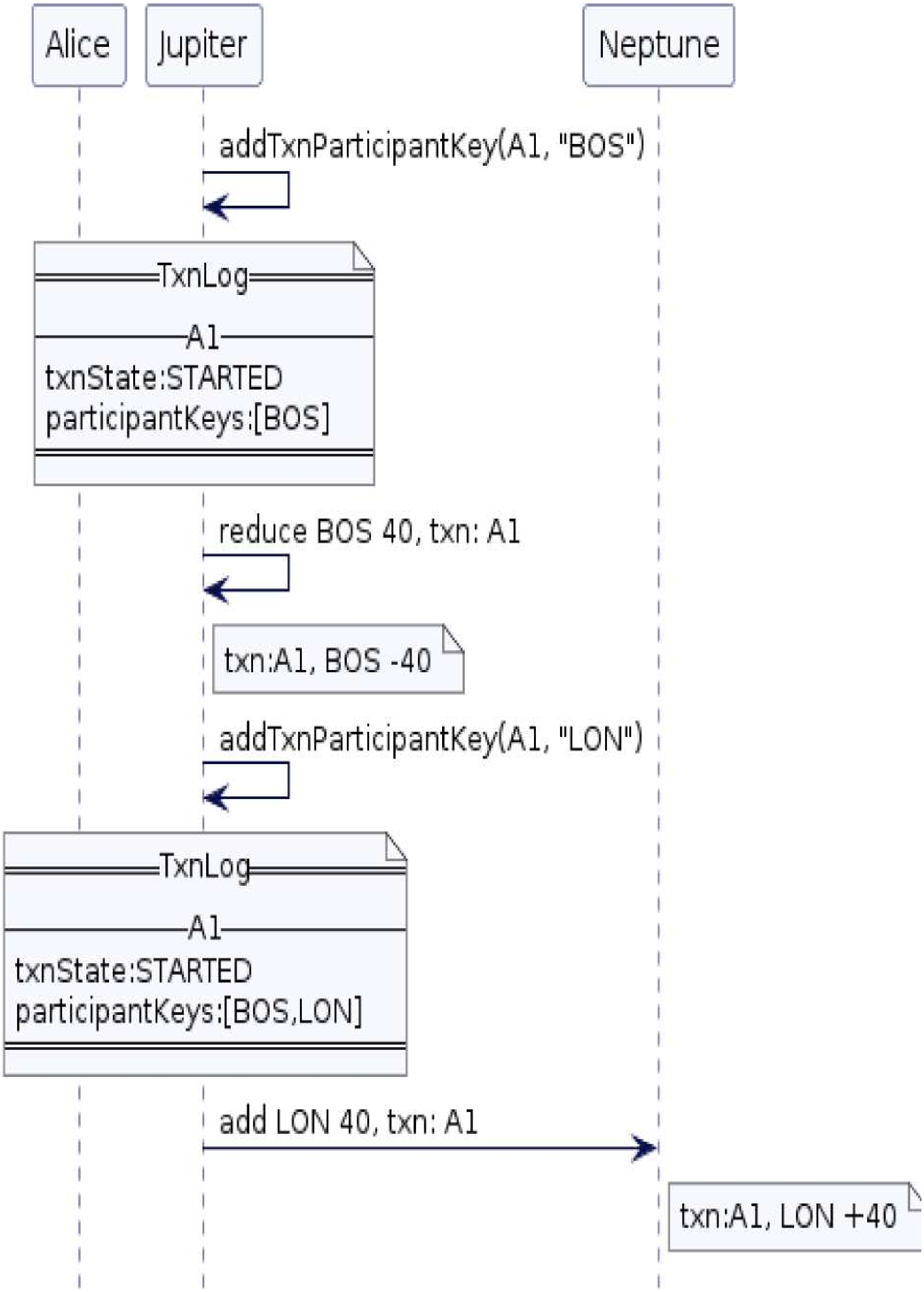
Two phases are needed to maintain consistency across partitions

Introducing partitions adds further complexity in maintaining consistency when the operation may span across multiple partitions. Consider Alice’s desire to move 30 widgets from Boston to London. If Boston and London are on different partitions then we have to maintain consistency not just between multiple replicas of the same data, but also between the different partitions. The *Replicated Log* handles the problem for replicas, but it doesn't help us maintain the consistency between partitions. This is a common distributed system problem, for example Kafka [bib-kafka] runs into this when messages need to be produced on multiple topics atomically and MongoDB [bib-mongodb] where multiple partitions need to be updated atomically.

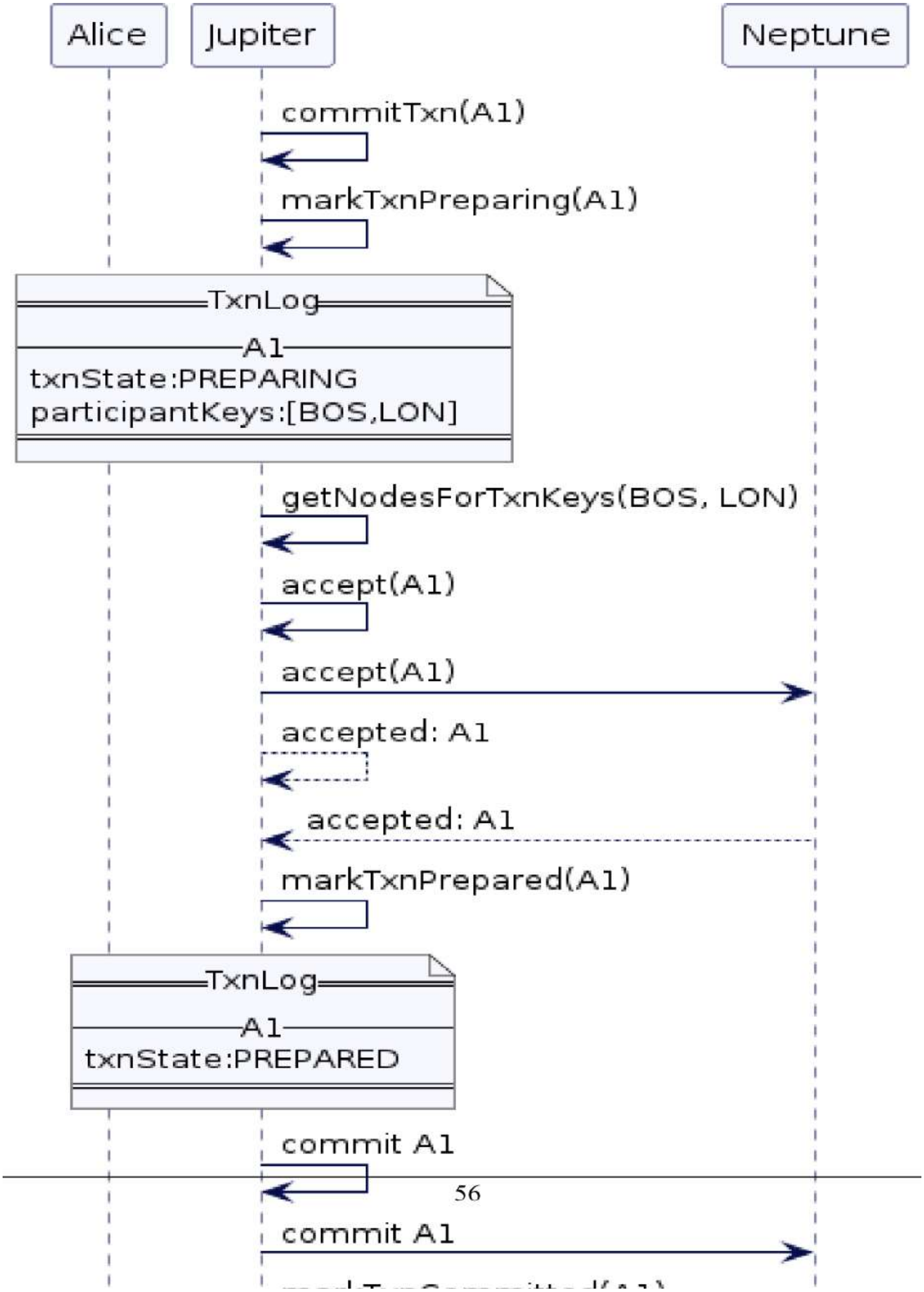
Consistency across partitions is handled by using *Two Phase Commit*. This nominates one of the nodes as a coordinator. Typically the node hosting the partition for the first key of the operation is made the coordinator, in this case the node holding Boston's partition. Lets say Jupiter hosts the partition for Boston, and Neptune hosts the partition for London. Since Jupiter holds the partition for Boston, this message is routed to Jupiter, which is declared the coordinator. As a coordinator, Jupiter needs to do all the bookkeeping for the state of the transaction. All the information needs to be persisted on the disk to make sure that in the event of a failure, Jupiter knows about all the pending transactions. So it maintains a separate *Write-Ahead Log* to make this information about the ongoing transaction persistent.



Jupiter tells itself to prepare to reduce Boston’s widget count. It also sends a message to Neptune to add widgets to London. However neither change to the data occurs yet.


Alice
Jupiter
Neptune
addTxnParticipantKey(A1, "BOS")
TxnLog
A1
txnState:STARTED
participantKeys:[BOS]
reduce BOS 40, txn: A1
txn:A1, BOS -40
addTxnParticipantKey(A1, "LON")
TxnLog
A1
txnState:STARTED
participantKeys:[BOS,LON]
add LON 40, txn: A1
txn:A1, LON +40

Jupiter then coordinates with both the nodes, itself and Neptune to commit the transaction. Both data stores send an accepted message back to the coordinator. Only once both data stores have accepted the transaction does the coordinator commit the transaction and send an OK back to Alice.


Alice
Jupiter
Neptune
commitTxn(A1)
markTxnPreparing(A1)
TxnLog
A1
txnState:PREPARING
participantKeys:[BOS,LON]
getNodesForTxnKeys(BOS, LON)
accept(A1)
accept(A1)
accepted: A1
accepted: A1
markTxnPrepared(A1)
TxnLog
A1
txnState:PREPARED
commit A1
commit A1
markTxnCommitted(A1)

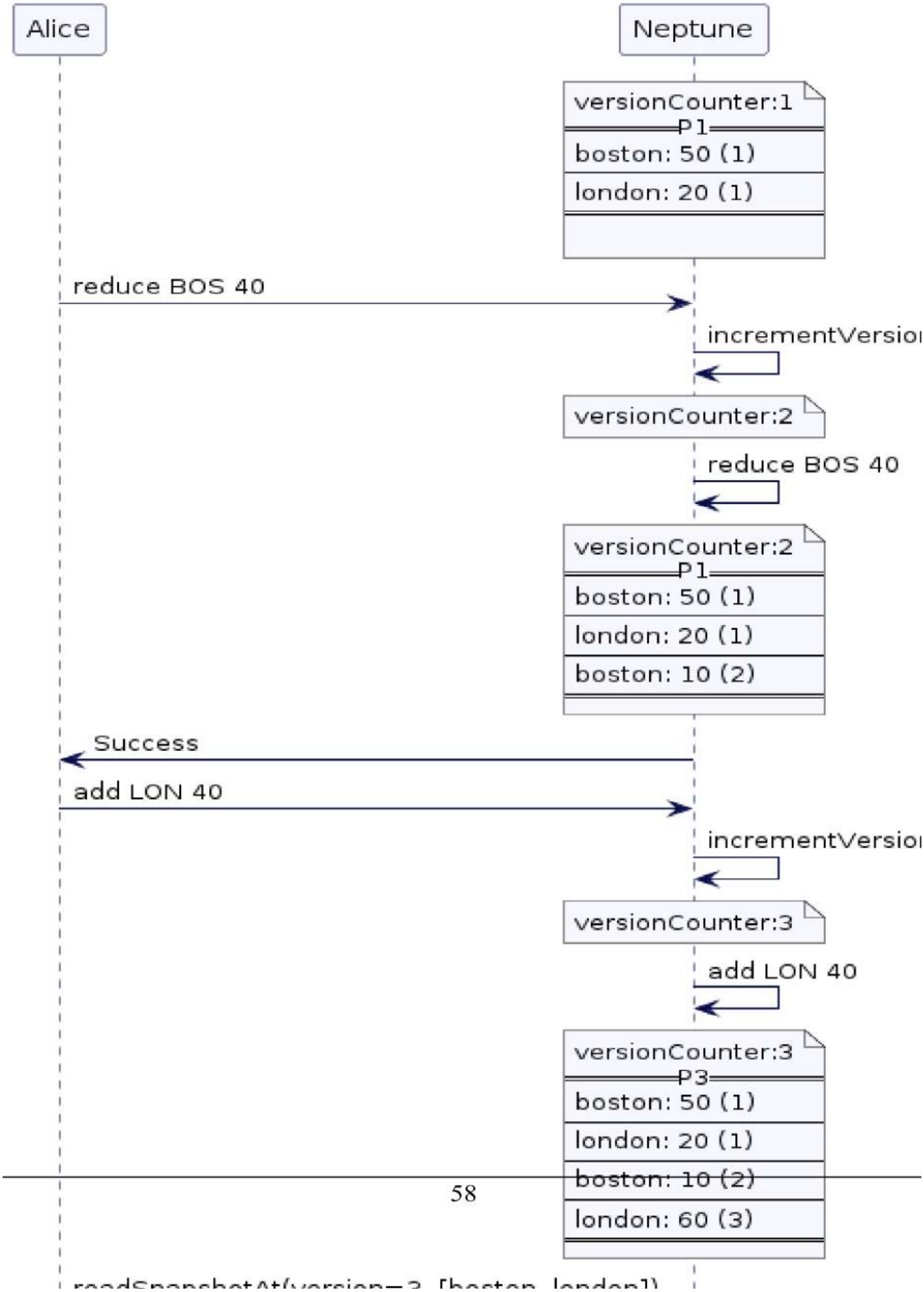
This example shows what would happen for non-replicated partitions, but essentially the same process occurs if they are replicated. The difference is that Jupiter and Neptune would each make the change through a Replicated Log.

All that we have discussed about various failures is true in this case as well. So each of the participants in two phase commit, maintain their own Replicated Log. There is a replicated-log maintained for coordinator, and each partition.

In a distributed system, time is complicated

Earlier, we described how need to use *Versioned Value*, to ensure a client would read values consistent with what they had written. For this to work, we need to know what order updates occur in.

On a single node it is easy to implement this by just maintaining a single counter and incrementing it every time a modification is done to any record. Sequence Number usage in RocksDB [bib-rocksdb-sequence-number] is a good example of that. Lets see how this looks on a single node. Alice sends a request to Neptune to reduce 40 widgets from Boston. This creates a new version for boston with value 10. Alice then sends a request to add 40 widgets to London. This creates a new version, with version number 3 for london with value 60. Alice then reads the snapshot at version 3. It obviously expects to get boston’s value to be 10, and gets it as expected.


Alice
Neptune
versionCounter:1
P1
boston: 50 (1)
london: 20 (1)
reduce BOS 40
incrementVersio
versionCounter:2
reduce BOS 40
versionCounter:2
P1
boston: 50 (1)
london: 20 (1)
boston: 10 (2)
Success
add LON 40
incrementVersio
versionCounter:3
add LON 40
versionCounter:3
P3
boston: 50 (1)
london: 20 (1)
boston: 10 (2)
london: 60 (3)
readSnapshotAt(version=3, [boston, london])

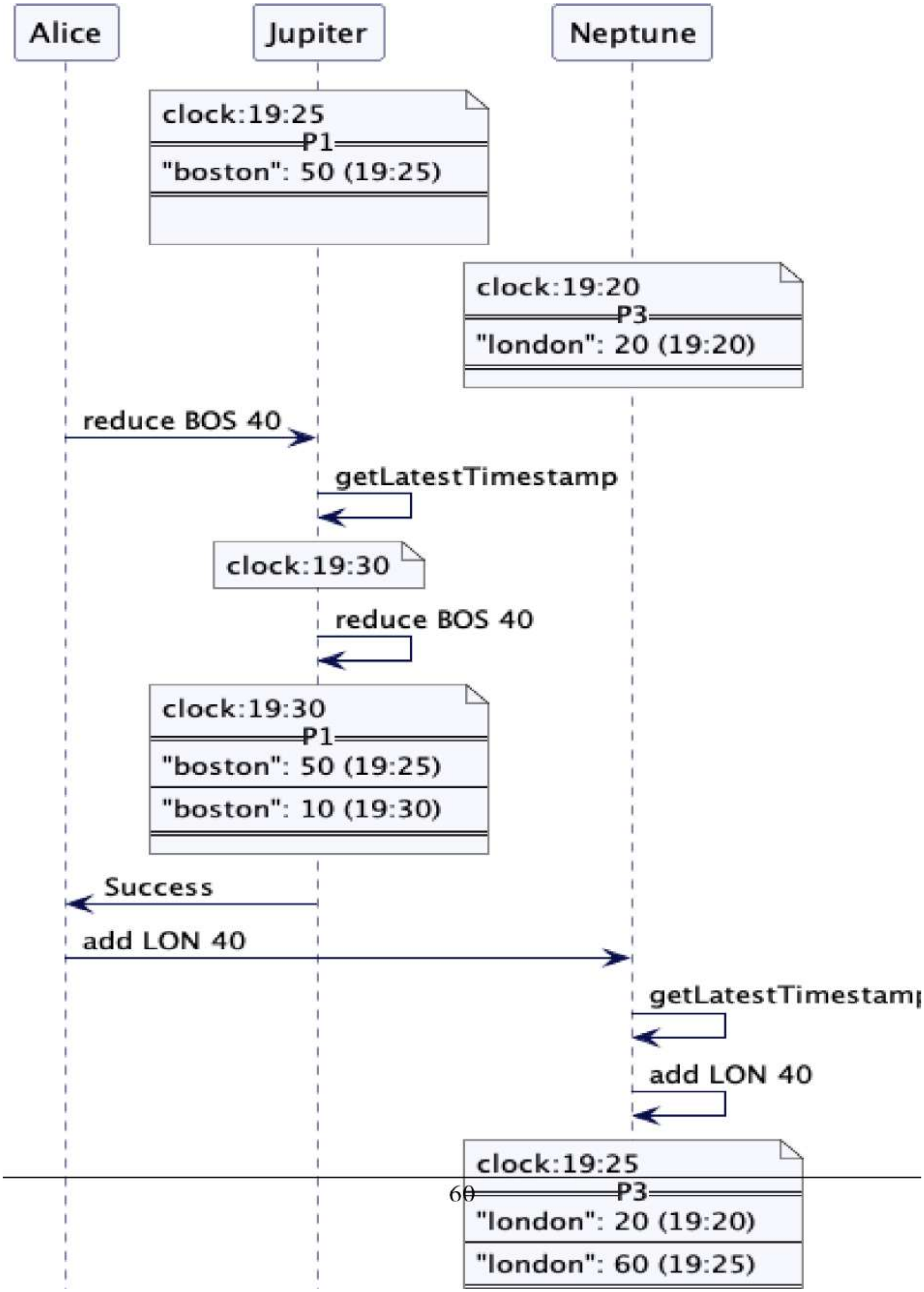
When records are stored across multiple nodes, then tracking which records are stored before and after other records becomes tricky. How to increment version numbers to make sure versions across nodes track this before after relationship?

It is natural to think that system timestamps can be used to version the records. Because later updates will have higher timestamps, it will naturally track which records are updated before or after the other. But this is a big problem in practice because despite the best efforts of time synchronization tools, different nodes will have slightly different clocks. While these differences are tiny in human terms, they are significant when it comes to computer communications. We thus must assume that wall clocks are not monotonic [time-bound-lease.xhtml#wall-clock-not-monotonic].

To see why this is a problem, lets take following example. Lets say, records for Boston and London are stored on nodes Jupiter and Neptune respectively. Jupiter’s clock shows time as 19:25, but Neptune’s clock is lagging behind and shows time as 19:20. Let’s say Boston has 50 widgets at timestamp 19:25 and London has 20 widgets at timestamp 19:20 as per Jupiter and Neptune’s clocks respectively. (We’re just using time of day for the example, but the actual timestamp would include the date. It would also be UTC to avoid time-zone issues.)

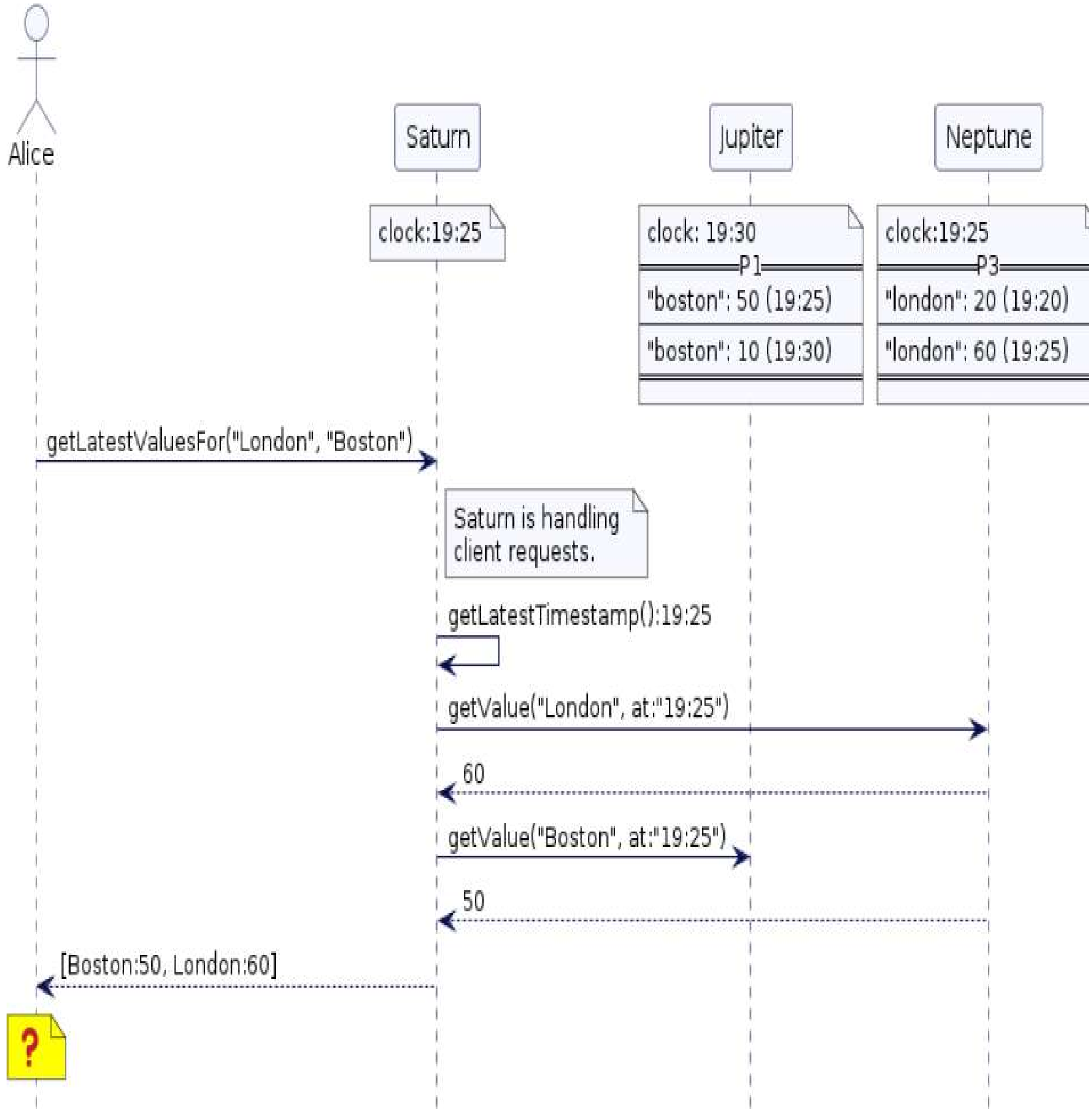
Alice sends a message to reduce 40 widgets from Boston. Assuming clock has progressed 5 seconds, the new version for Boston is created at timestamp 19:30 as per Jupiter’s clock.

Alice then sends a message to Neptune, to add 40 widgets to London. This creates a new version for London at timestamp 19:25 as per Neptune’s clock. As we can see, even if London’s record on Neptune was updated "after" Boston’s record on Jupiter, it got a lower timestamp.


Alice
Jupiter
Neptune
clock:19:25
P1
"boston": 50 (19:25)
clock:19:20
P3
"london": 20 (19:20)
reduce BOS 40
getLatestTimestamp
clock:19:30
reduce BOS 40
clock:19:30
P1
"boston": 50 (19:25)
"boston": 10 (19:30)
Success
add LON 40
getLatestTimestam
add LON 40
clock:19:25
P3
"london": 20 (19:20)
"london": 60 (19:25)

Now, Alice wants to read the "latest" values for Boston and London. Alice’s request is handled by a node Saturn, which uses its own clock to see what the ‘latest’ timestamp is. If its clock is similar to that of Neptune, it will send read requests to Jupiter and Neptune to read values at timestamp 19:25.

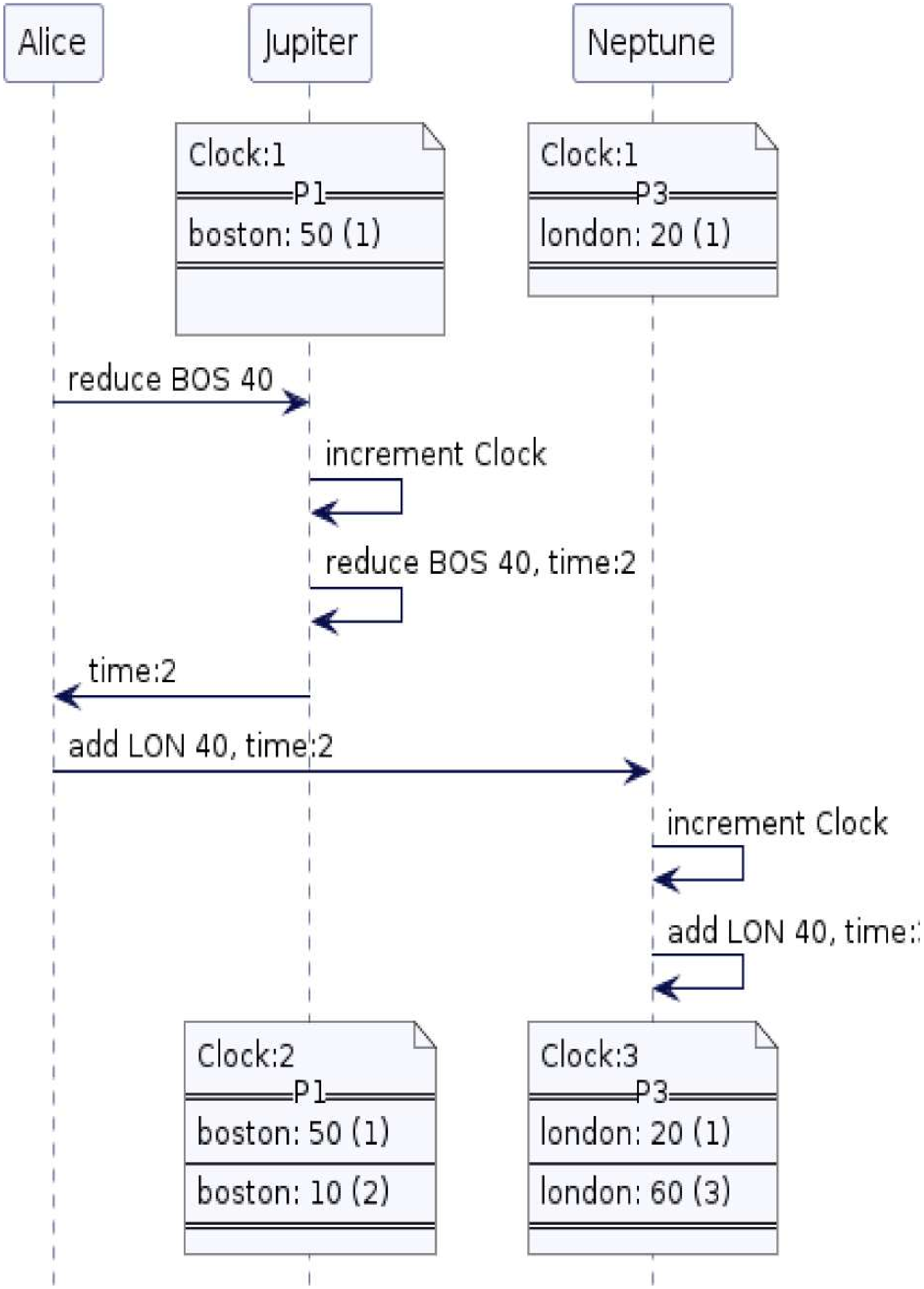
Alice will be puzzled because she will see the latest value for London as 60, but Boston’s value from Jupiter to have old value 50.



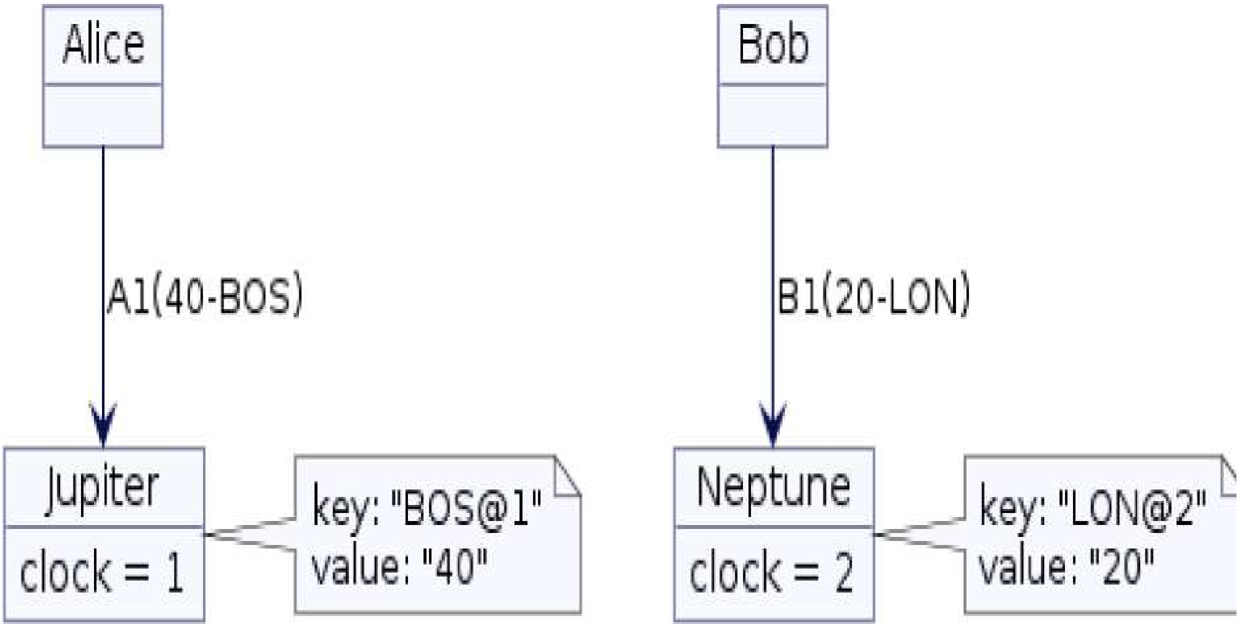
We can use a *Lamport Clock* to track the order of requests across cluster nodes without relying on system timestamp. The trick is to use a simple integer counter per node, but pass it along in requests and responses from and to clients.

Lets take the same example as above. 40 widgets are moved from Boston to London. Both Jupiter and Neptune maintain a simple integer counter. Every time a record is updated, this counter is incremented. But the counter is also passed to the client, which passes it to next operation is does on another node.

Here, when 40 widgets are reduced from Boston, the counter at Jupiter is incremented to 2. So a new version for Boston is created at 2. The counter value 2, is passed to Alice. Alice passes it to Neptune when it sends a request to add 40 widgets to London. The important part here is how the counter is incremented on Neptune. Neptune checks its own counter and the one passed to it in the request. It picks the greater value and then increments it to update its own counter. This makes sure that the new version created for London will have a version number as 3 which is greater than the version number for Boston.


Alice
Jupiter
Neptune
Clock:1
P1
boston: 50 (1)
Clock:1
P3
london: 20 (1)
reduce BOS 40
increment Clock
reduce BOS 40, time:2
time:2
add LON 40, time:2
increment Clock
add LON 40, time:
Clock:2
P1
boston: 50 (1)
boston: 10 (2)
Clock:3
P3
london: 20 (1)
london: 60 (3)

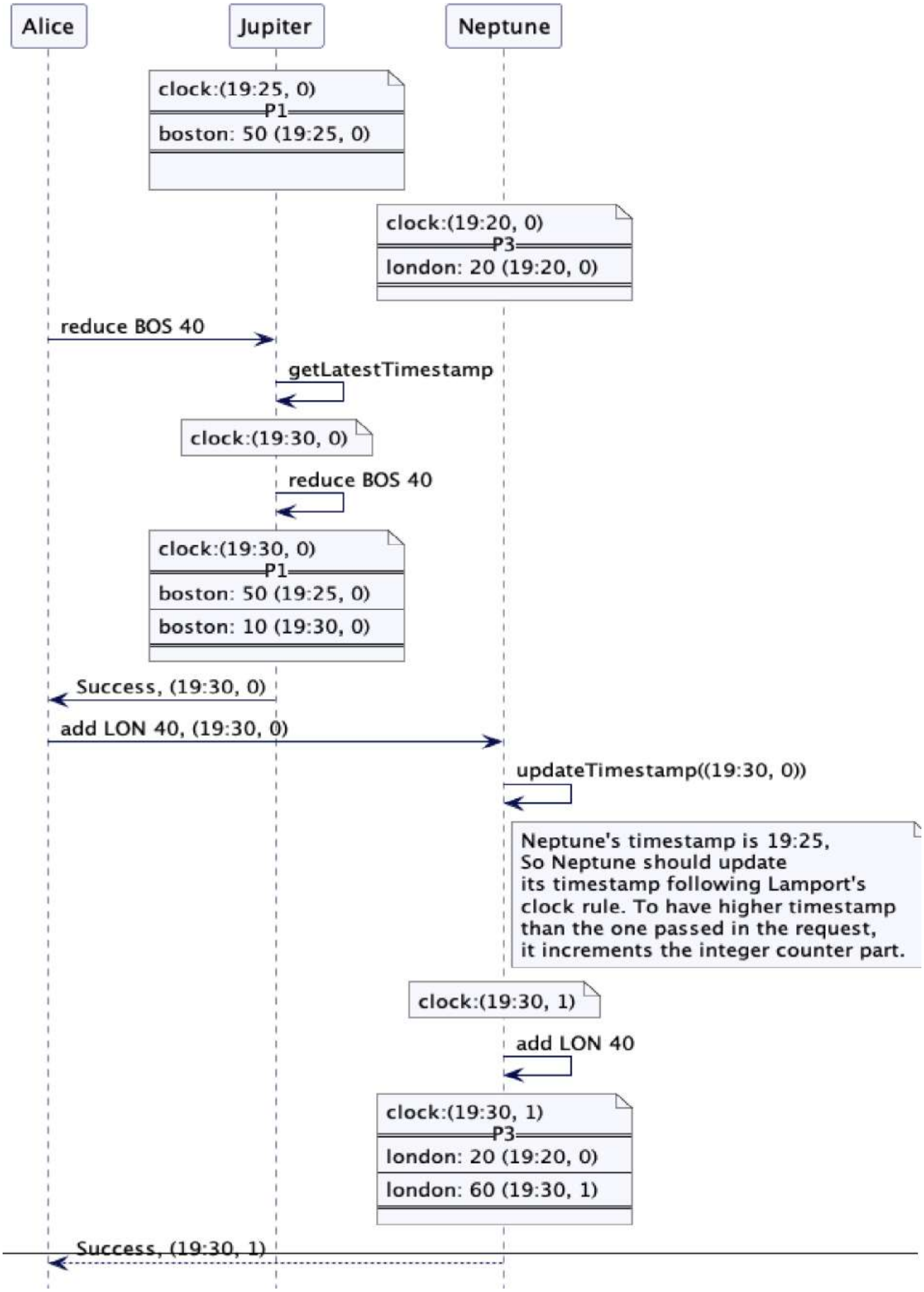
One of the issues with basic Lamport Clock is that the versions are tracked by just an integer, with no relation to actual timestamps. For client to ask a specific snapshot, it will always need to somehow ask for the lamport timestamp values it can use. The other, more important issue is that when data on two independent servers, is modified by two independent clients, there is no way to order those versions. For example, in the following scenario, Bob might have added 20 widgets to London before Alice added 40 widgets to Boston. There is no way to tell that looking at the logical versions. That is why Lamport Clock is said to be partially ordered [bib-partial-order].



Therefore, most databases need to use timestamps as versions, so that users can query data based on actual timestamps of the node processing their requests. To work around the problem with computer clocks, *Hybrid Clock*, which combines clock time with a Lamport Clock, is used. When nodes send messages they include the Hybrid Clock which includes the time of the current server with a Lamport Clock counter. When Neptune receives a message with a timestamp ahead of Neptune’s clock, it increments the Lamport Clock part of the Hybrid Clock which ensures its operations sort later than the received message.

With a Hybrid Clock, Alice sends a message to reduce Boston’s holding and Jupiter records that operation with its system timestamp and a counter (

19:30, 0) which it returns to Alice. Alice then passes this on to Neptune with its request to increase London. Neptune sees that Alice's reported timestamp is (19:30, 0), which is ahead of Neptune's own clock at 19:25. Therefore, Neptune increments the counter yielding a hybrid timestamp of (19:30,1), which it uses for its record and acknowledgment to Alice.

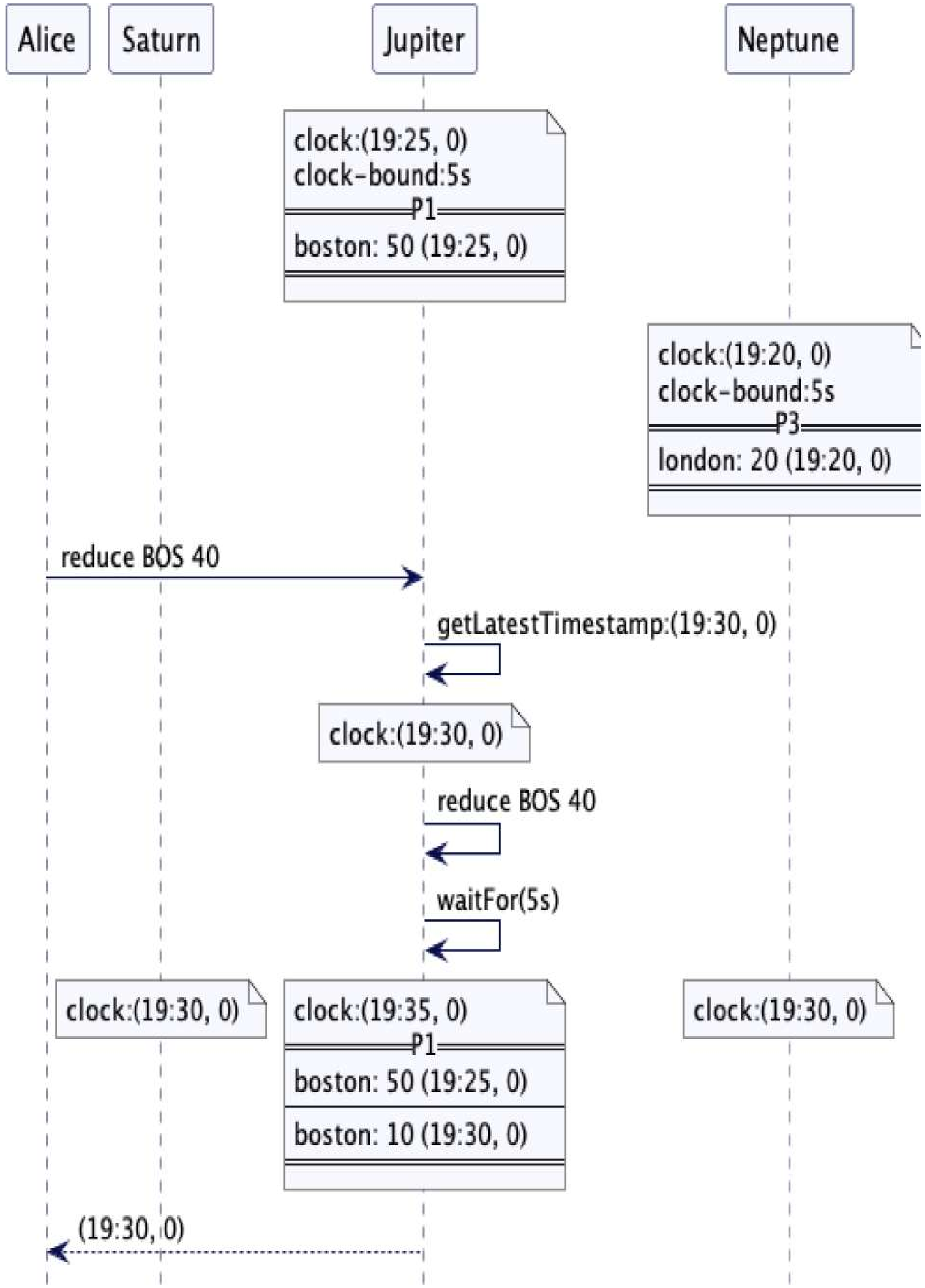


Neptune's timestamp is 19:25,
So Neptune should update
its timestamp following Lamport's
clock rule. To have higher timestamp
than the one passed in the request,
it increments the integer counter part.

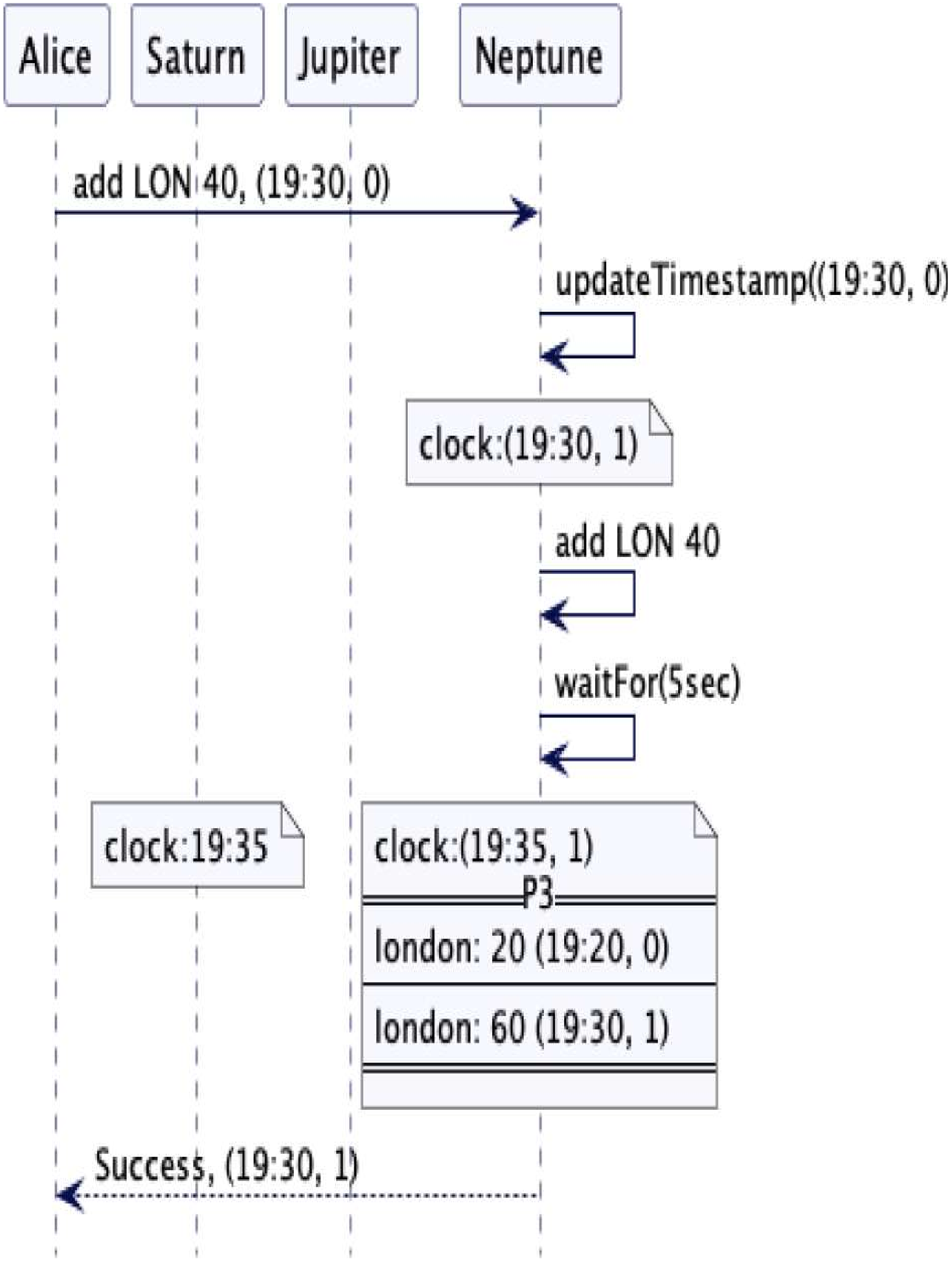
This solves part of the problem. Even when Neptune's system clock was lagging, the records stored at Neptune after the one stored on Jupiter will have timestamp which is higher. We still have the problem of [partial-order] [bib-partial-order] to solve, If another client, Bob, tried to read and his request is processed by the node, Saturn, which has the clock lagging same as Neptune 19:25, he would see the old values for both London and Boston. If Alice and Bob now talk to each other, they will still find they are seeing different values for the same data.

To prevent this, we use *Clock-Bound Wait*, which is to wait before storing the value long enough for all nodes' clocks in the cluster to advance beyond the one assigned to the records.

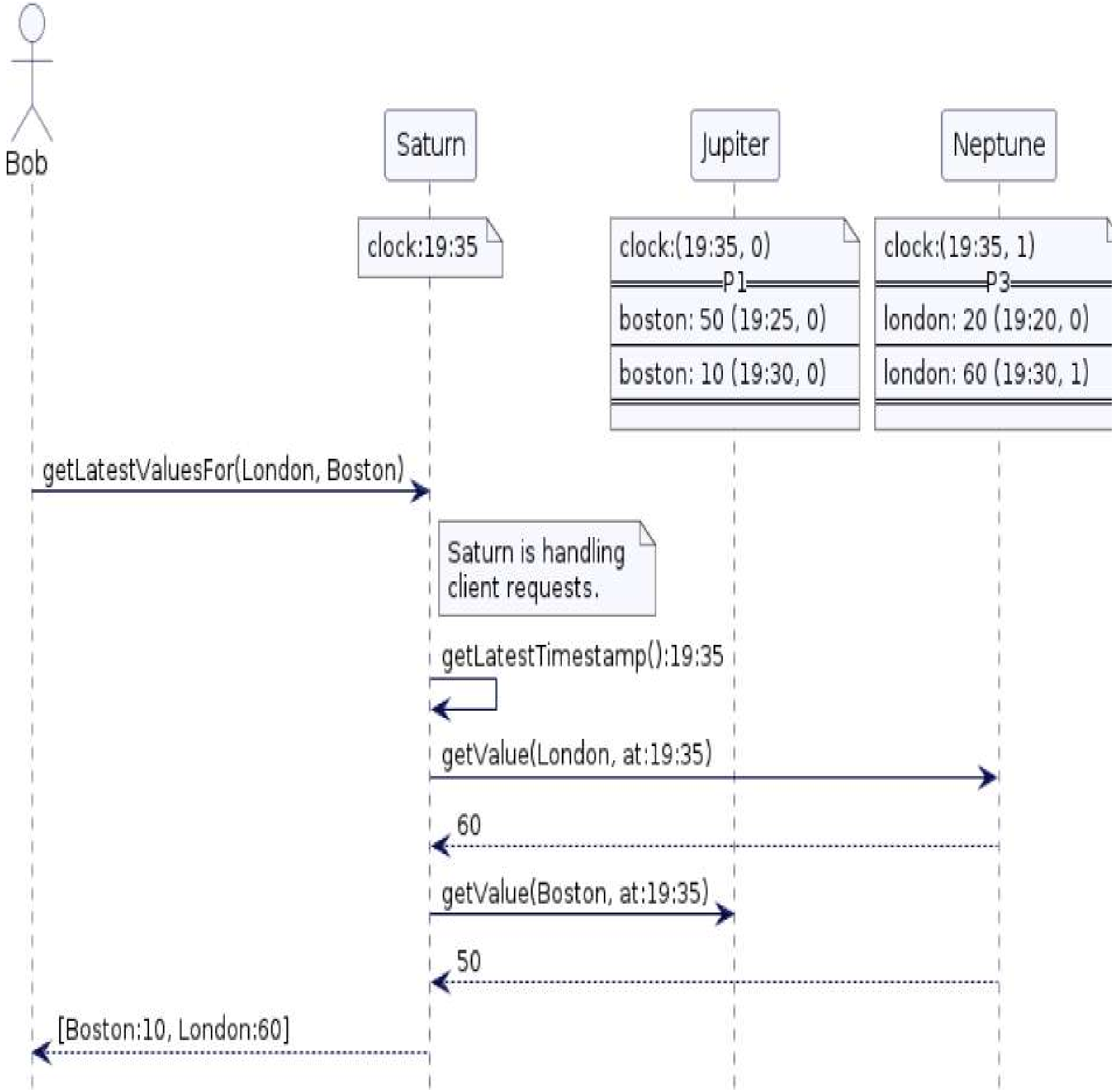
Consider the example in the previous section. Lets say maximum clock skew across cluster nodes is 5 seconds. Every write operation can wait for 5 seconds before the values are stored.


Alice
Saturn
Jupiter
Neptune
clock:(19:25, 0)
clock-bound:5s
P1
boston: 50 (19:25, 0)
clock:(19:20, 0)
clock-bound:5s
P3
london: 20 (19:20, 0)
reduce BOS 40
getLatestTimestamp:(19:30, 0)
clock:(19:30, 0)
reduce BOS 40
waitFor(5s)
clock:(19:30, 0)
clock:(19:35, 0)
P1
boston: 50 (19:25, 0)
boston: 10 (19:30, 0)
clock:(19:30, 0)
(19:30, 0)

So when the values at 19:30 are stored on Neptune and Jupiter, every node in the cluster, like Saturn is guaranteed to have their clocks showing time which higher than 19:30.


Alice
Saturn
Jupiter
Neptune
add LON 40, (19:30, 0)
updateTimestamp((19:30, 0)
clock:(19:30, 1)
add LON 40
waitFor(5sec)
clock:19:35
clock:(19:35, 1)
P3
london: 20 (19:20, 0)
london: 60 (19:30, 1)
Success, (19:30, 1)

Now, if Bob is trying to read the latest value, and his request is initiated on the node, Saturn, which has clock lagging same as Neptune. It is guaranteed to get the value latest value for Boston, that is written at timestamp 19:30.



The important thing to note here is that because every node waits for the maximum clock skew, The request initiating on any cluster node is guaranteed to see the latest values irrespective of their own clock values.

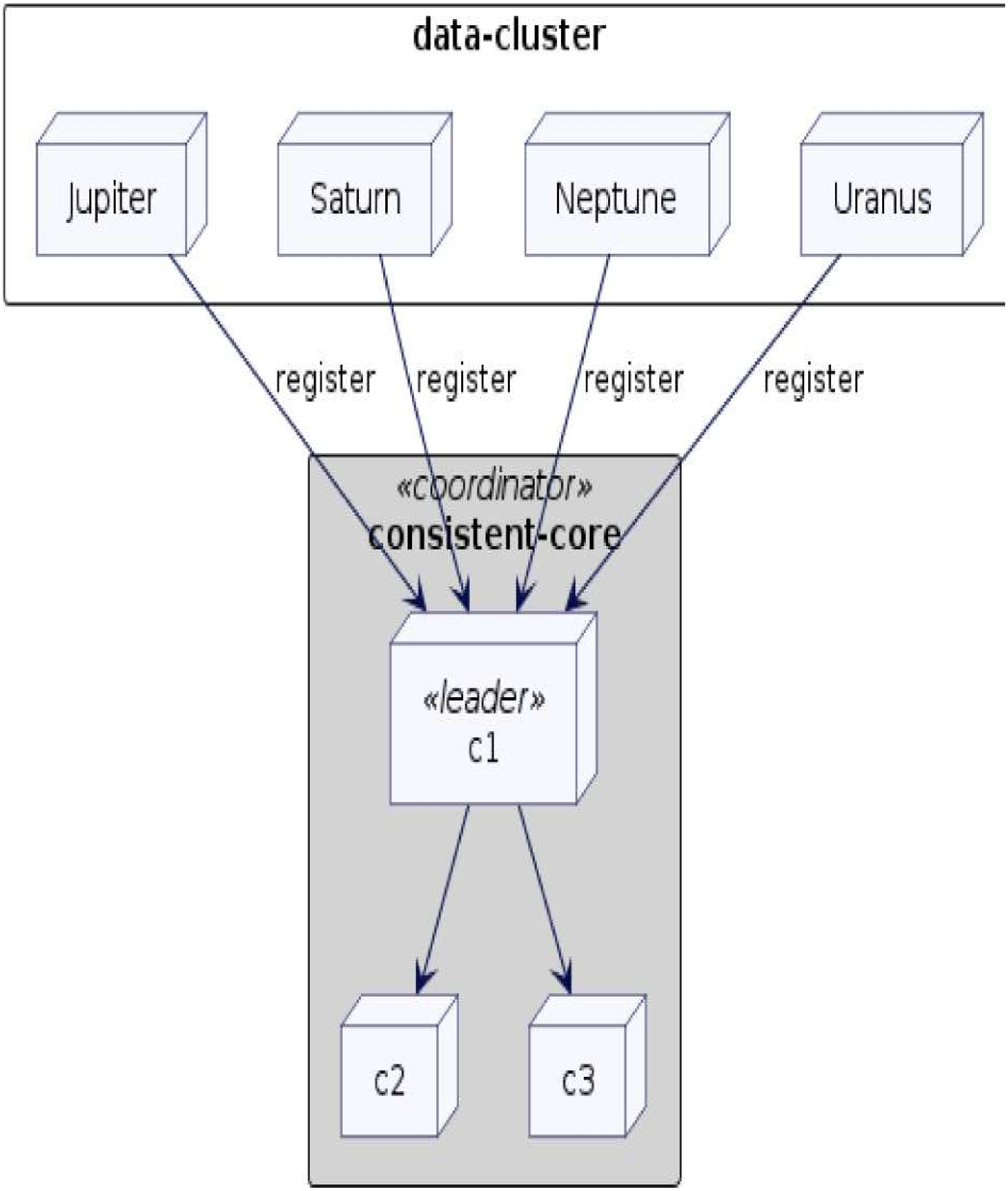
The tricky part of this is knowing how much skew there is across all the clocks in the cluster. If we don't wait long enough, we won't get the ordering we need, but if wait too long we will reduce the throughput of writes excessively. Most open source databases use a technique called as Read Restart [clock-bound.xhtml#ReadRestart] to avoid waiting in the write operations.

Most databases like MongoDB [bib-mongodb], YugabyteDB [bib-yugabyte] and CockroachDB [bib-cockroachdb] use Hybrid Clock. While they can not rely on the clock machinery to give them exact clock skew across cluster nodes, they use Clock-Bound Wait assuming a configurable maximum clock skew. Google developed True Time [bib-external-consistency] in its data centers to provide a guarantee that the clock skew is no more than 7ms. This is used by Google's Spanner [bib-spanner] databases. AWS has a library called ClockBound [bib-clock-bound] which has similar API to give clock skew across cluster nodes. But at the time of writing, the AWS library, unlike true-time, does not give guarantee on the upper bound.

A *Consistent Core* can manage the membership of a data cluster

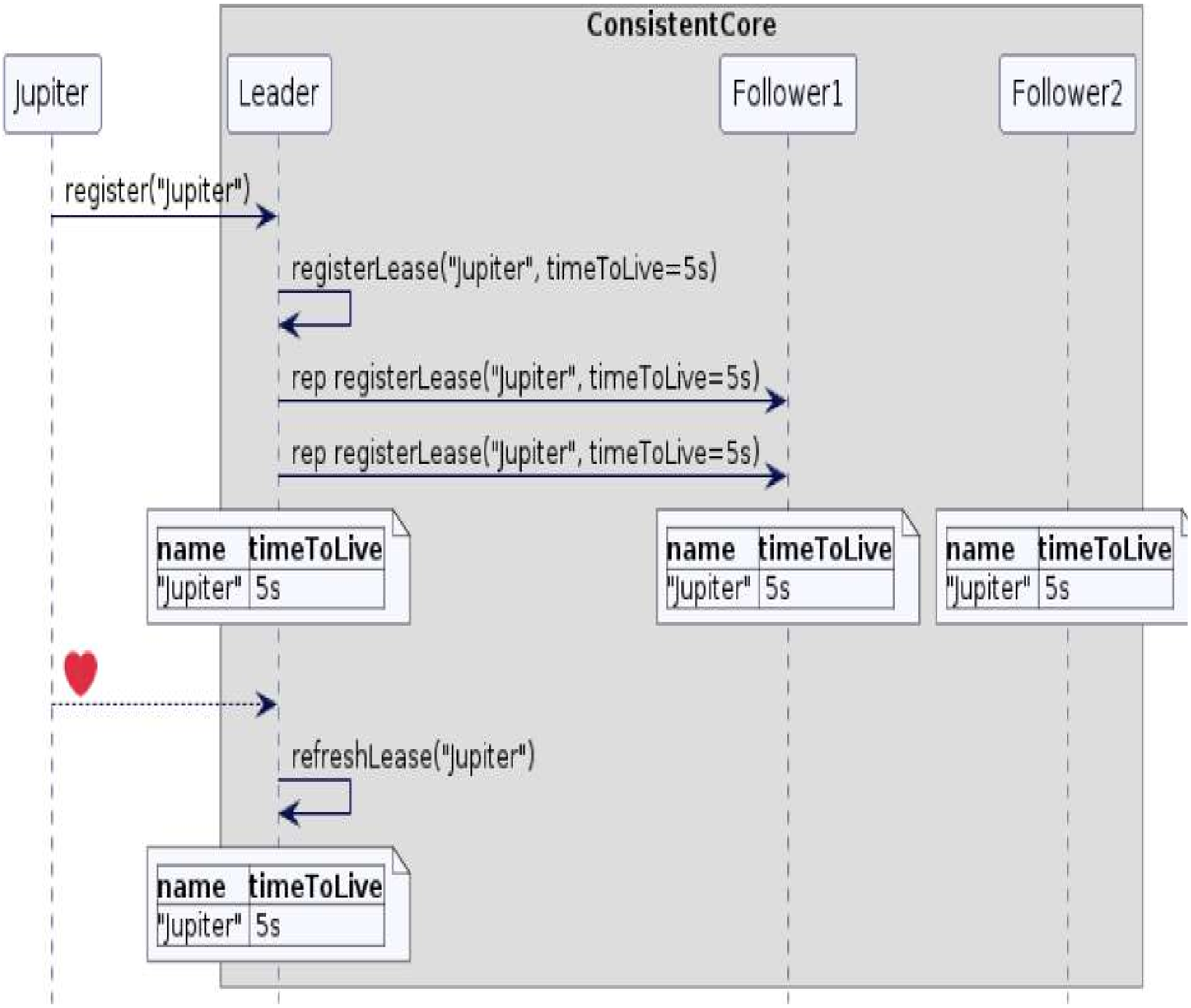
A cluster can have hundreds or thousands of nodes. Such a cluster is also dynamic, allowing more nodes to be added to handle increasing load, or shedding nodes due to failures or reductions in traffic. We see large clusters regularly with Kafka [bib-kafka] or Kubernetes [bib-kubernetes] clusters and with databases like MongoDB [bib-mongodb], Cassandra [bib-cassandra], CockroachDB [bib-cockroachdb] or YugabyteDB [bib-yugabyte]. To manage such a cluster, we need to keep track of which nodes are part of the cluster. With partitioned data we to track the mapping of keys to logical partitions and partitions to nodes. It's important that this management data is handled in a fault tolerant way, so that failure of one control node doesn't bring down the whole cluster. This management data also needs strong consistency, otherwise we risk corruption of our data.

As we've seen earlier, *Replicated Log* is an excellent way to achieve this. But the a Replicated Log depends on *Quorum*, which cannot scale to thousands of replicas. Therefore the management of a large cluster is given to a Consistent Core - a small set of nodes whose responsibility is to manage a larger data cluster. The Consistent Core tracks the data cluster membership with *Lease* and *State Watch*.

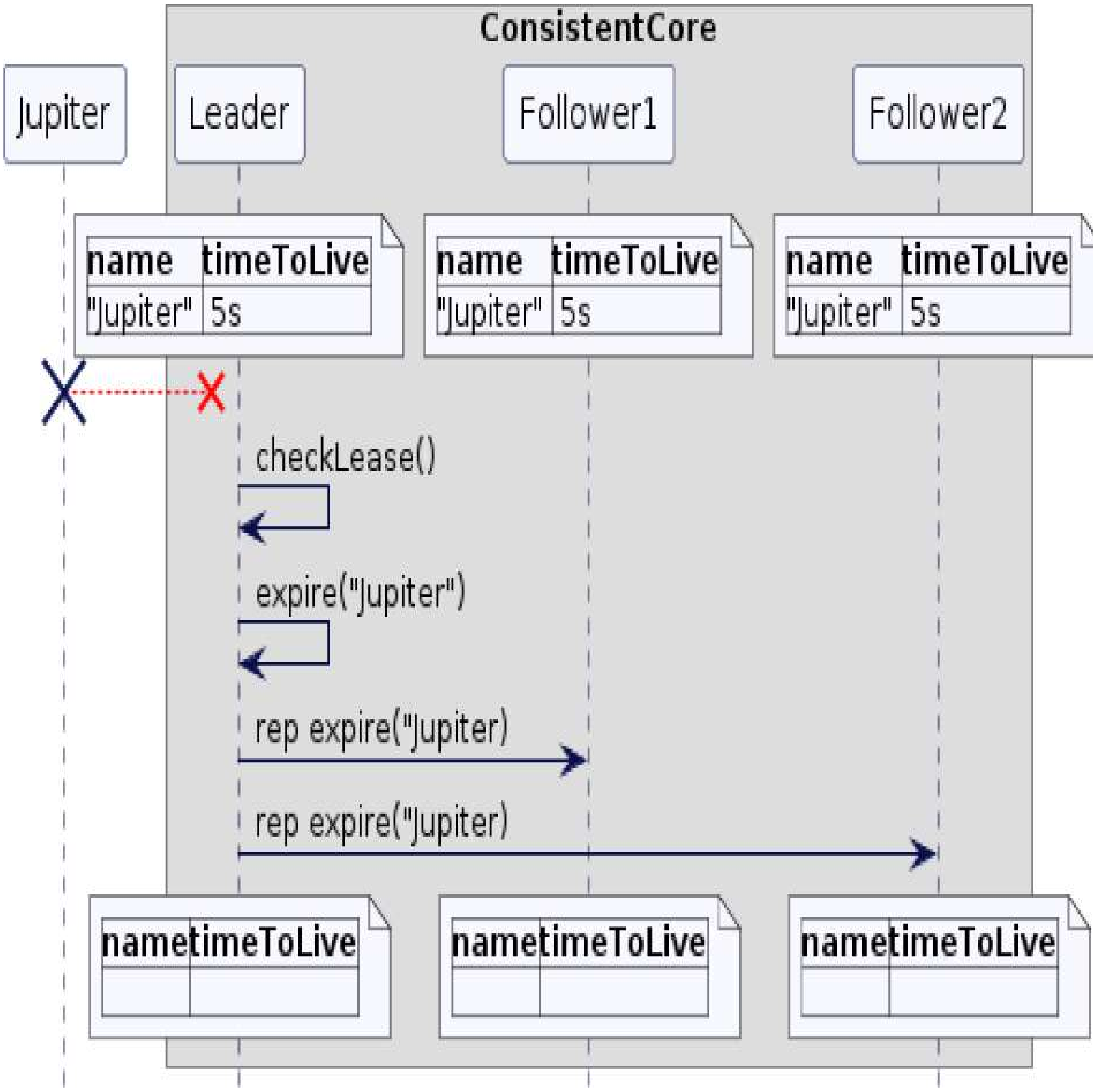


This requirement is so common in distributed services, that some products implement the generic functionality needed to use a Consistent Core like Lease and State Watch. Zookeeper [bib-zookeeper] and etcd [bib-etcd] are good examples of products which are mainly used as consistent-core. But some systems also have their own implementation, e.g. Kafka’s raft [bib-kafka-raft].

An example usage of Lease is for the node registration and failure detection of cluster nodes. The ephemeral node implementation in Zookeeper [bib-zookeeper] or lease functionality in etcd [bib-etcd] is a good example of this. Here Jupiter, a data cluster node registers with the Consistent Core with its unique id or name. The node entries are tracked as leases and renewed with periodic *HeartBeat*.



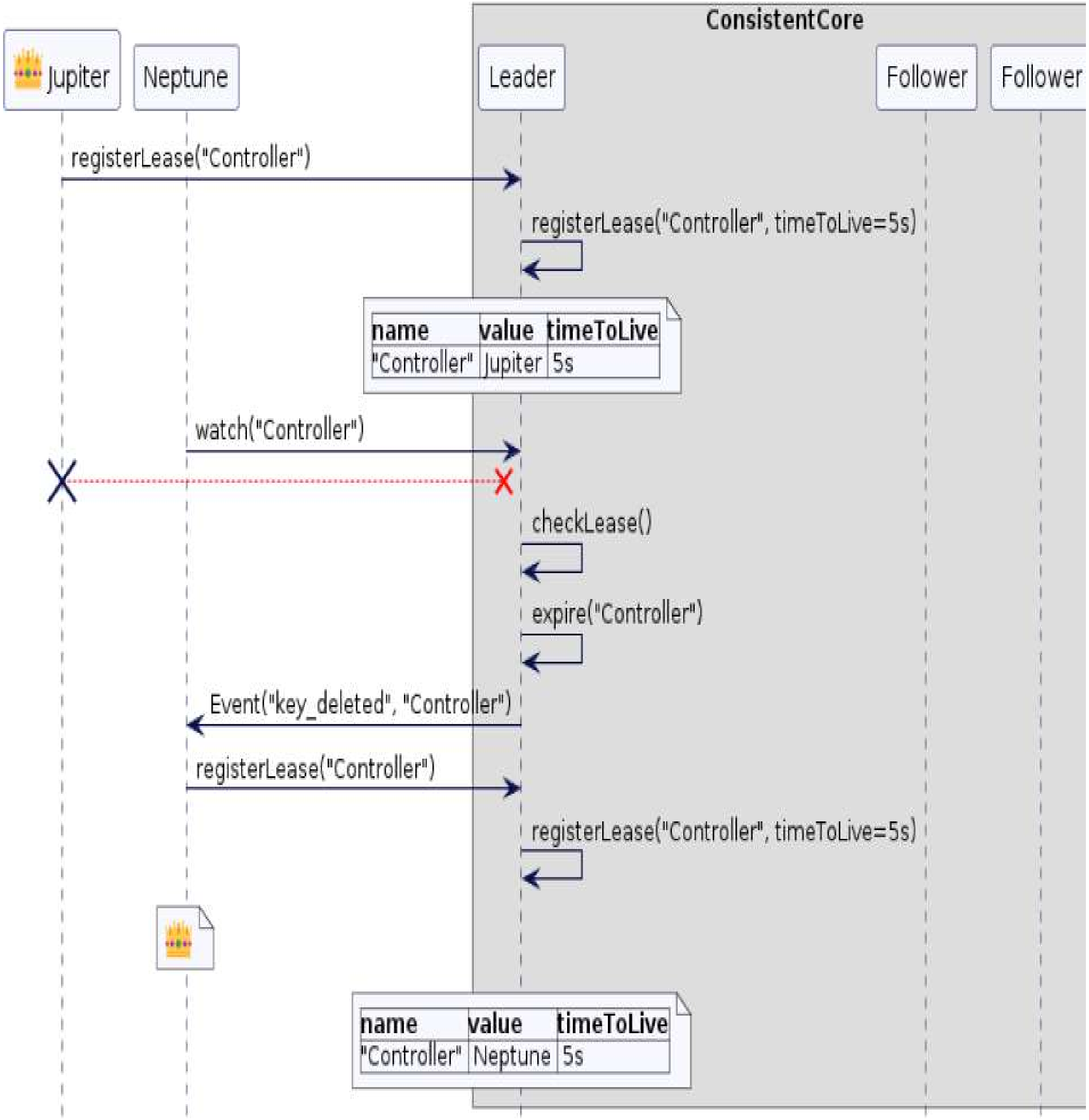
The leader of the Consistent Core periodically checks leases which are not refreshed. If Jupiter crashes and stops sending HeartBeat, the lease will be removed.



Leases are replicated, so if a leader fails, the new leader of the consistent-core will start tracking the leases. Jupiter then needs to connect with the new leader and keep sending the heartbeats.

When a generic Consistent Core like Zookeeper [bib-zookeeper] or etcd [bib-etcd] is used, a dedicated data cluster node uses information stored in etcd or zookeeper to take decisions on behalf of the cluster - called a *cluster controller*. Kafka Controller [bib-kafka-controller] is a good example of this. Other nodes need to know when this particular node is down, so that someone else can take this responsibility. To do this the data cluster node registers a State Watch with the Consistent Core. The Consistent Core notifies all the interested nodes when a particular node fails.

Lets say Jupiter is assuming the role of cluster controller, Neptune wants to know when Jupiter's lease expires. It registers its interest with the Consistent Core by contacting the core's leader. When the lease for Jupiter expires, indicating that Jupiter is probably no longer up and running, the leader of the consistent-core checks to see if any nodes need to be notified. In this case, Neptune is notified with a "lease-deleted" event.


ConsistentCore
Jupiter
Neptune
Leader
Follower
Follower
registerLease("Controller")
registerLease("Controller", timeToLive=5s)
watch("Controller")
checkLease()
expire("Controller")
Event("key_deleted", "Controller")
registerLease("Controller")
registerLease("Controller", timeToLive=5s)

name	value	timeToLive
"Controller"	Jupiter	5s

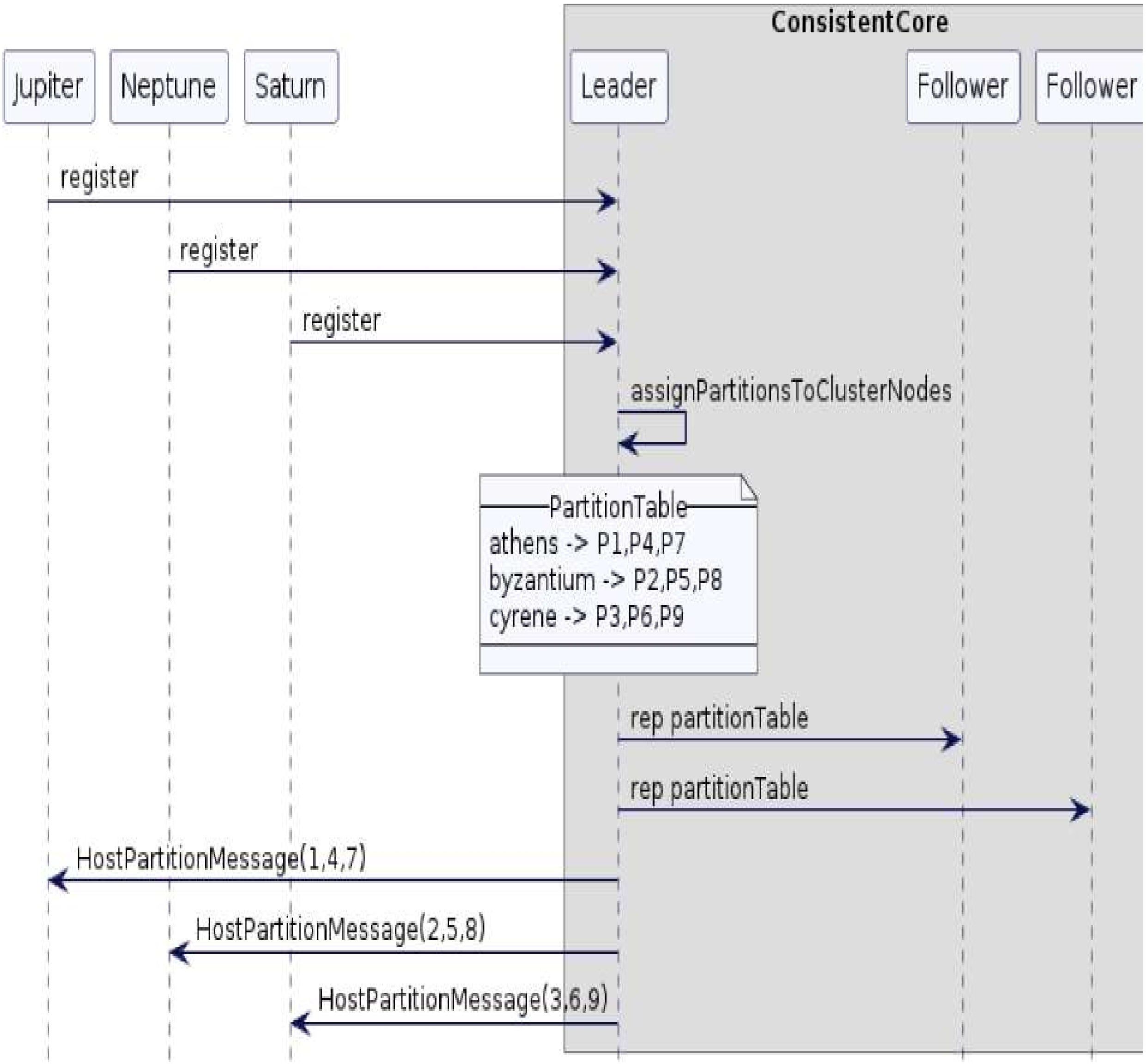
name	value	timeToLive
"Controller"	Neptune	5s

(Replication messages from leader to followers of the consistent-core are not shown in the diagram.)

It’s important that Neptune receives all its events in the correct order. While the communication protocol can sort all that out, it usually more efficient for Neptune to connect to the Consistent Core with a *Single Socket Channel*.

Zookeeper [bib-zookeeper] and etcd [bib-etcd] are examples of generic frameworks which are used by products like Kafka [bib-kafka] or Kubernetes [bib-kubernetes]. A lot of times it is convinient for clustered software to have their own implementation based on Replicated Log and have decision making done in the Consistent Core itself. KIP-631 [bib-kip-631] for Kafka [bib-kafka], the primary cluster of MongoDB [bib-mongodb], and the master cluster of YugabyteDB [bib-yugabyte] are examples of these.

An example of this is when a Consistent Core assigns *Fixed Partitions* to the cluster nodes. Three nodes Jupiter, Neptune and Saturn register with the consistent core. Once the registration is done, the consistent core maps partitions evenly across the cluster nodes.

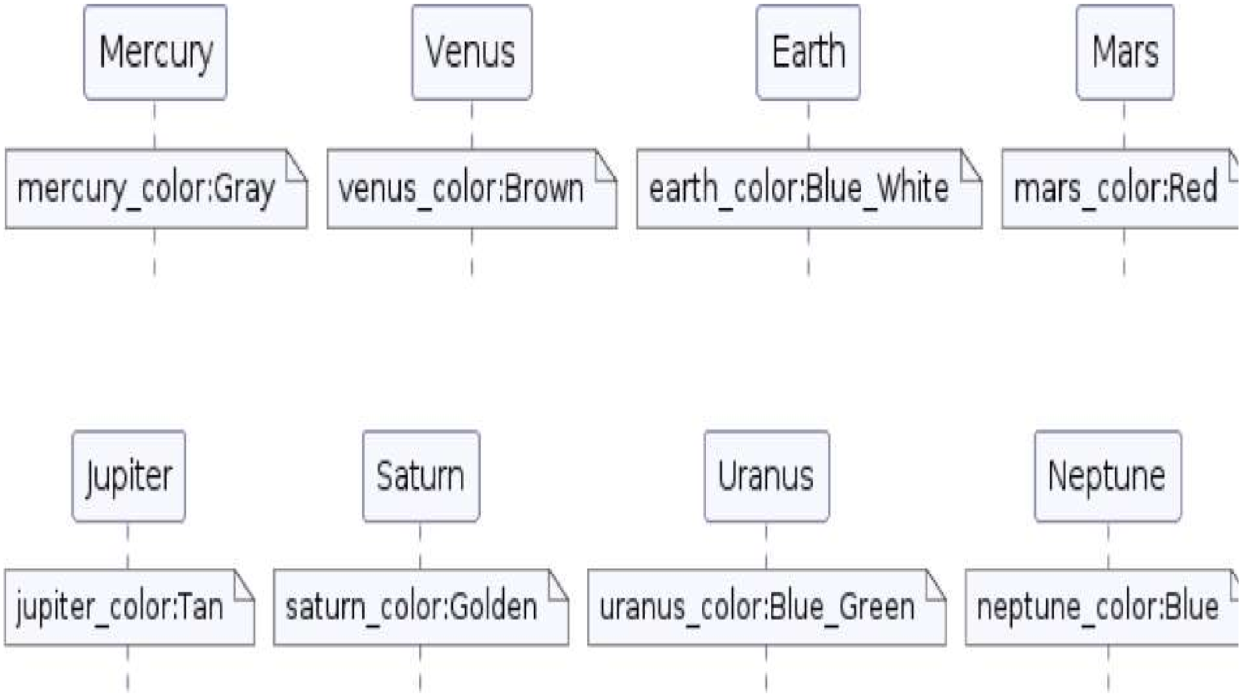


Gossip Dissemination can be used to manage a cluster without a centralized controller

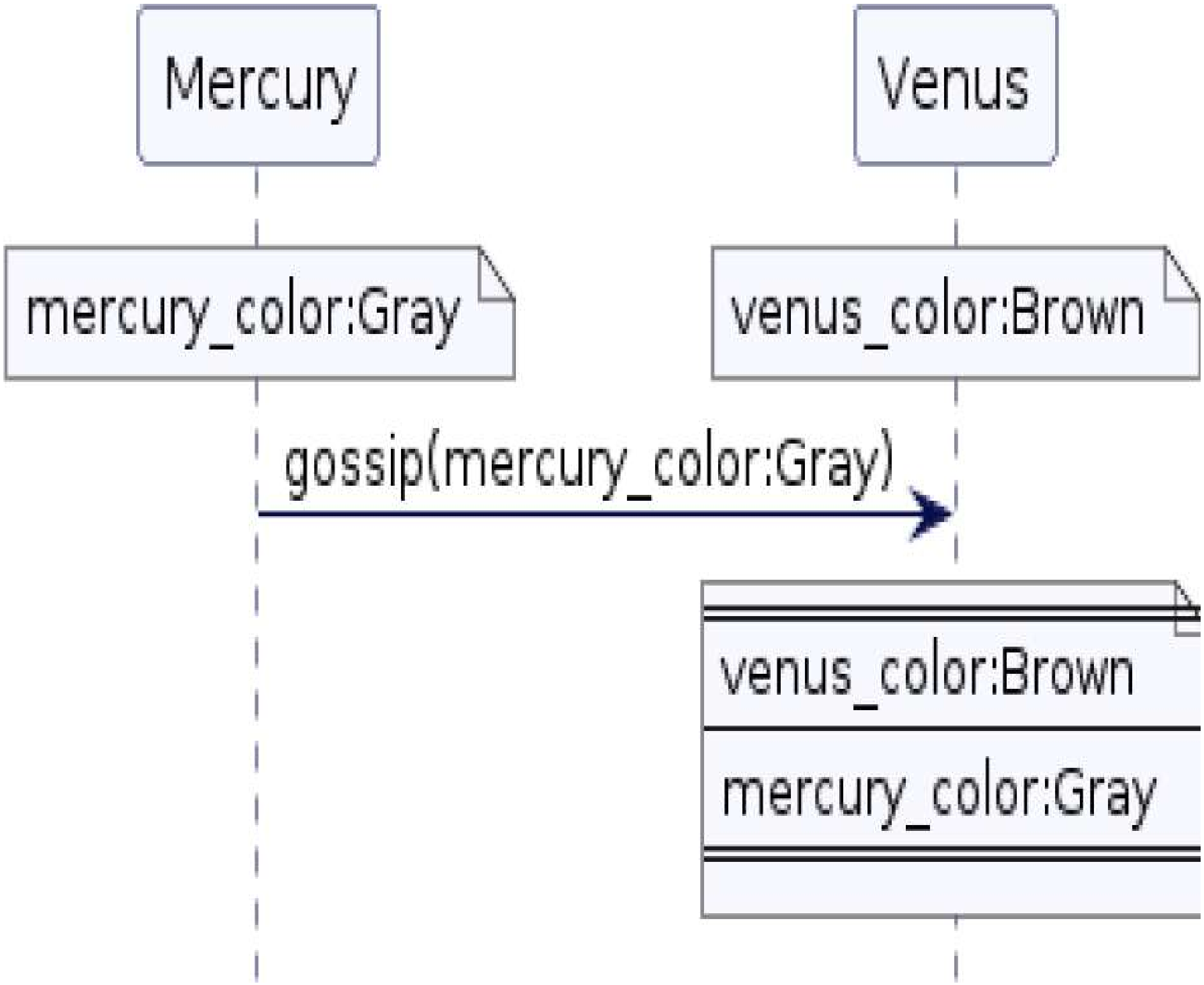
Some systems, like Cassandra [bib-cassandra], lean more towards eventual consistency, and do not want to rely on a centralized *Consistent Core*. They tolerate some inconsisncy in the cluster metadata provided it converges quickly. The need to share metadata like total number of nodes, their network addresses, partitions they host etc, still needs to be propogated somehow to everyone. (As we write this, however, there is a proposal to migrate Cassandra [bib-cassandra] to a Consistent Core.)

We discussed in the last section that there can be thousands of nodes in the cluster. Each node has some information and it needs to make sure it reaches every other node. It will be too much communication overhead if every node talks to every other node. Gossip Dissemination is an interesting way out. At regular intervals, each node picks another node at random and sends it the information it has on the state of the cluster. This style of communication has a nice property. In a cluster with n nodes, this information reaches every node in time proportional to log(n). Interestingly this matches how epidemics spread in large communities. The mathematical branch of epidemiology studies how an epidemic or rumours spread in a society. A disease spreads very quickly even if each person comes into contact with only a few individuals at random. An entire population can become infected from very few interactions. Gossip Dissemination is based on these mathematical models from epidemiology. If nodes send information to few other nodes regularly, even a large cluster will get that information quickly.

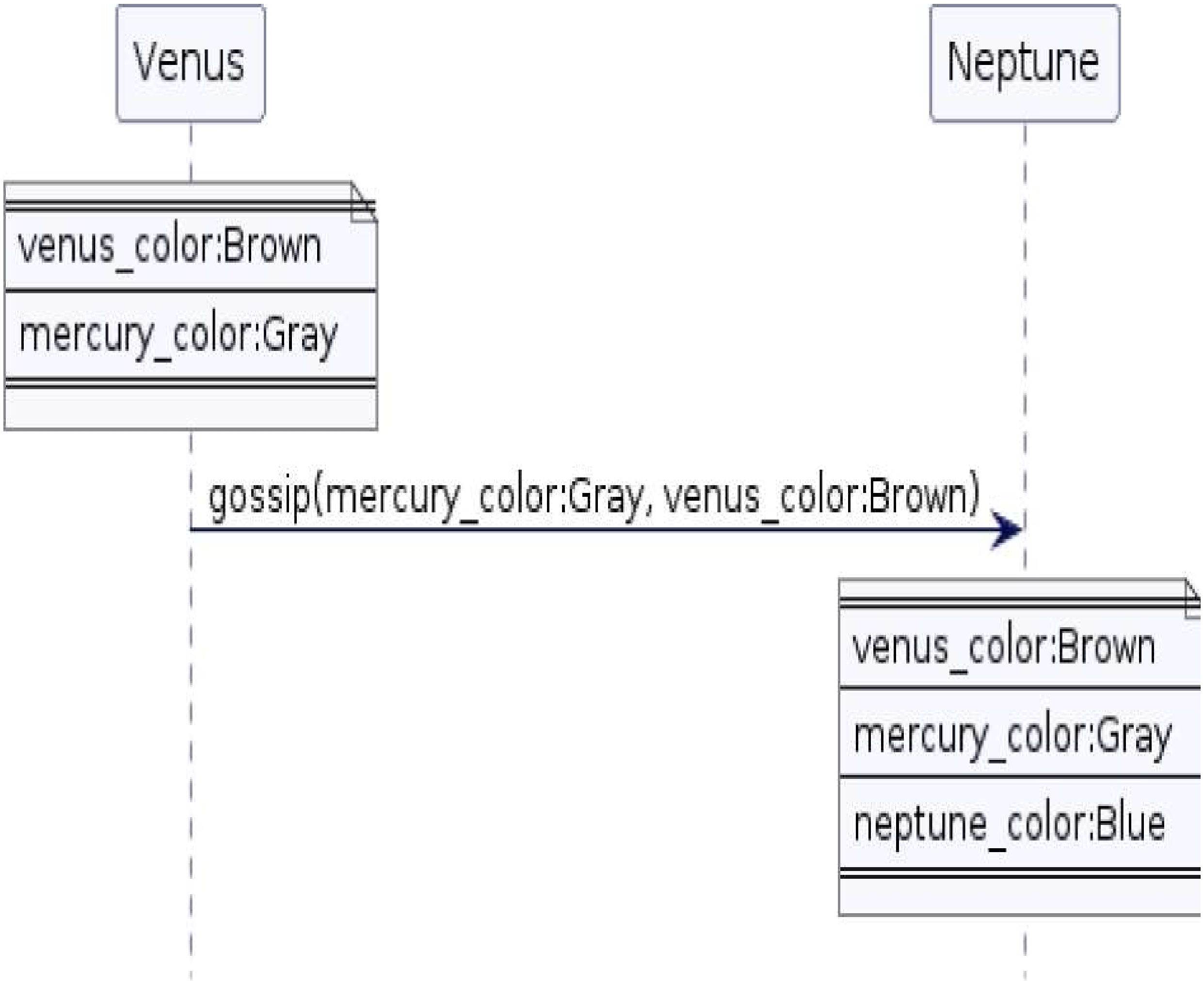
Say we have eight servers, each having information about a color of the planet they are named after. We want all these servers to know about every color.



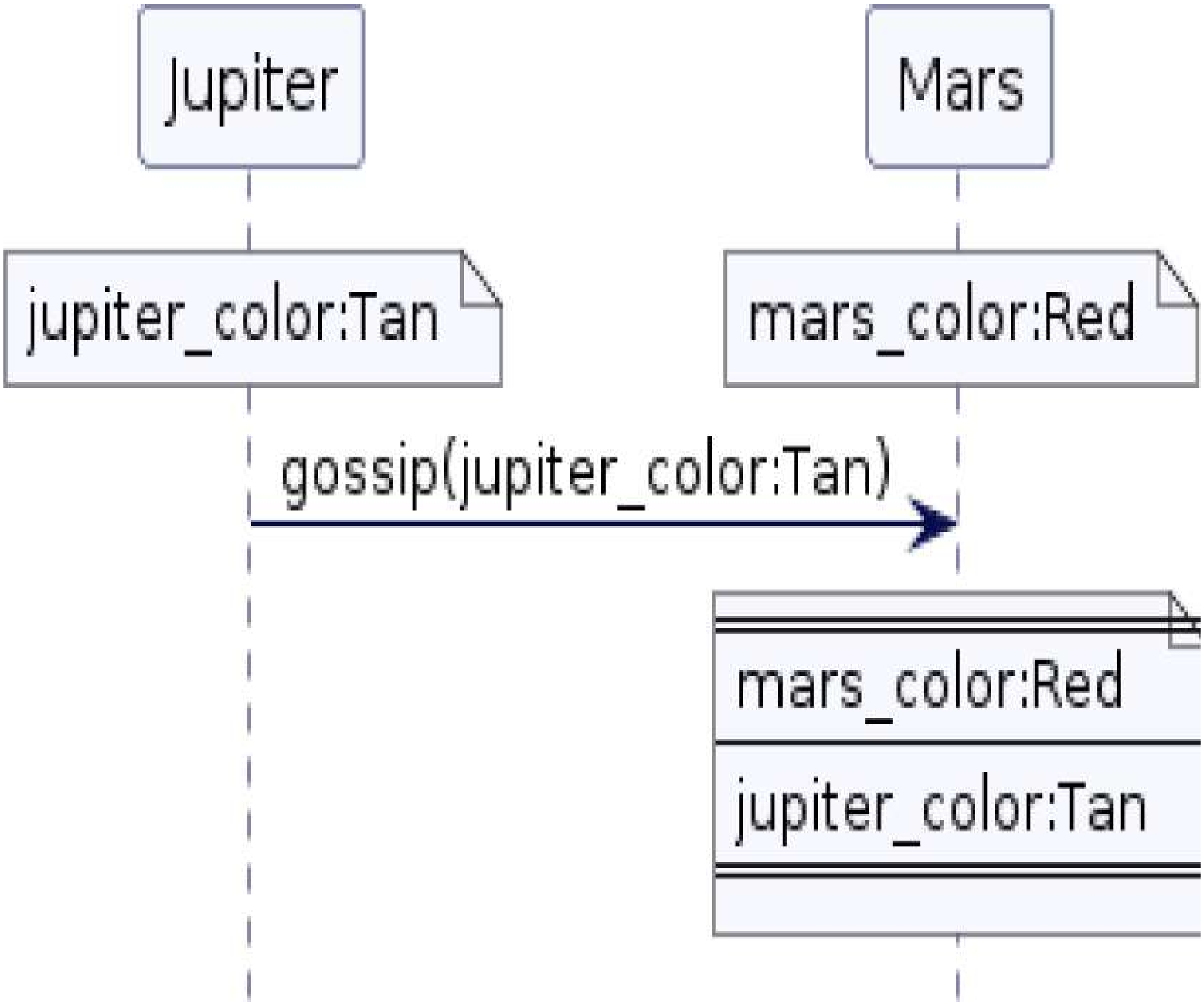
To start with Mercury sends a gossip message to Venus.



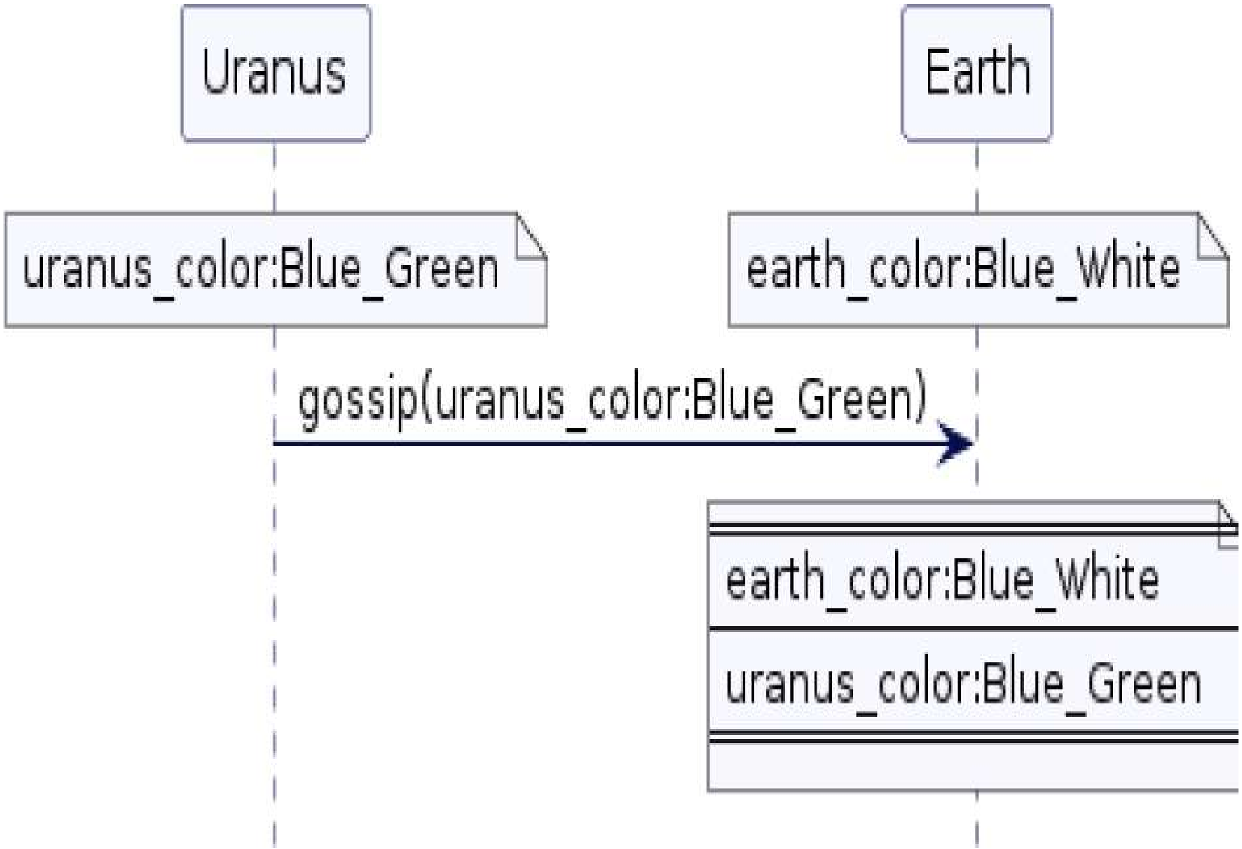
The next time Venus sends gossip message to Neptune. It includes everything it has in the gossip message.



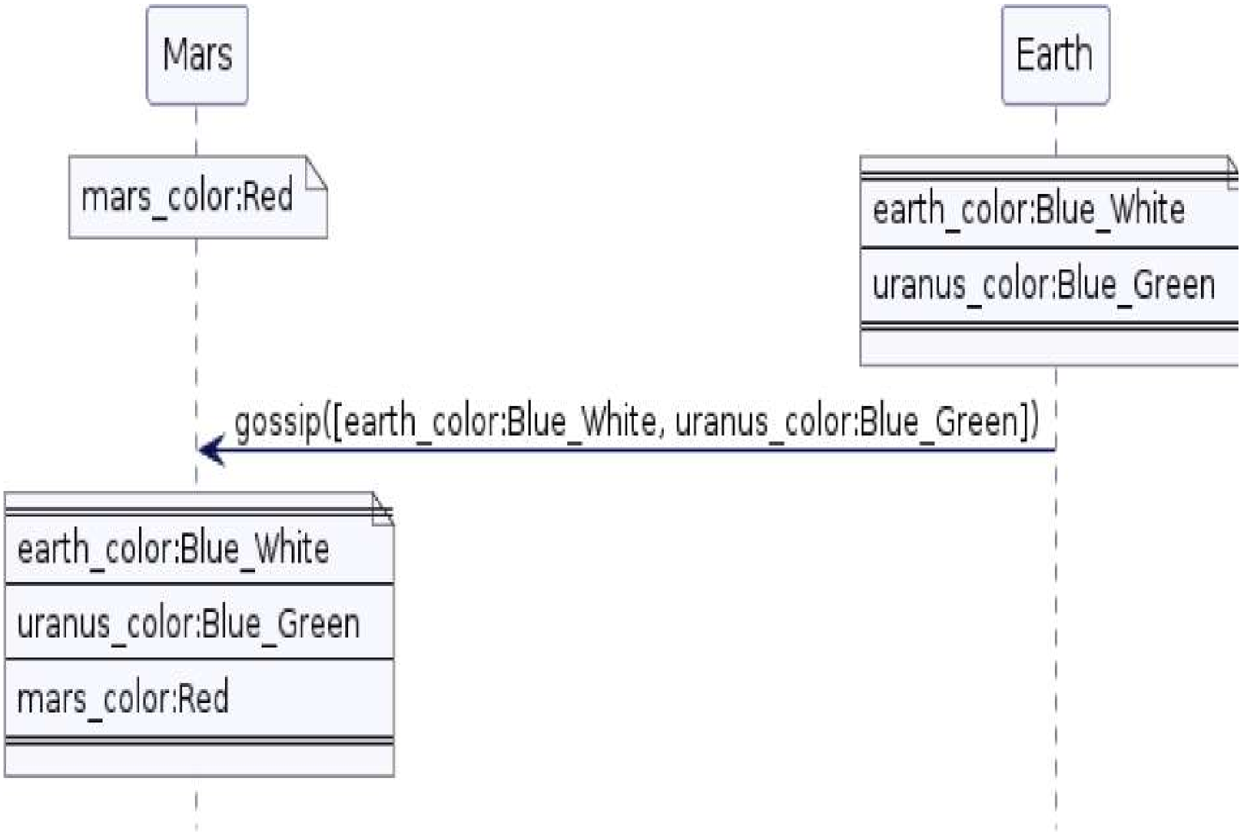
Meanwhile Jupiter has sent gossip message to Mars.



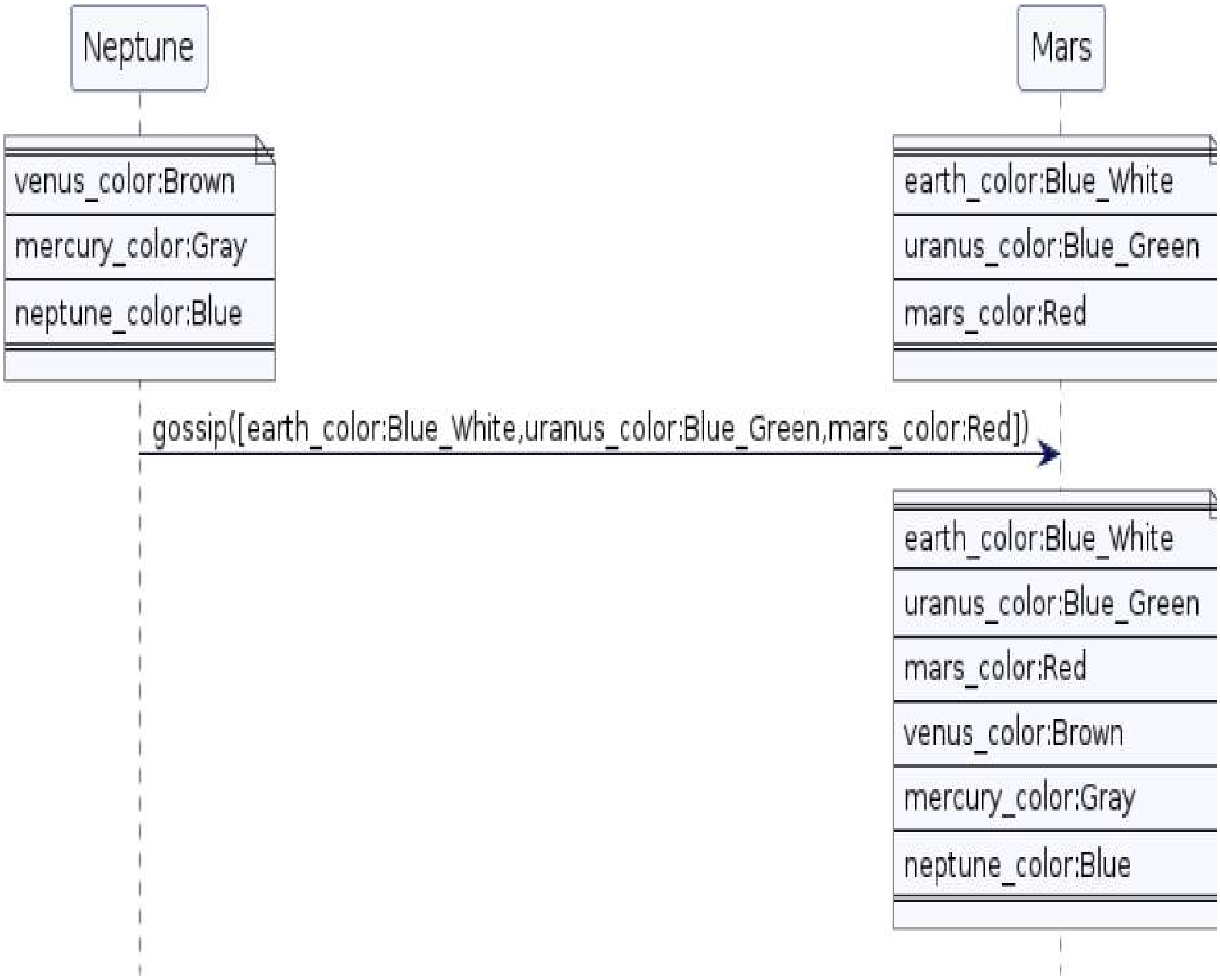
Uranus gossiped to Earth.



Earth gossips to Mars.



When Neptune gossips with Mars, Mars will have colors for Mercury, Venus, Earth, Jupiter and Mars.



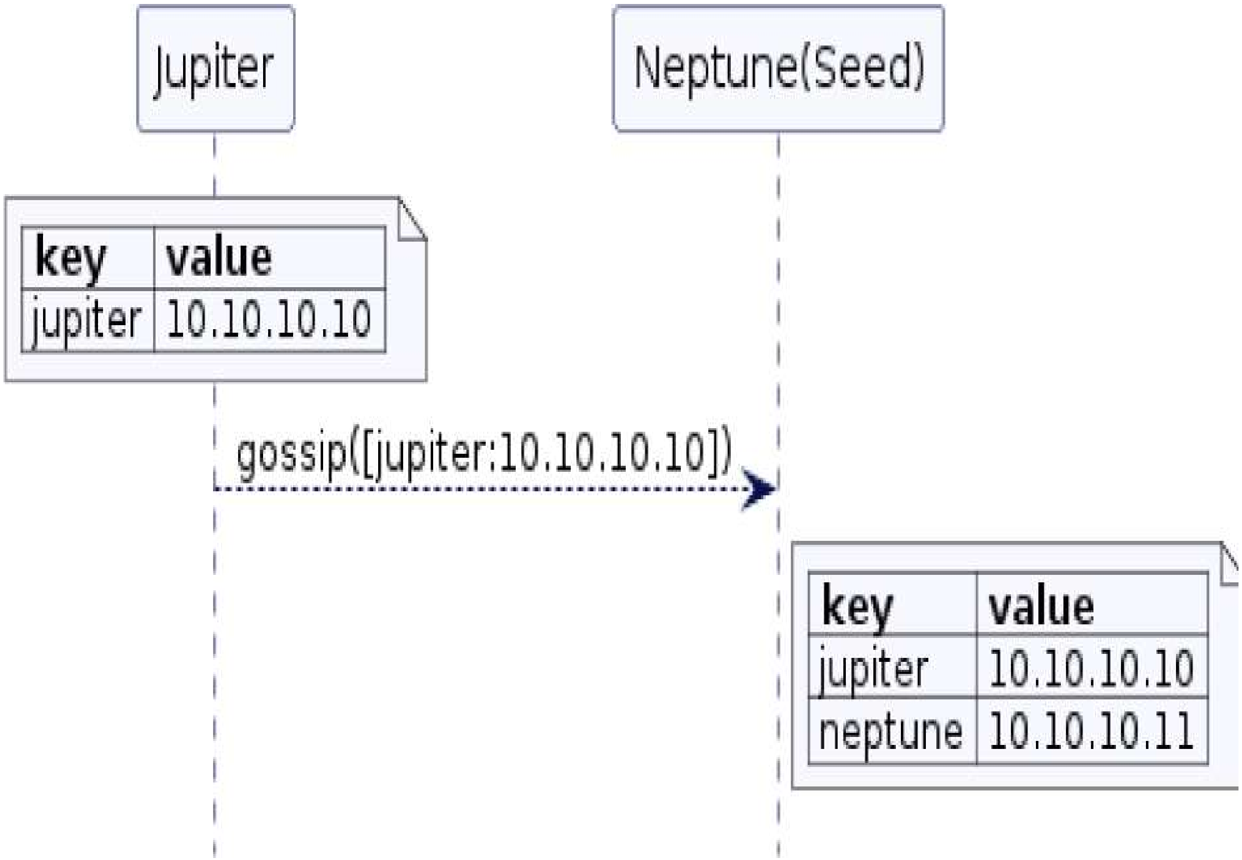
This happens at regular intervals at each node and in a very small amount of time, all nodes eventually get the same information. Consul [bib-consul] has a very nice convergence simulator [bib-serf-convergence-simulator] which shows how quickly the information converges with its gossip implementation.

How is this technique used in real life? One common use is to manage group membership in products like Cassandra [bib-cassandra]. As an example, lets say there is a large cluster, of 100 nodes. We need all the nodes to know about each other. The cluster nodes can achieve this by doing a repeated communication, each time with a random node.

To start with, there is at least one special node, which needs to be known to every one. This well-known node is called the ‘seed’ node. It can either be configured or there can be mechanism to know about the seed node at the startup. The ‘seed’ node is not any special node, but just one of the nodes in the cluster. It does not implement any special functionality. It only needs to be known to every one else.

Every node starts by first communicating with the seed node and send its own address to seed node. Seed node, as like any other node has a scheduled task to repeatedly send what it knows about to a randomly selected node.

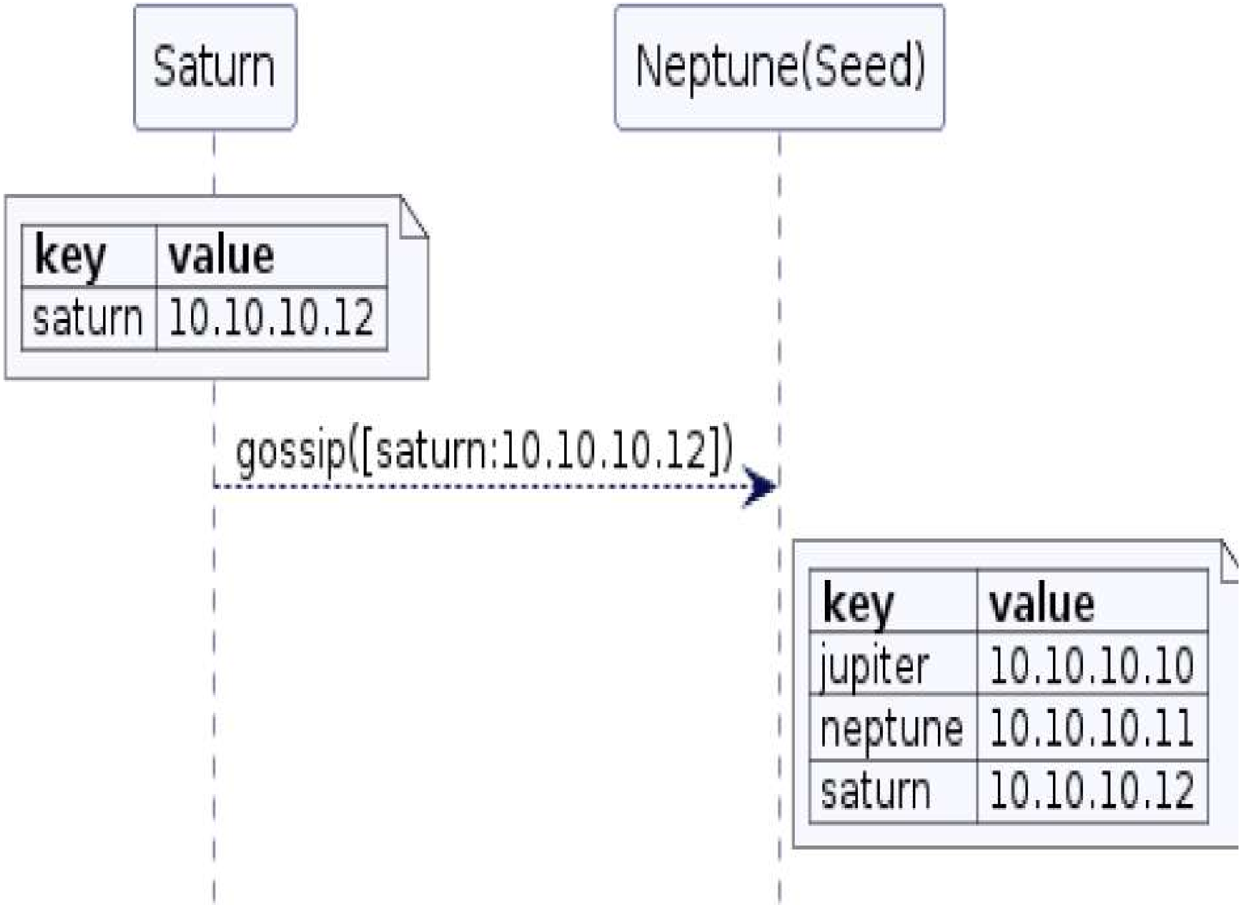
Consider as an example, that Neptune is the seed node. When Jupiter starts, it sends its own address to Neptune.



key	value
jupiter	10.10.10.10

key	value
jupiter	10.10.10.10
neptune	10.10.10.11

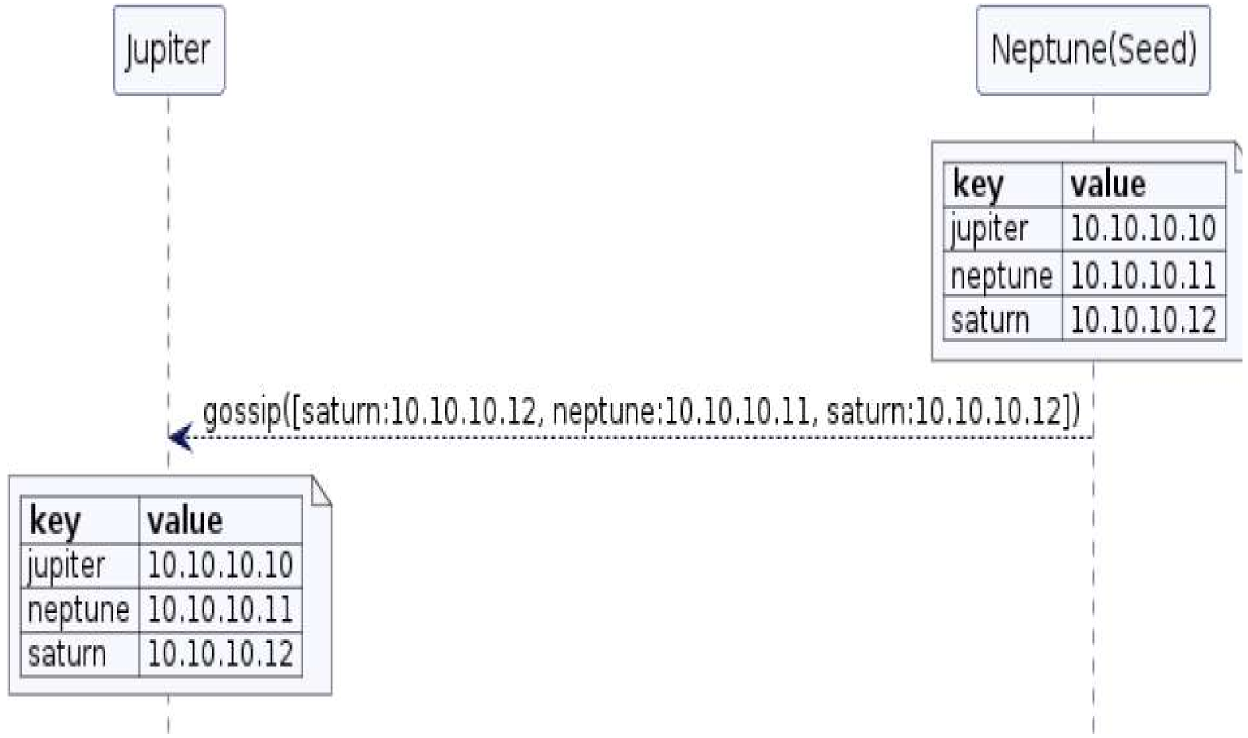
Then Saturn starts, it sends its own address to Neptune the same way.



key	value
saturn	10.10.10.12

key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

All of Jupiter, Saturn and Neptune repeatedly pick a random node and send it all the information that they have. So in a while, Neptune picks up Jupiter and sends it information that it has. Now both, Neptune and Jupiter know about Saturn.

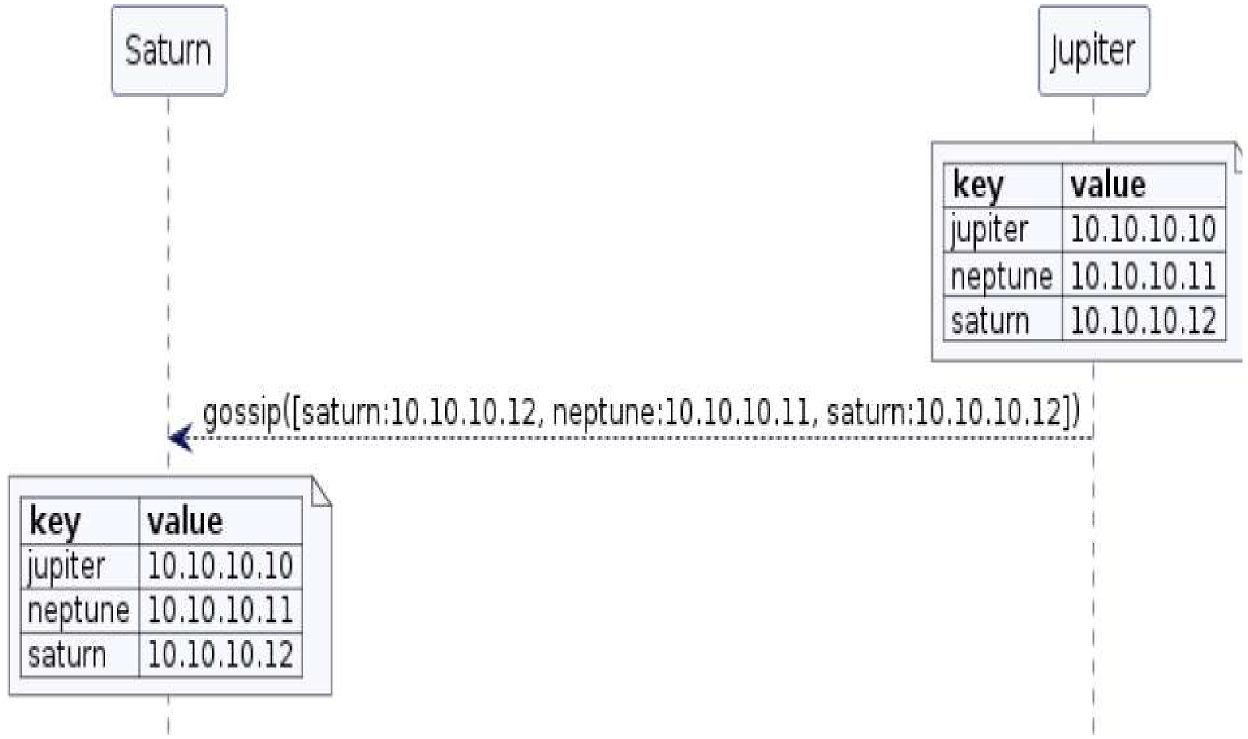


key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

gossip([saturn:10.10.10.12, neptune:10.10.10.11, saturn:10.10.10.12])

key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

In the next cycle, either Jupiter or Neptune might pick up Saturn and send it all the information that they have. Now, Saturn will know about Jupiter. At this point, all three nodes know about each other.

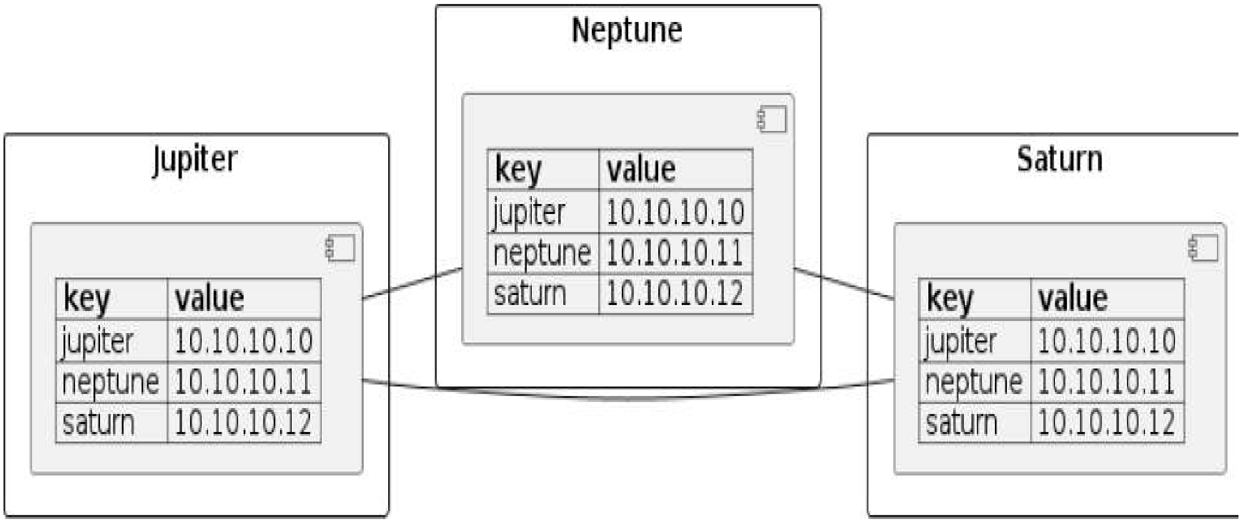


key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

gossip([saturn:10.10.10.12, neptune:10.10.10.11, saturn:10.10.10.12])

key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

At this point, all of Saturn, Neptune and Jupiter have exactly the same information.



key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

key	value
jupiter	10.10.10.10
neptune	10.10.10.11
saturn	10.10.10.12

Gossip Dissemination is a commonly used technique for information dissemination in large clusters. Products like Cassandra [bib-cassandra], Akka [bib-akka] and Consul [bib-consul] use it for managing group membership information in large clusters.

Some systems like Akka [bib-akka] designate a single cluster node to act as a cluster coordinator without running an explicit leader election. Similar to Consistent Core, an *Emergent Leader* takes decisions on behalf of the cluster. A common technique is for the nodes to be ordered based on their age in the cluster. The node with the highest age is designated as a leader of the cluster to take decisions. Because no explicit election is run, there can be inconsistencies caused by problems like Split Brain [emergentleader.xhtml#SplitBrainSituation].

Part II: Patterns of Data Replication

Chapter 3. Write-Ahead Log

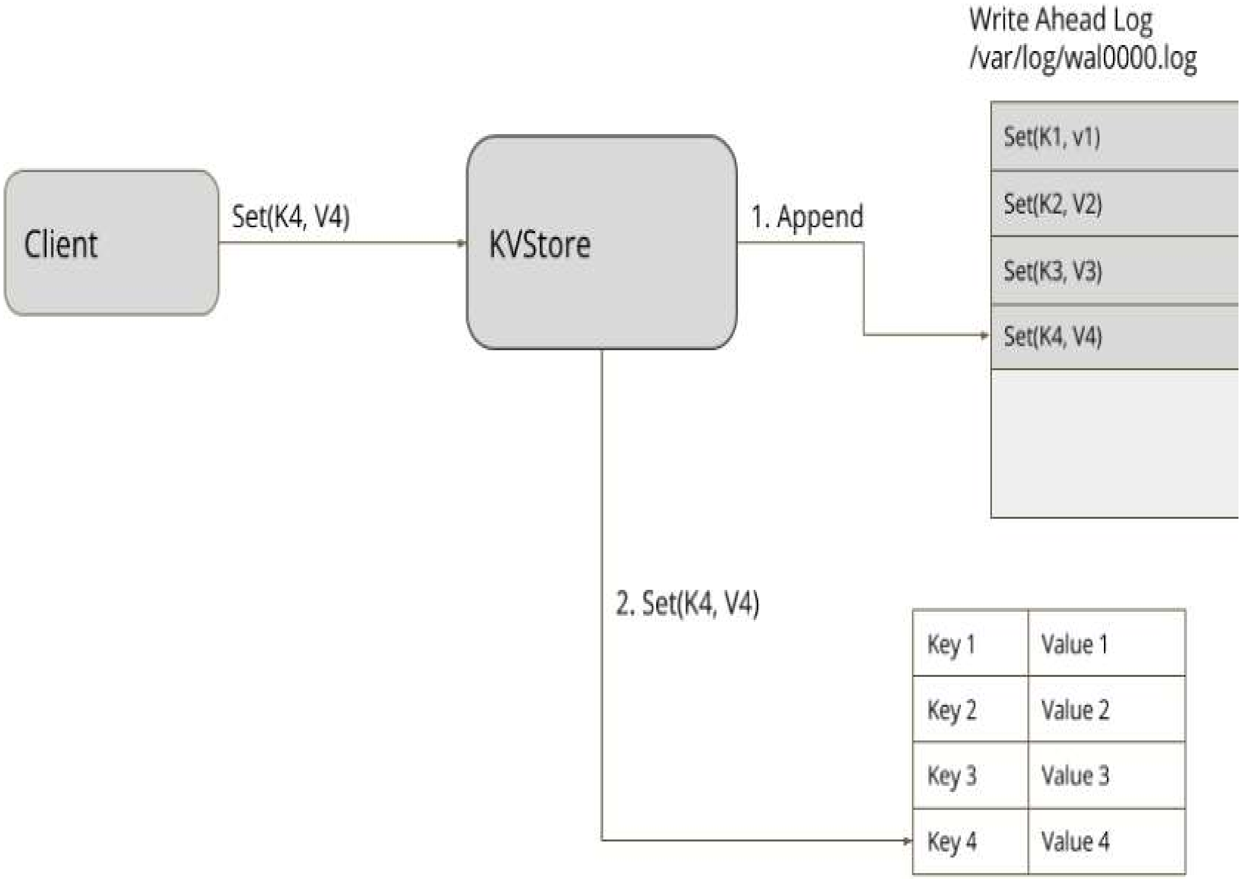
Provide durability guarantee without the storage data structures to be flushed to disk, by persisting every state change as a command to the append only log.

Also known as: Commit Log

Problem

Strong durability guarantee is needed even in the case of the server machines storing data failing. Once a server agrees to perform an action, it should do so even if it fails and restarts losing all of its in-memory state.

Solution



Key 1	Value 1
Key 2	Value 2
Key 3	Value 3
Key 4	Value 4

© 2019 ThoughtWorks

Store each state change as a command in a file on a hard disk. A single log is maintained for each server process which is sequentially appended. A single log which is appended sequentially, simplifies the handling of logs at restart and for subsequent online operations (when the log is appended with new commands). Each log entry is given a unique identifier. The unique log identifier helps in implementing certain other operations on the log like *Segmented Log* or cleaning the log with *Low-Water Mark* etc. The log updates can be implemented with *Singular Update Queue*

The typical log entry structure looks like following

class WALEntry…

```
  private final Long entryIndex;
  private final byte[] data;
  private final EntryType entryType;
  private final long timeStamp;
```

The file can be read on every restart and the state can be recovered by replaying all the log entries.

Consider a simple in memory key-value store:

class KVStore…

```
  private Map<String, String> kv = new HashMap<>();

  public String get(String key) {
    return kv.get(key);

  }

  public void put(String key, String value) {
    appendLog(key, value);
    kv.put(key, value);
  }

  private Long appendLog(String key, String value) {
     return wal.writeEntry(new SetValueCommand(key, value).serializ
  }
```

The put action is represented as *Command*, which is serialized and stored in the log before updating the in memory hashmap.

class SetValueCommand…

```
  final String key;
  final String value;
  final String attachLease;
  public SetValueCommand(String key, String value) {
    this(key, value, "");
```

```
}
public SetValueCommand(String key, String value, String attachLea
  this.key = key;
  this.value = value;
  this.attachLease = attachLease;
}

@Override
public void serialize(DataOutputStream os) throws IOException {
  os.writeInt(Command.SetValueType);
  os.writeUTF(key);
  os.writeUTF(value);
  os.writeUTF(attachLease);
}

public static SetValueCommand deserialize(InputStream is) {
  try {
     DataInputStream dataInputStream = new DataInputStream(is);
     return new SetValueCommand(dataInputStream.readUTF(), dataIn
  } catch (IOException e) {
     throw new RuntimeException(e);
  }
}
```

This makes sure that once the put method returns successfully, even if the process holding the KVStore crashes, its state can be restored by reading the log file at startup.

class KVStore…

```
public KVStore(Config config) {
  this.config = config;
  this.wal = WriteAheadLog.openWAL(config);
  this.applyLog();
}

public void applyLog() {
  List<WALEntry> walEntries = wal.readAll();
  applyEntries(walEntries);
}

private void applyEntries(List<WALEntry> walEntries) {
   for (WALEntry walEntry : walEntries) {
      Command command = deserialize(walEntry);
      if (command instanceof SetValueCommand) {
         SetValueCommand setValueCommand = (SetValueCommand)comm
         kv.put(setValueCommand.key, setValueCommand.value);
     }
   }
 }
 public void initialiseFromSnapshot(SnapShot snapShot) {
   kv.putAll(snapShot.deserializeState());
 }
```

Implementation Considerations

It’s important to make sure that entries written to the log file are actually persisted on the physical media. File handling libraries provided in all programming languages provide a mechanism to force the operating system to ‘flush’ the file changes to physical media. There is a trade off to be considered while using flushing mechanism.

Flushing every log write to the disk gives a strong durability guarantee (which is the main purpose of having logs in the first place), but this severely limits performance and can quickly become a bottleneck. If flushing is delayed or done asynchronously, it improves performance but there is a risk of losing entries from the log if the server crashes before entries are flushed. Most implementations use techniques like Batching, to limit the impact of the flush operation.

The other consideration is to make sure that corrupted log files are detected while reading the log. To handle this, log entries are generally written with the CRC records, which then can be validated when the files are read.

Single Log files can become difficult to manage and can quickly consume all the storage. To handle this issue, techniques like *Segmented Log* and *Low-*

Water Mark are used.

The write ahead log is append-only. Because of this behaviour, in case of client communication failure and retries, logs can contain duplicate entries. When the log entries are applied, it needs to make sure that the duplicates are ignored. If the final state is something like a HashMap, where the updates to the same key are idempotent, no special mechanism is needed. If they're not, there needs to be some mechanism implemented to mark each request with a unique identifier and detect duplicates.

Compared to Event Sourcing

The use of a log of changes is similar to the log of events in Event Sourcing [bib-event-sourcing]. Indeed when an event-sourced system uses its log to synchronize multiple systems, it is using its log as a write-ahead log. However an event-sourced system uses its log for more than just that, for instance the ability to reconstruct a state at previous points in history. For this an event sourcing log is the persistent source of truth and log entries are kept for a long time, often indefinitely.

The entries for a write-ahead log, however, are only needed for the state recovery. Thus they can be discarded when all the nodes have acknowledged an update, i.e. below the *Low-Water Mark*.

Examples

- The log implementation in all Consensus algorithms like Zookeeper [bib-zookeeper-wal] and RAFT [bib-etcd-wal] is similar to write ahead log
- The storage implementation in Kafka [bib-kafka-log] follows similar structure as that of commit logs in databases
- All the databases, including the nosql databases like Cassandra use write ahead log technique [bib-cassandra-wal] to guarantee durability

Chapter 4. Segmented Log

Split log into multiple smaller files instead of a single large file for easier operations.

Problem

A single log file can grow and become a performance bottleneck while its read at the startup. Older logs are cleaned up periodically and doing cleanup operations on a single huge file is difficult to implement

Solution

Single log is split into multiple segments. Log files are rolled after a specified size limit.

```
public synchronized Long writeEntry(WALEntry entry) {
  maybeRoll();
  return openSegment.writeEntry(entry);
}

private void maybeRoll() {
   if (openSegment.
          size() >= config.getMaxLogSize()) {
       openSegment.flush();
       sortedSavedSegments.add(openSegment);
       long lastId = openSegment.getLastLogEntryIndex();
       openSegment = WALSegment.open(lastId, config.getWalDir());
   }
}
```

With log segmentation, there needs to be an easy way to map logical log offsets (or log sequence numbers) to the log segment files. This can be done in two ways:

- Each log segment name is generated by some well known prefix and the base offset (or log sequence number).
- Each log sequence number is divided into two parts, the name of the file and the transaction offset.

```
public static String createFileName(Long startIndex) {
  return logPrefix + "_" + startIndex + logSuffix;
}

public static Long getBaseOffsetFromFileName(String fileName) {
  String[] nameAndSuffix = fileName.split(logSuffix);
  String[] prefixAndOffset = nameAndSuffix[0].split("_");
  if (prefixAndOffset[0].equals(logPrefix))
     return Long.parseLong(prefixAndOffset[1]);
     return -1l;
}
```

With this information, the read operation is two steps. For a given offset (or transaction id), the log segment is identified and all the log records are read from subsequent log segments.

```
public synchronized List<WALEntry> readFrom(Long startIndex) {
  List<WALSegment> segments = getAllSegmentsContainingLogGreaterTh
  return readWalEntriesFrom(startIndex, segments);
}

private List<WALSegment> getAllSegmentsContainingLogGreaterThan(Lo
  List<WALSegment> segments = new ArrayList<>();
  //Start from the last segment to the first segment with starting
  //This will get all the segments which have log entries more tha
  for (int i = sortedSavedSegments.size() - 1; i >= 0; i--) {
     WALSegment walSegment = sortedSavedSegments.get(i);
     segments.add(walSegment);

     if (walSegment.getBaseOffset() <= startIndex) {
        break; // break for the first segment with baseoffset less
     }
   }

   if (openSegment.getBaseOffset() <= startIndex) {
      segments.add(openSegment);
   }

   return segments;
 }
```

Examples

- The log implementation in all Consensus implementations like Zookeeper [https://github.com/apache/zookeeper/blob/master/zookeeper-server/src/main/java/org/apache/zookeeper/server/persistence/FileTxnLog.java] and RAFT [https://github.com/etcd-io/etcd/blob/master/wal/wal.go] uses log segmentation.
- The storage implementation in Kafka [https://github.com/axbaretto/kafka/blob/master/core/src/main/scala/kafka/log/Log.scala] follows log segmentation.
- All the databases, including the nosql databases like Cassandra [https://github.com/facebookarchive/cassandra/blob/master/src/org/apache/cassandra/db/CommitLog.java] use roll over strategy based on some pre configured log size.

Chapter 5. Low-Water Mark

An index in the write ahead log showing which portion of the log can be discarded.

Problem

The write ahead log maintains every update to persistent store. It can grow indefinitely over time. *Segmented Log* allows dealing with smaller files at a time, but total disk storage can grow indefinitely if not checked.

Solution

Have a mechanism to tell logging machinery which portion of the log can be safely discarded. The mechanism gives the lowest offset or low water mark, before which point the logs can be discarded. Have a task running in the background, in a separate thread, which continuously checks which portion of the log can be discarded and deletes the files on the disk.

```
this.logCleaner = newLogCleaner(config);
this.logCleaner.startup();
```

The Log cleaner can be implemented as a scheduled task

```
public void startup() {
  scheduleLogCleaning();
}

private void scheduleLogCleaning() {
   singleThreadedExecutor.schedule(() -> {
      cleanLogs();
   }, config.getCleanTaskIntervalMs(), TimeUnit.MILLISECONDS);
}
```

Snapshot based Low-Water Mark

Most consensus implementations like Zookeeper, or etcd (as defined in RAFT), implement snapshot mechanisms. In this implementation, the storage engine takes periodic snapshots. Along with snapshot, it also stores the log index which is successfully applied. Referring to the simple key value store implementation in the *Write-Ahead Log* pattern, the snapshot can be taken as following:

```
public SnapShot takeSnapshot() {
  Long snapShotTakenAtLogIndex = wal.getLastLogIndex();
  return new SnapShot(serializeState(kv), snapShotTakenAtLogIndex)
}
```

Once a snapshot is successfully persisted on the disk, the log manager is given the low water mark to discard the older logs.

```
List<WALSegment> getSegmentsBefore(Long snapshotIndex) {
  List<WALSegment> markedForDeletion = new ArrayList<>();
  List<WALSegment> sortedSavedSegments = wal.sortedSavedSegments;
  for (WALSegment sortedSavedSegment : sortedSavedSegments) {
    if (sortedSavedSegment.getLastLogEntryIndex() < snapshotIndex)
       markedForDeletion.add(sortedSavedSegment);
    }
  }
  return markedForDeletion;
}
```

Time based Low-Water Mark

In some systems, where log is not necessarily used to update the state of the system, log can be discarded after a given time window, without waiting for any other subsystem to share the lowest log index which can be removed. For example, in systems like Kafka, logs are maintained for 7 weeks; all the log segments which have messages older than 7 weeks are discarded. For this implementation, each log entry also includes the timestamp when it was created. The log cleaner can then check the last entry of each log segment, and discard segments which are older than the configured time window.

```
private List<WALSegment> getSegmentsPast(Long logMaxDurationMs) {
  long now = System.currentTimeMillis();
  List<WALSegment> markedForDeletion = new ArrayList<>();
  List<WALSegment> sortedSavedSegments = wal.sortedSavedSegments;
  for (WALSegment sortedSavedSegment : sortedSavedSegments) {
    if (timeElaspedSince(now, sortedSavedSegment.getLastLogEntryTi
       markedForDeletion.add(sortedSavedSegment);
    }
  }
  return markedForDeletion;
}

private long timeElaspedSince(long now, long lastLogEntryTimestamp
   return now - lastLogEntryTimestamp;
}
```

Examples

- The log implementation in all Consensus algorithms like Zookeeper [https://github.com/apache/zookeeper/blob/master/zookeeper-server/src/main/java/org/apache/zookeeper/server/persistence/FileTxnLog.java] and RAFT [https://github.com/etcd-io/etcd/blob/master/wal/wal.go] implement snapshot based log cleaning

- The storage implementation in Kafka [https://github.com/axbaretto/kafka/blob/master/core/src/main/scala/kafka/log/Log.scala] follows time based log cleaning

Chapter 6. Leader and Followers

Have a single server to coordinate replication across a set of servers.

Problem

To achieve fault tolerance in systems which manage data, the data needs to be replicated on multiple servers.

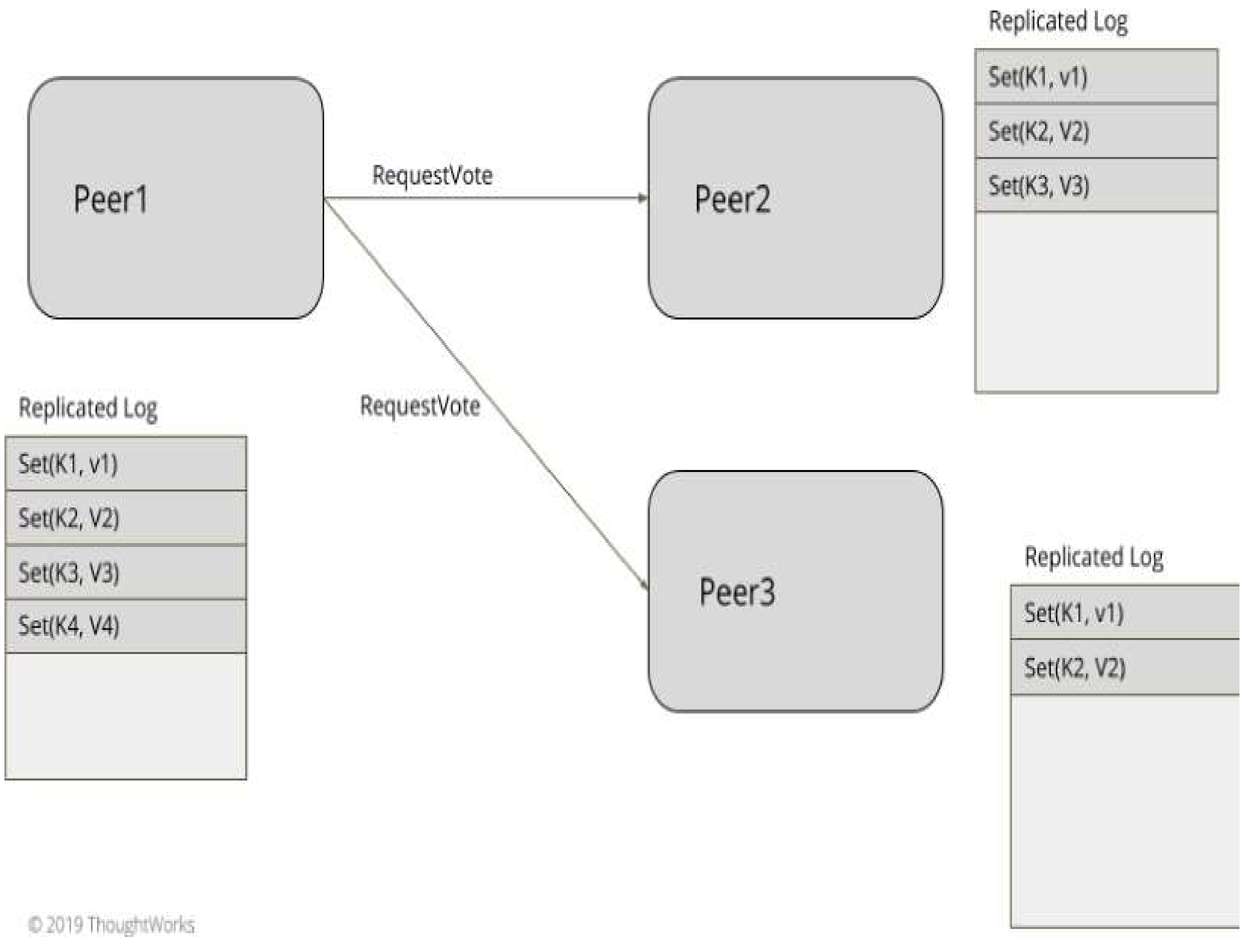
It’s also important to give some guarantee about consistency to clients. When data is updated on multiple servers, a decision about when to make it visible to clients is required. Write and read *Quorum* is not sufficient, as some failure scenarios can cause clients to see data inconsistently. Each individual server does not know about the state of data on the other servers in the quorum, It’s only when data is read from multiple servers, the inconsistencies can be resolved. In some cases, this is not enough. Stronger guarantees are needed about the data that is sent to the clients.

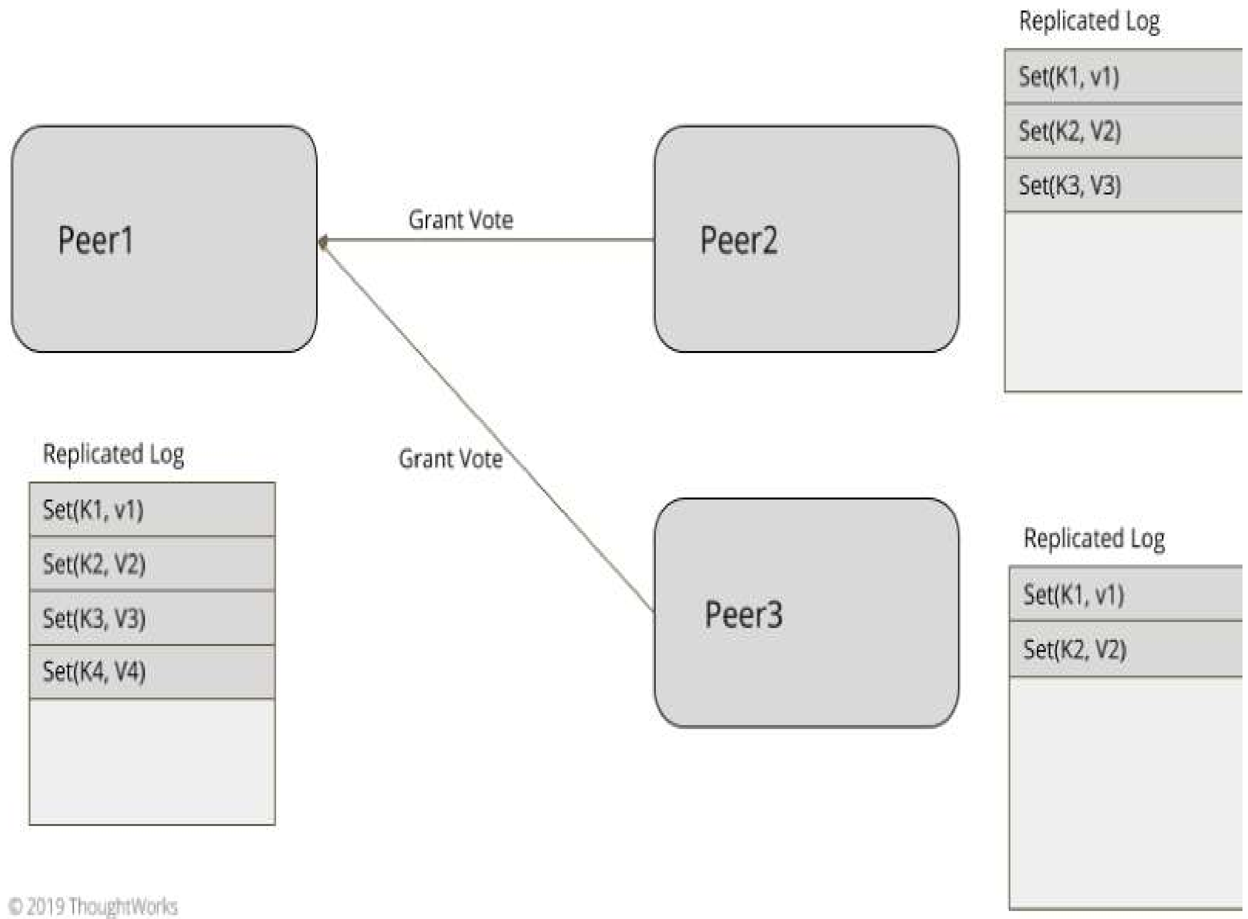
Solution

Select one server amongst the cluster as leader. The leader is responsible for taking decisions on behalf of the entire cluster and propagating the decisions to all the other servers.

Every server at startup looks for an existing leader. If no leader is found, it triggers a leader election. The servers accept requests only after a leader is selected successfully. Only the leader handles the client requests. If a request is sent to a follower server, the follower can forward it to the leader server.

Leader Election





Replicated Log

Set(K1, v1)

Set(K2, V2)

Set(K3, V3)

Set(K4, V4)

Replicated Log

Set(K1, v1)

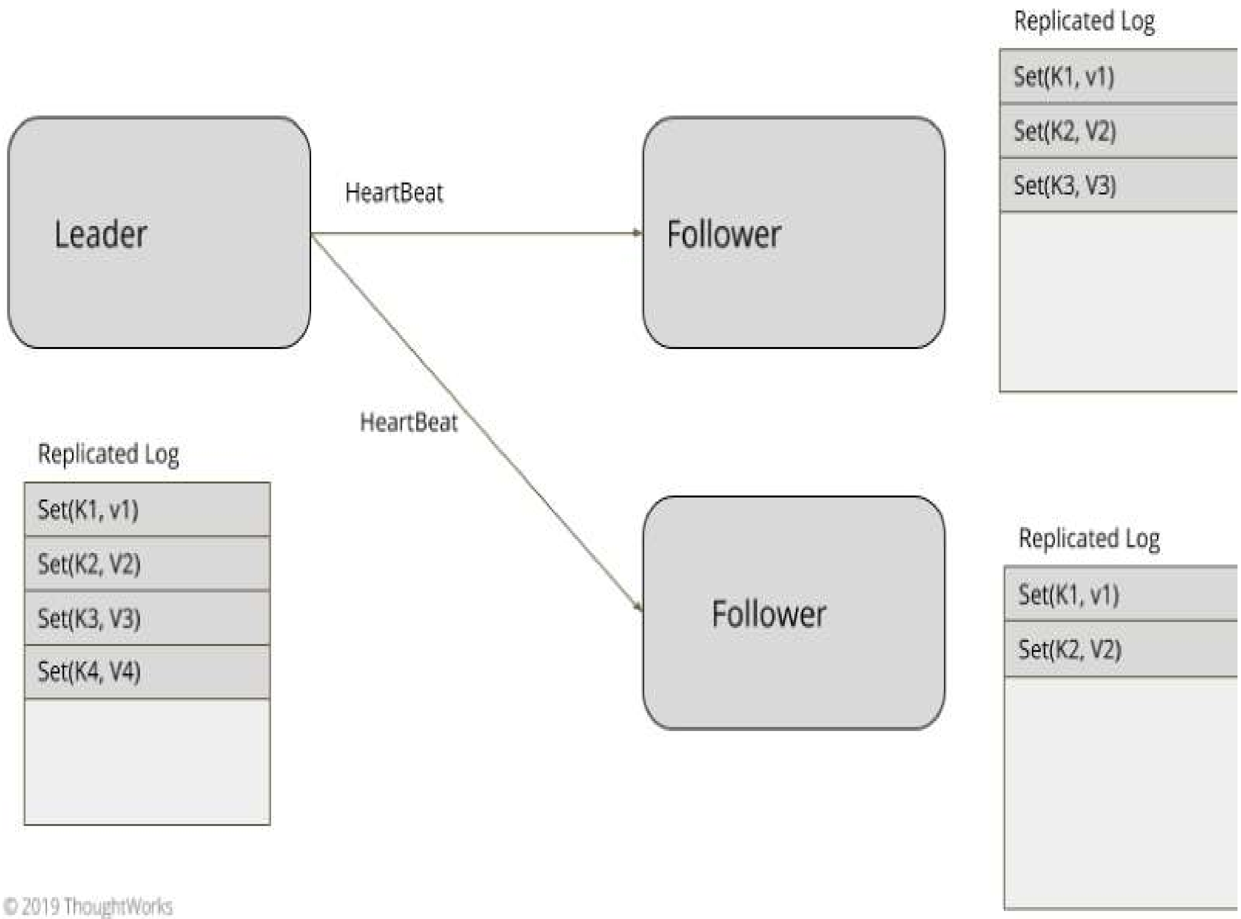
Set(K2, V2)

Set(K3, V3)

Replicated Log

Set(K1, v1)

Set(K2, V2)



© 2019 ThoughtWorks

For smaller clusters of three to five nodes, like in the systems which implement consensus, leader election can be implemented within the data cluster itself without dependending on any external system. Leader election happens at server startup. Every server starts a leader election at startup and tries to elect a leader. The system does not accept any client requests unless a leader is elected. As explained in the *Generation Clock* pattern, every leader election also needs to update the generation number. The server can always be in one of the three states, Leader, Follower or Looking For Leader (or Candidate)

```
public enum ServerRole {
  LOOKING_FOR_LEADER,
  FOLLOWING,
  LEADING;
}
```

HeartBeat mechanism is used to detect if an existing leader has failed, so that new leader election can be started.

Concurrency, Locks and State updates

State updates can be done without any hassle of manipulating synchronization and locking by using *Singular Update Queue*

New leader election is started by sending each of the peer servers a message requesting a vote.

class ReplicatedLog…

```
private void startLeaderElection() {
  replicationState.setGeneration(replicationState.getGeneration() +
  registerSelfVote();
  requestVoteFrom(followers);
}
```

Election Algorithm

ZAB and RAFT

There are two popular mainstream implementations which have leader election algorithms with few subtle differences. Zab [bib-zab] as part of Zookeeper implementation and leader election algorithm in Raft [bib-raft]

There are subtle differences in things like the point at which the generation number is incremented, the default state the server starts in and how to make sure there are no split votes. In Zab, each server looks for the leader at the startup, generation number is incremented only by a leader when its elected and split vote is avoided by making sure each server runs the same logic to choose a leader when multiple servers are equally upto date. In the case of RAFT, servers start in the follower

> state by default, expecting to get heartbeats from the existing leader. If no heartbeat is received, they start election by incrementing the generation number. The split vote is avoided by using randomized timeouts before starting the election.

There are two factors which are considered while electing a leader.

- Because these systems are mostly used for data replication, it puts some extra restrictions on which servers can win the election. Only servers, which are the ‘most up to date’ can be a legitimate leader. For example, in typical consensus based systems, The ‘most up to date’ is defined by two things:
  - The latest *Generation Clock*
  - The latest log index in *Write-Ahead Log*
- If all the servers are equally upto date, then the leader is chosen based following criterias:
  - Some implementation specific criteria, like which server is ranked better or has higher id. (e.g. Zab [bib-zab])
- If care is taken to make sure only one server asks for a vote at a time, then whichever server starts the election before others. (e.g. Raft [bib-raft])

Once a server is voted for in a given Generation Clock, the same vote is returned for that generation always. This makes sure that some other server requesting a vote for the same generation is not elected, when a successful election has already happened. The handling of vote request happens as following:

class ReplicatedLog…

```
VoteResponse handleVoteRequest(VoteRequest voteRequest) {
  //for higher generation request become follower.
  // But we do not know who the leader is yet.
  if (voteRequest.getGeneration() > replicationState.getGeneration(
     becomeFollower(LEADER_NOT_KNOWN, voteRequest.getGeneration());
  }

  VoteTracker voteTracker = replicationState.getVoteTracker();
  if (voteRequest.getGeneration() == replicationState.getGeneration
     if (isUptoDate(voteRequest) && !voteTracker.alreadyVoted()) {
        voteTracker.registerVote(voteRequest.getServerId());
        return grantVote();
      }
     if (voteTracker.alreadyVoted()) {
        return voteTracker.votedFor == voteRequest.getServerId() ?
             grantVote() : rejectVote();

     }
   }
   return rejectVote();
 }

 private boolean isUptoDate(VoteRequest voteRequest) {
   boolean result = voteRequest.getLastLogEntryGeneration() > wal.g
       || (voteRequest.getLastLogEntryGeneration() == wal.getLastLo
       voteRequest.getLastLogEntryIndex() >= wal.getLastLogIndex())
   return result;
 }
```

The server which receives votes from the majority of the servers, transitions to leader state. The majority is determined as discussed in *Quorum*. Once elected, the leader continuously sends *HeartBeat* to all the followers. If followers do not get a heartbeat in specified time interval, a new leader election is triggered.

Leader Election using External [Linearizable] [bib-Linearizable] Store

Running a leader election within a data cluster works well for smaller clusters. For large data clusters, which can be upto few thousand nodes, it’s easier to use an external store like Zookeeper or etcd. (which internally use consensus and provide linearizability guarantees). These large clusters typically have a server which is marked as a master or a controller node, which makes all the decisions on behalf of the entire cluster. There are three functionalities needed for implementing a leader election:

- A compareAndSwap instruction to set a key atomically.
- A heartbeat implementation to expire the key if no heartbeat is received from the elected leader, so that a new election can be triggered.
- A notification mechanism to notify all the interested servers if a key expires.

For electing the leader, each server uses the compareAndSwap instruction to try and create a key in the external store, and whichever server succeeds first, is elected as a leader. Depending on the external store used, the key is created with a small time to live. The elected leader repeatedly updates the key before the time to live value. Every server can set a watch on this key, and servers get notified if the key expires without getting updated from the existing leader within the time to live setting. e.g. etcd [bib-etcd] allows a compareAndSwap operation, by allowing a set key operation only if the key does not exist previously. In Zookeeper [bib-zookeeper] there is no explicit compareAndSwap kind of operation supported, but it can be implemented by trying to create a node, and expecting an exception if the node already exists. There is no explicit time to live either, but zookeeper has a concept of ephemeral node. The node exists until the server has an active session with zookeeper, else the node is removed and everyone who is watching that node is notified. For example, Zookeeper can be used to elect leader as following:

class ServerImpl…

```
public void startup() {
  zookeeperClient.subscribeLeaderChangeListener(this);
  elect();
}

public void elect() {
  var leaderId = serverId;
  try {
     zookeeperClient.tryCreatingLeaderPath(leaderId);
     this.currentLeader = serverId;
     onBecomingLeader();
  } catch (ZkNodeExistsException e) {
    //back off
    this.currentLeader = zookeeperClient.getLeaderId();
  }
 }
```

All other servers watch for the liveness of the existing leader. When it is detected that the existing leader is down, a new leader election is triggered. The failure detection happens using the same external linearizable store used for the leader election. This external store also has facilities to implement group membership and failure detection mechanisms. For example, extending the above Zookeeper based implementation, a change listener can be configured with Zookeeper which is triggered when a change in the existing leader node happens.

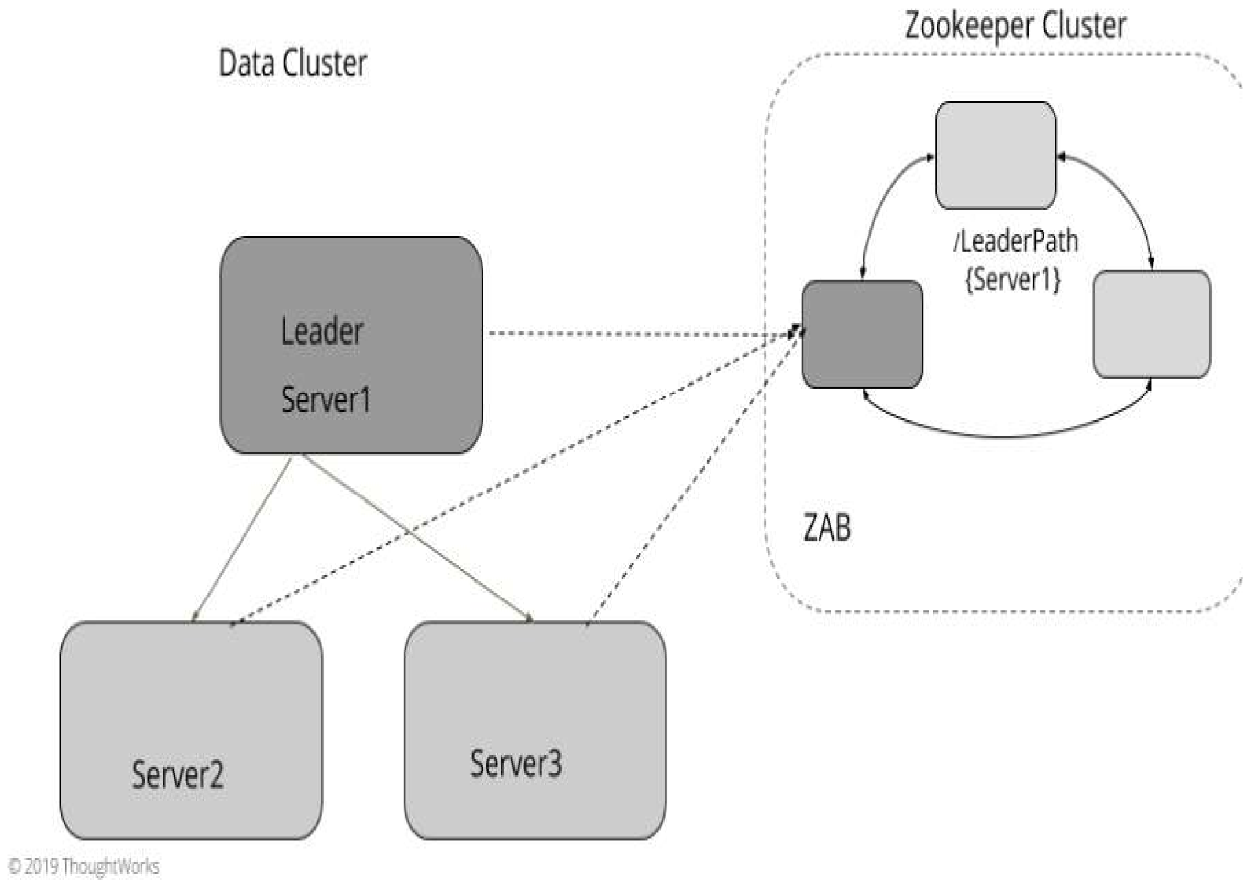
class ZookeeperClient…

```
public void subscribeLeaderChangeListener(IZkDataListener listener)
  zkClient.subscribeDataChanges(LeaderPath, listener);
}
```

Every server in the cluster subscribes for this change, and whenever the callback is triggered, a new election is triggered again the same way shown above.

class ServerImpl…

```
@Override
public void handleDataDeleted(String dataPath) throws Exception {
  elect();
}
```



Systems like etcd [bib-etcd] or Consul [bib-consul] can be used the same way to implement leader election.

Why Quorum read/writes are not enough for strong consistency guarantees

It might look like Quorum read/write, provided by Dynamo style databases like Cassandra, is enough for getting strong consistency in case of server failures. But that is not the case. Consider the following example. Let's say we have a cluster with three servers. Variable x is stored on all three servers. (It has a replication factor of 3). Value of x = 1 at startup.

- Let's say writer1 writes x = 2, with replication factor of 3. The write request is sent to all the three servers. The write is successful on server1 but fails for server2 and server3. (either a network glitch or writer1 just went into a long garbage collection pause after sending the write request to server1.).

- Client c1 reads the value of x from server1 and server2. It gets the latest value of x=2 because server1 has the latest value.
- Client c2 triggers a read for x. But Server1 goes down temporarily. So c1 reads it from server2, server3, which have old values for x, x=1. So c2 gets the old value even when it read it after c1 read the latest value.

This way two consecutive reads show the latest values disappearing. Once server1 comes back up, subsequent reads will give the latest value. And assuming the read repair or anti entropy process is running, the rest of the servers will get the latest value as well ‘eventually’. But there is no guarantee provided by the storage cluster to make sure that once a particular value is visible to any clients, all subsequent reads will continue to get that value even if a server fails.

Examples

- For systems which implement consensus, it’s important that only one server coordinates activities for the replication process. As noted in the paper Paxos Made Simple [bib-lamport-paxos-simple], it’s important for the liveness of the system.
- In Raft [bib-raft] and Zab [bib-zab] consensus algorithms, leader election is an explicit phase that happens at the startup or on the leader failure
- viewstamped replication [bib-view-stamp-replication] algorithm has a concept of Primary, similar to leader in other algorithms
- Kafka [bib-kafka] has a Controller [bib-kafka-controller] which is responsible for taking all the decisions on behalf of the rest of the cluster. It reacts to events from Zookeeper and for each partition in Kafka, there is a designated leader broker and follower brokers. The leader and follower selection is done by the Controller broker.

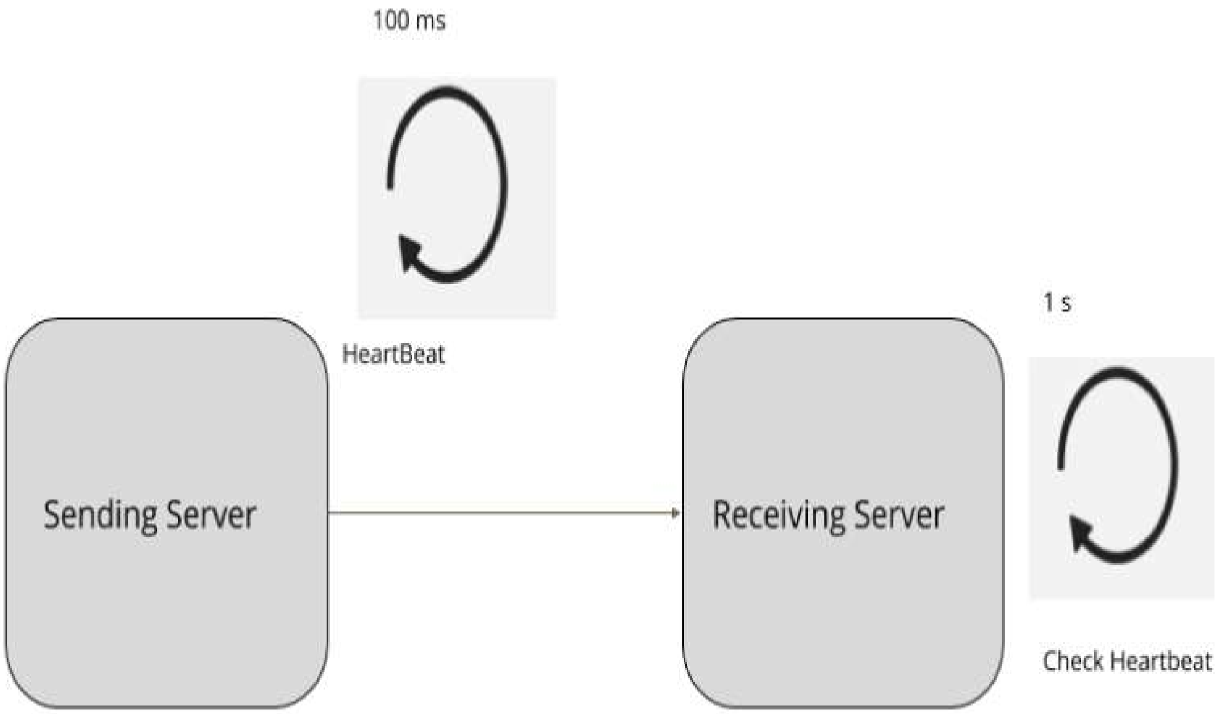
Chapter 7. HeartBeat

Show a server is available by periodically sending a message to all the other servers.

Problem

When multiple servers form a cluster, the servers are responsible for storing some portion of the data, based on the partitioning and replication schemes used. Timely detection of server failures is important to make sure corrective actions can be taken by making some other server responsible for handling requests for the data on failed servers.

Solution



© 2019 ThoughtWorks

Periodically send a request to all the other servers indicating liveness of the sending server. Select the request interval to be more than the network round trip time between the servers. All the servers wait for the timeout interval, which is multiple of the request interval to check for the heartbeats. In general,

> It is useful to know the network round trip times within and between datacenters when deciding values for heartbeat interval and timeouts. [numbers-every-programmer-should-know] [bib-numbers-every-programmer-should-know] is a good reference.

Timeout Interval > Request Interval > Network round trip time between the servers.

e.g. If the network round trip time between the servers is 20ms, the heartbeats can be sent every 100ms, and servers check after 1 second to give enough time for multiple heartbeats to be sent and not get false negatives. If no heartbeat is received in this interval, then it declares the sending server as failed.

Both the servers, the one sending the heartbeat and the one receiving it, have a scheduler defined as follows. The scheduler is given a method to be executed at a regular interval. When started, the task is scheduled to execute the given method.

class HeartBeatScheduler…

```
public class HeartBeatScheduler implements Logging {
  private ScheduledThreadPoolExecutor executor = new ScheduledThrea

  private Runnable action;
  private Long heartBeatInterval;
  public HeartBeatScheduler(Runnable action, Long heartBeatInterval
    this.action = action;
    this.heartBeatInterval = heartBeatIntervalMs;
  }

  private ScheduledFuture<?> scheduledTask;
  public void start() {
    scheduledTask = executor.scheduleWithFixedDelay(new HeartBeatTa
  }
```

On the sending server side, the scheduler executes a method to send heartbeat messages.

class SendingServer…

```
private void sendHeartbeat() throws IOException {
   socketChannel.blockingSend(newHeartbeatRequest(serverId));
}
```

On the receiving server, the failure detection mechanism has a similar scheduler started. At regular intervals, it checks if the heartbeat was received or not.

class AbstractFailureDetector…

```
private HeartBeatScheduler heartbeatScheduler = new HeartBeatSchedu
abstract void heartBeatCheck();
abstract void heartBeatReceived(T serverId);
```

The failure detector needs to have two methods:

class ReceivingServer…

```
private void handleRequest(Message<RequestOrResponse> request) {
  RequestOrResponse clientRequest = request.getRequest();
  if (isHeartbeatRequest(clientRequest)) {
     HeartbeatRequest heartbeatRequest = JsonSerDes.deserialize(cli
     failureDetector.heartBeatReceived(heartbeatRequest.getServerId
     sendResponse(request);
  } else {
     //processes other requests
  }
}
```

- A method to be called whenever the receiving server receives the heartbeat, to tell the failure detector that heartbeat is received
- A method to periodically check the heartbeat status and detect possible failures.

The implementation of when to mark a server as failed depends on various criterias. There are different trade offs. In general, the smaller the heartbeat interval, the quicker the failures are detected, but then there is higher probability of false failure detections. So the heartbeat intervals and interpretation of missing heartbeats is implemented as per the requirements of the cluster. In general there are following two broad categories.

Small Clusters - e.g. Consensus Based Systems like RAFT, Zookeeper

In all the consensus implementations, Heartbeats are sent from the leader server to all followers servers. Every time a heartbeat is received, the timestamp of heartbeat arrival is recorded

class TimeoutBasedFailureDetector…

```
@Override
public void heartBeatReceived(T serverId) {
  Long currentTime = System.nanoTime();
  heartbeatReceivedTimes.put(serverId, currentTime);
  markUp(serverId);
}
```

If no heartbeat is received in a fixed time window, the leader is considered crashed, and a new server is elected as a leader. There are chances of false failure detection because of slow processes or networks. So *Generation Clock* needs to be used to detect the stale leader. This provides better availability of the system, as crashes are detected in smaller time periods. This is suitable for smaller clusters, typically 3 to 5 node setup which is observed in most consensus implementations like Zookeeper or Raft.

class TimeoutBasedFailureDetector…

```
@Override
void heartBeatCheck() {
  Long now = System.nanoTime();
  Set<T> serverIds = heartbeatReceivedTimes.keySet();
  for (T serverId : serverIds) {
    Long lastHeartbeatReceivedTime = heartbeatReceivedTimes.get(ser
    Long timeSinceLastHeartbeat = now - lastHeartbeatReceivedTime;
    if (timeSinceLastHeartbeat >= timeoutNanos) {
       markDown(serverId);
    }
    }
}
```

Technical Considerations

When *Single Socket Channel* is used to communicate between servers, care must be taken to make sure that the [head-of-line-blocking] [bib-head-of-line-blocking] does not prevent heartbeat messages from being processed. Otherwise it can cause delays long enough to falsely detect the sending server to be down, even when it was sending heart beats at the regular intervals. *Request Pipeline* can be used to make sure servers do not wait for the response of previous requests before sending heartbeats. Sometimes, when using *Singular Update Queue*, some tasks like disk writes, can cause delays which might delay processing of timing interrupts and delay sending heartbeats.

This can be solved by using a separate thread for sending heartbeats asynchronously. Frameworks like Consul [bib-consul] and Akka [bib-akka] send heartbeats asynchronously. This can be the issue on receiving servers as well. A receiving server doing a disk write, can check the heartbeat only after the write is complete, causing false failure detection. So the receiving server using Singular Update Queue can reset its heartbeat-checking mechanism to incorporate those delays. Reference implementation of Raft [bib-raft], [log-cabin] [bib-log-cabin] does this.

Sometimes, a [local-pause] [bib-local-pause] because of some runtime-specific events like Garbage Collection can delay the processing of heartbeats. There needs to be a mechanism to check if the processing is happening after a possible local pause. A simple mechanism, to use is to check if the processing is happening after a long enough time window, e.g. 5 seconds. In that case nothing is marked as failed based on the time window, and it's deferred to the next cycle. Implementation in Cassandra [bib-cassandra-local-pause-detection] is a good example of this.

Large Clusters. Gossip Based Protocols

Heartbeating, described in the previous section, does not scale to larger clusters with a few hundred to thousand servers spanning across wide area networks. In large clusters, two things need to be considered:

- Fixed limit on the number of messages generated per server
- The total bandwidth consumed by the heartbeat messages. It should not consume a lot of network bandwidth. There should be an upper bound of a few hundred kilo bytes, making sure that too many heartbeat messages do not affect actual data transfer across the cluster.

For these reasons,all-to-all heartbeating is avoided. Failure detectors, along with Gossip [bib-gossip] protocols for propagating failure information across the cluster are typically used in these situations. These clusters typically take actions like moving data across nodes in case of failures, so prefer correct detections and tolerate some more delays (although bounded). The main challenge is not to incorrectly detect a node as faulty because of network delays or slow processes. A common mechanism used then, is for each process to be assigned a suspicion number, which increments if there is no gossip including that process in bounded time. It’s calculated based on past statistics, and only after this suspicion number reaches a configured upper limit, is it marked as failed.

There are two mainstream implementations: 1) Phi Accrual failure detector (used in Akka, Cassandra) 2) SWIM with Lifeguard enhancement (used in Hashicorp Consul, memberlist) These implementations scale over a wide area network with thousands of machines. Akka is known to be tried for 2400 [bib-akka-2400-node-cluster] servers. Hashicorp Consul is routinely deployed with several thousand consul servers in a group. Having a reliable failure detector, which works efficiently for large cluster deployments and at the same time provides some consistency guarantees, remains an area of active development. Some recent developments in frameworks like Rapid [bib-rapid] look promising.

Examples

- Consensus implementations like ZAB or RAFT, which work with a small three to five node cluster, implement a fixed time window based failure detection.
- Akka Actors and Cassandra use Phi Accrual failure detector [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.7427&rep=rep1&type=pdf].
- Hashicorp consul use gossip based failure detector SWIM [https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf].

Chapter 8. Paxos

by Unmesh Joshi and Martin Fowler

Use two consensus building phases to reach safe consensus even when nodes disconnect

Problem

When multiple nodes share state, they often need to agree between themselves on a particular value. With *Leader and Followers*, the leader decides and passes its value to the followers. But if there is no leader, then the nodes need to determine a value themselves. (Even with a leader-follower, they may need to do this to elect a leader.)

A leader can ensure replicas safely acquire an update by using *Two Phase Commit*, but without a leader we can have competing nodes attempt to gather a *Quorum*. This process is further complicated because any node may fail or disconnect. A node may achieve quorum on a value, but disconnect before it is able to communicate this value to the entire cluster.

Solution

> In the original paxos papers (1998 [bib-lamport-paxos-original] and 2001 [bib-lamport-paxos-simple]), there is no mention of the commit phase,as the focus of the algorithm is to prove that only a single value is chosen and it’s enough, even if only the proposer cluster node knows about the chosen value. But in practice, all of the cluster nodes need to know about the chosen value and there is a need for a commit phase where the proposer communicates the decision to all of the cluster nodes.

The Paxos algorithm was developed by Leslie Lamport [bib-lamport], published in his 1998 paper The Part-Time Parliament [bib-lamport-paxos-original]. Paxos works in three phases to make sure multiple nodes agree on the same value in spite of partial network or node failures. The first two phases act to build consensus around a value, the last phase then communicates that consensus to the remaining replicas.

- Prepare phase: establish the latest *Generation Clock* and gather any already accepted values.
- Accept phase: propose a value for this generation for replicas to accept.
- Commit Phase: let all the replicas know that a value has been chosen.

In the first phase (called *prepare phase*), the node proposing a value (called a *proposer*) contacts all the nodes in the cluster (called *acceptors*) and asks them if they will promise to consider its value. Once a quorum of acceptors return such a promise, the proposer moves onto the second phase. In the second phase (called the *accept phase*) the proposer sends out a proposed value, if a quorum [1] of nodes accepts this value then the value is *chosen*. In the final phase (called the *commit phase*), the proposer can then commit the chosen value to all the nodes in the cluster.

[1] The original description of Paxos requires majority Quorum in both the prepare and the accept phases. Some recent work by Heidi Howard and others [bib-flexible-paxos] show that the main requirement of Paxos is to have overlap in the quorums of the prepare and the accept phase. As long as this requirement is fulfilled, it does not require a majority Quorum in both the phases.

Flow of the Protocol

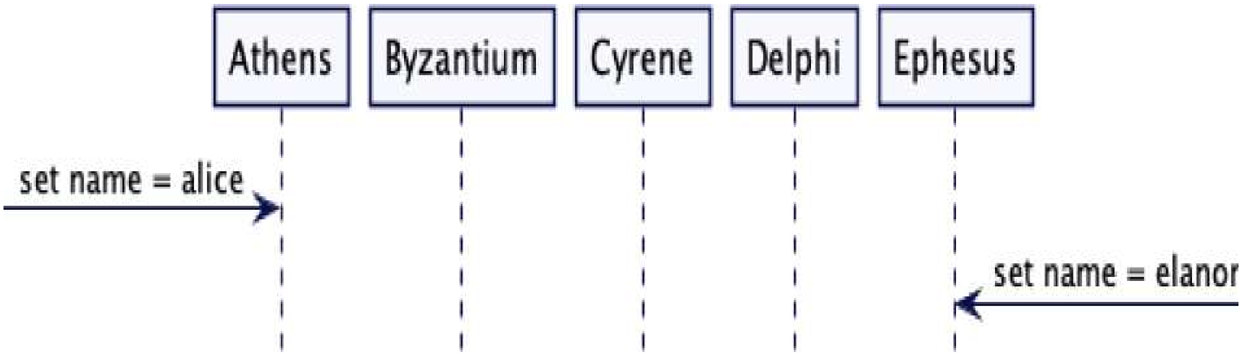
Paxos is a difficult protocol to understand. We’ll start by showing an example of a typical flow of the protocol, and then dig into some of the details of how it works. We intend this explanation to provide an intuitive sense of how the protocol works, but not as a comprehensive description to base an implementation upon.

Here’s a very brief summary of the protocol.

Table 8.1.

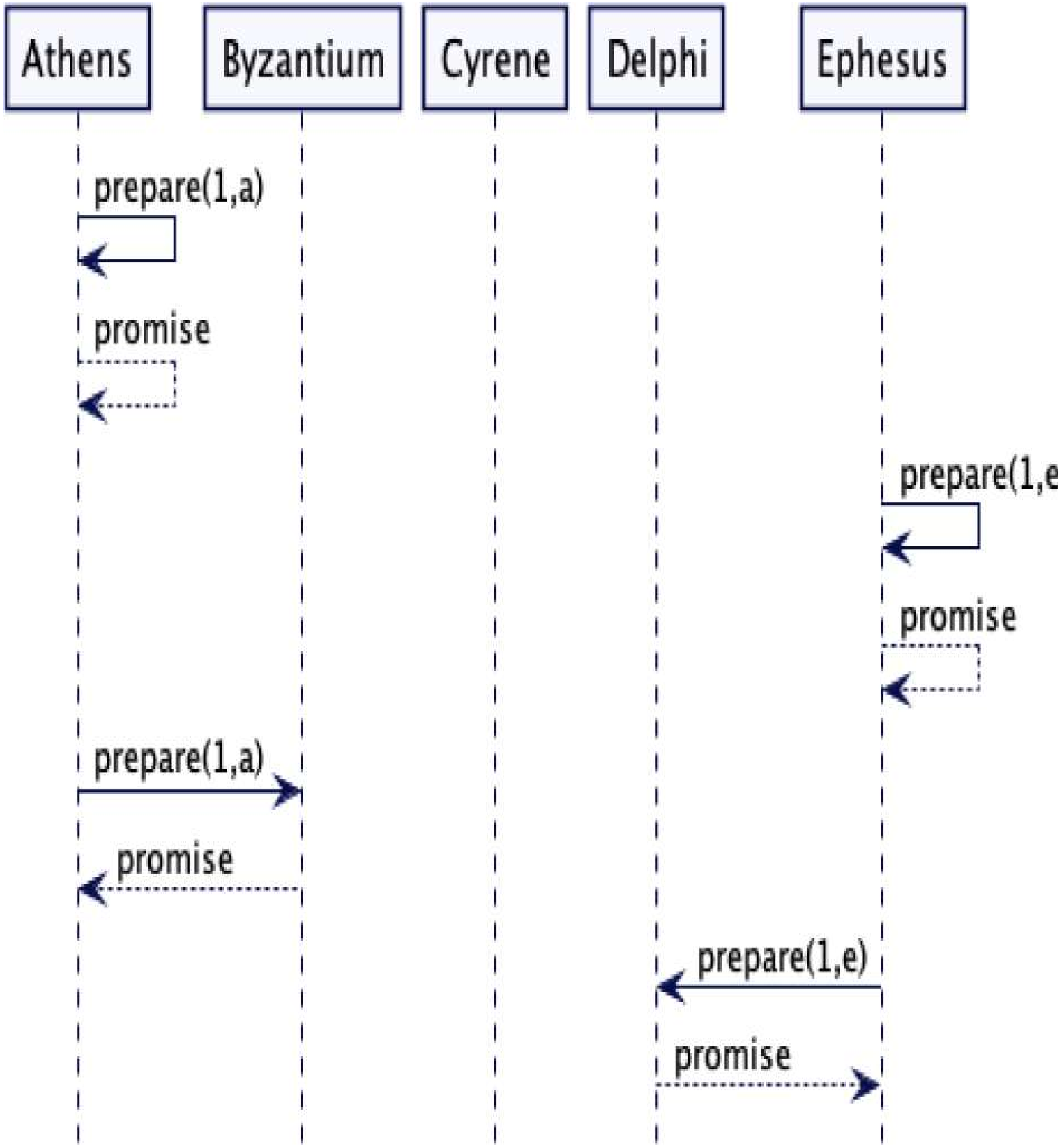
Proposer	Acceptor
Obtains the next generation number from a *Generation Clock*. Sends a prepare request with this generation number to all acceptors.	
	If the generation number of the prep request is later than its promi generation variable, it updates its prom generation with this later value a returns a promise response. If it l already accepted a proposal it returns t proposal.
When it receives promises from quorum of acceptors, it looks to see if any of these responses contain accepted values. If so it changes its own proposed value to that of the returned proposal with the highest generation number. Sends accept requests to all acceptors with its generation number and proposed value.	
	If the generation number of the acc request is later than its promi generation variable it stores the propo as its accepted proposal and responds t it has accepted the request.
When it receives a successful response from a quorum of acceptors, it records the value as chosen and sends commit messages to all nodes.	

Those are basic rules for paxos, but it’s very hard to understand how they combine for an effective behavior. So here’s an example to show how this works.



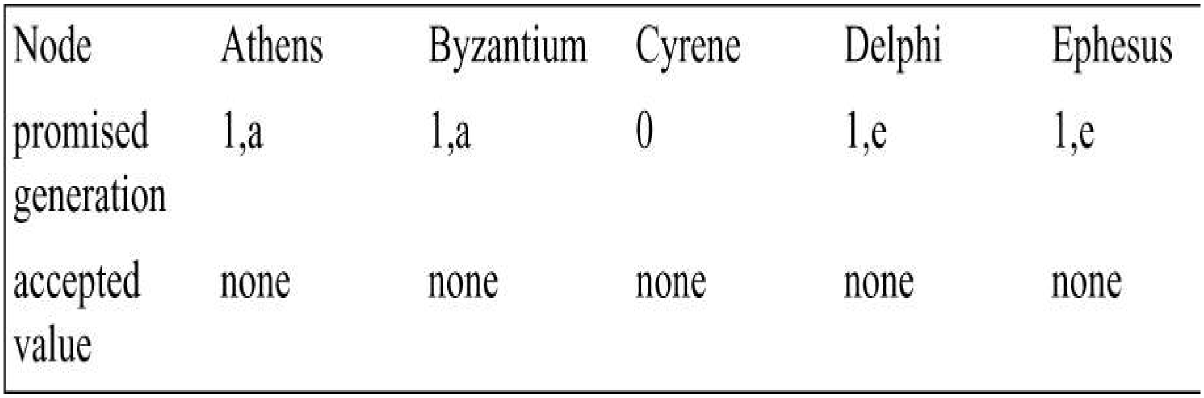
Consider a cluster of five nodes: Athens, Byzantium, Cyrene, Delphi, and Ephesus. A client contacts the Athens node, requesting to set the name to "alice". The Athens node now needs to initiate a Paxos interaction to see if all the nodes will agree to this change. Athens is called the proposer, in that Athens will propose to all the other nodes that the name of the cluster become "alice". All the nodes in the cluster (including Athens) are "acceptors", meaning they are capable of accepting proposals.

At the same time that Athens is proposing "alice", the node Ephesus gets a request to set the name to "elanor". This makes Ephesus also be a proposer.



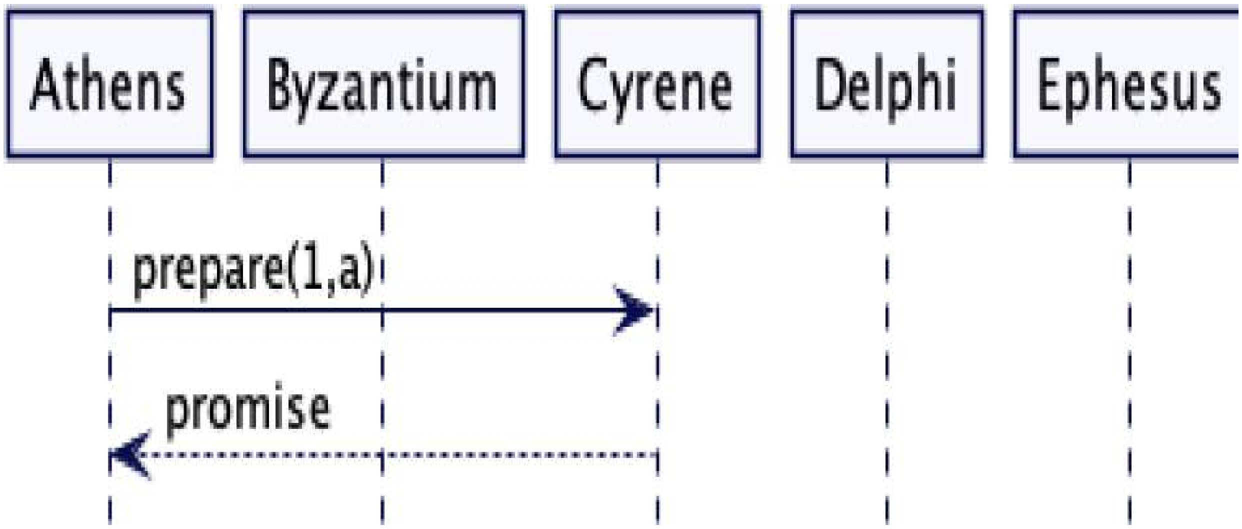
In the prepare phase the proposers begin by sending some prepare requests, which all include a generation number. Since Paxos is intended to avoid single points of failure, we don't take this from a single generation clock. Instead each node maintains its own generation clock where it combines a generation number with a node ID. The node ID is used to break ties, so [2,a] > [1,e] > [1,a]. Each acceptor records the latest promise it's seen so far.

Table 8.2.



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	0	1,e	1,e
accepted value	none	none	none	none	none

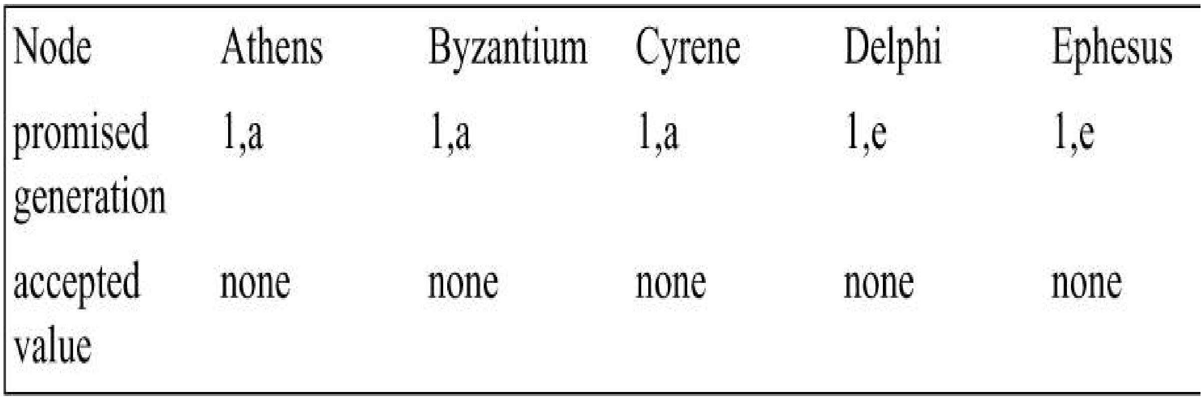
Since they haven’t seen any requests before this, they all return a promise to the calling proposer. We call the returned value a "promise" because it indicates that the acceptor promises to not consider any messages with an earlier generation clock than the promised one.



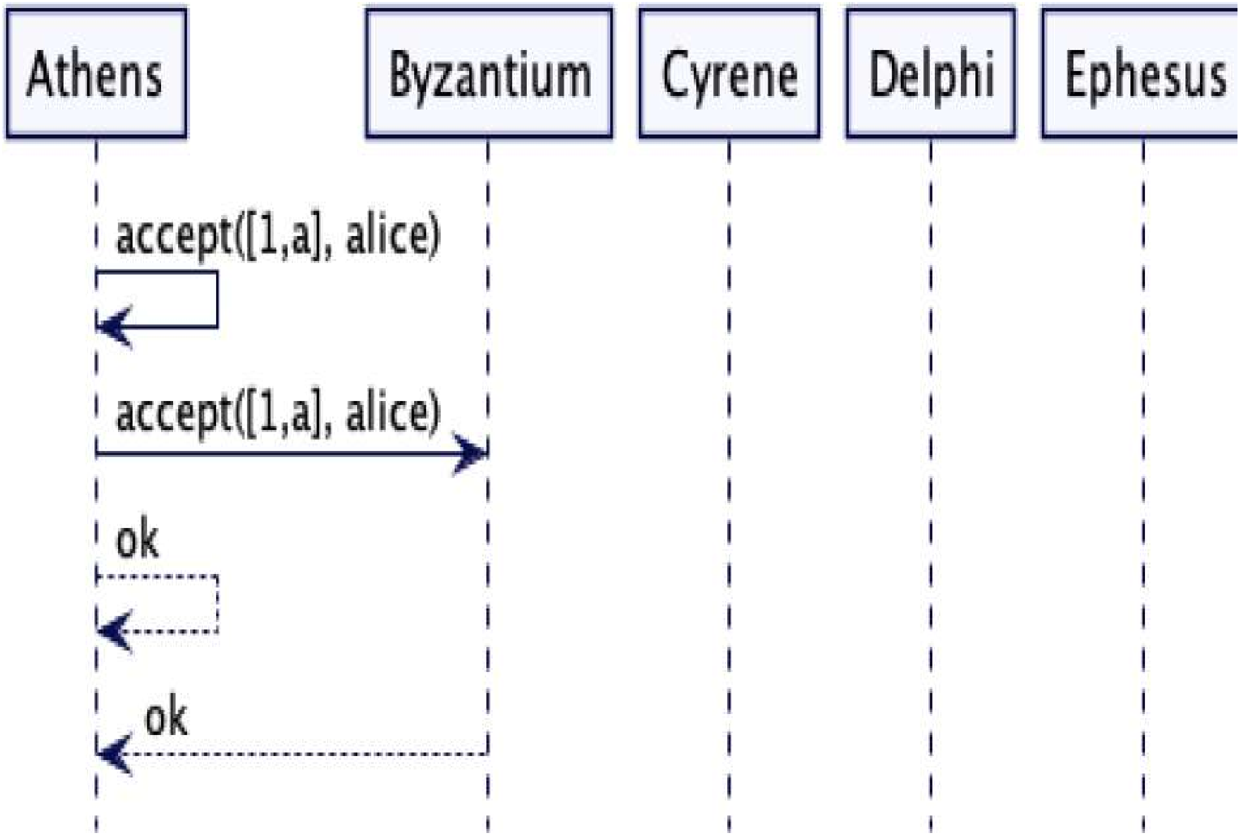
Athens sends its prepare message to Cyrene. When it receives a promise in return, this means it has now got promises from from three of the five nodes, which represents a *Quorum*. Athens now shifts from sending prepare messages to sending accept messages.

It is possible that Athens fails to receive a promise from a majority of the cluster nodes. In that case Athens retries the prepare request by incrementing the generation clock.

Table 8.3.

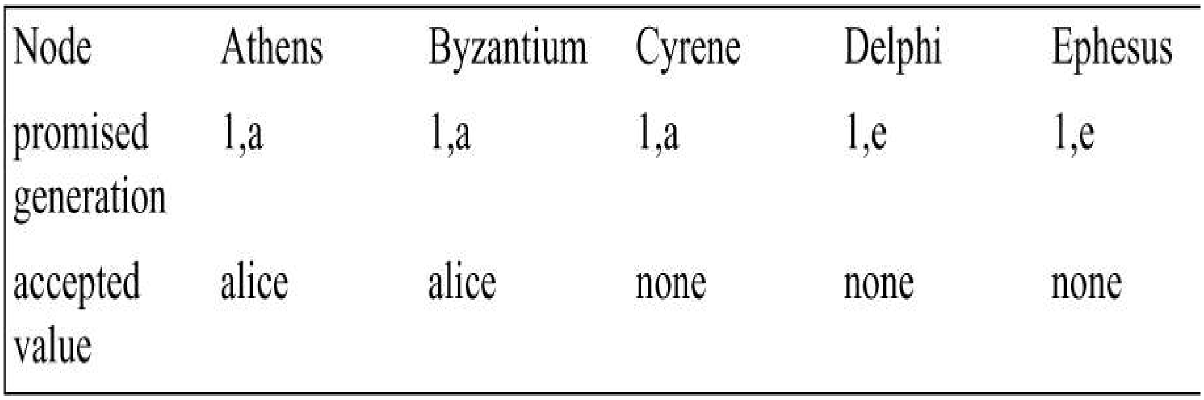


Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,a	1,e	1,e
accepted value	none	none	none	none	none

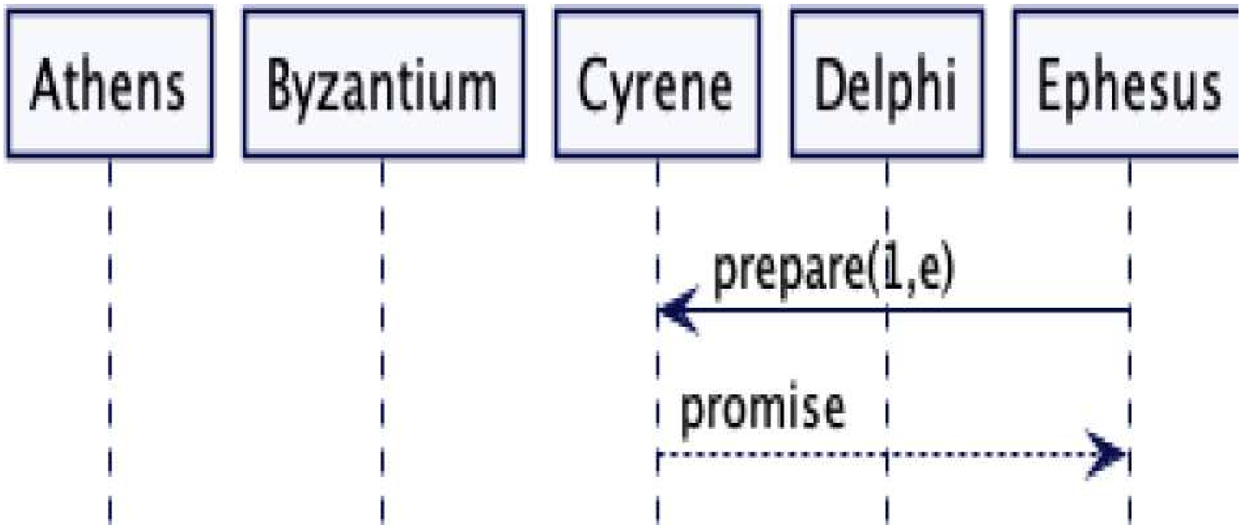


Athens now starts sending accept messages, containing the generation and the proposed value. Athens and Byzantium accept the proposal.

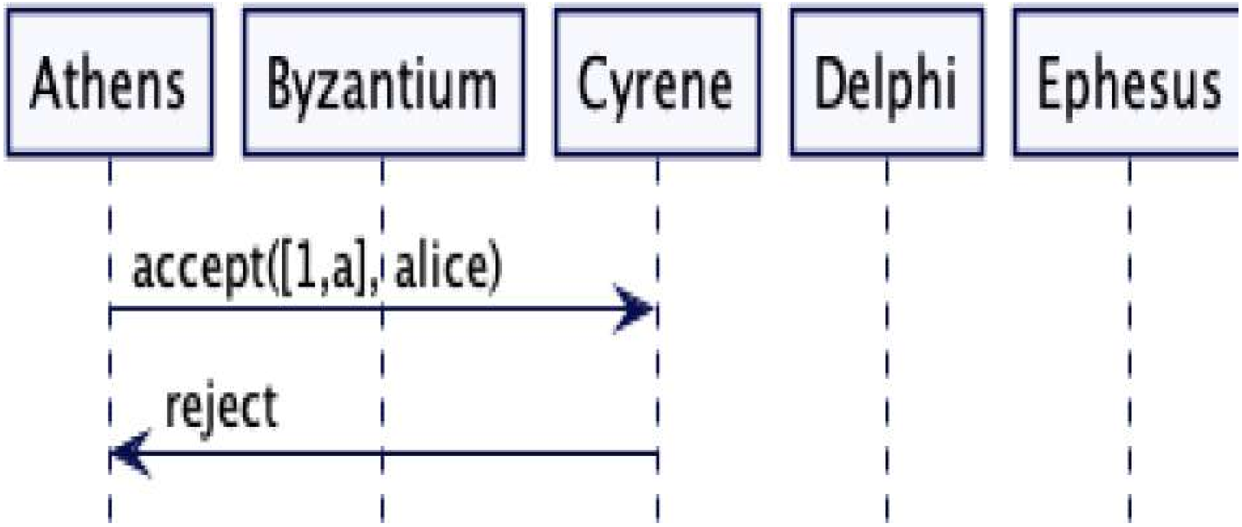
Table 8.4.



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,a	1,e	1,e
accepted value	alice	alice	none	none	none

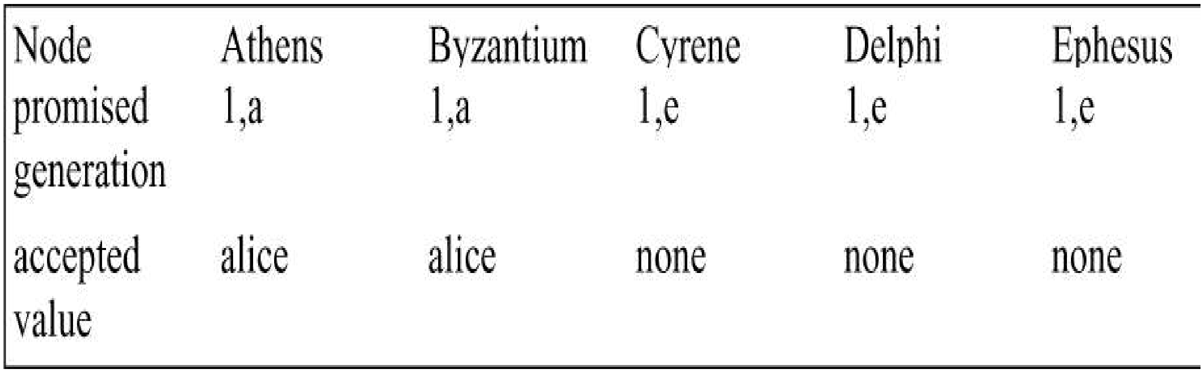


Ephesus now sends a prepare message to Cyrene. Cyrene had sent a promise to Athens, but Ephesus's request has a higher generation, so it takes precedence. Cyrene sends back a promise to Ephesus.

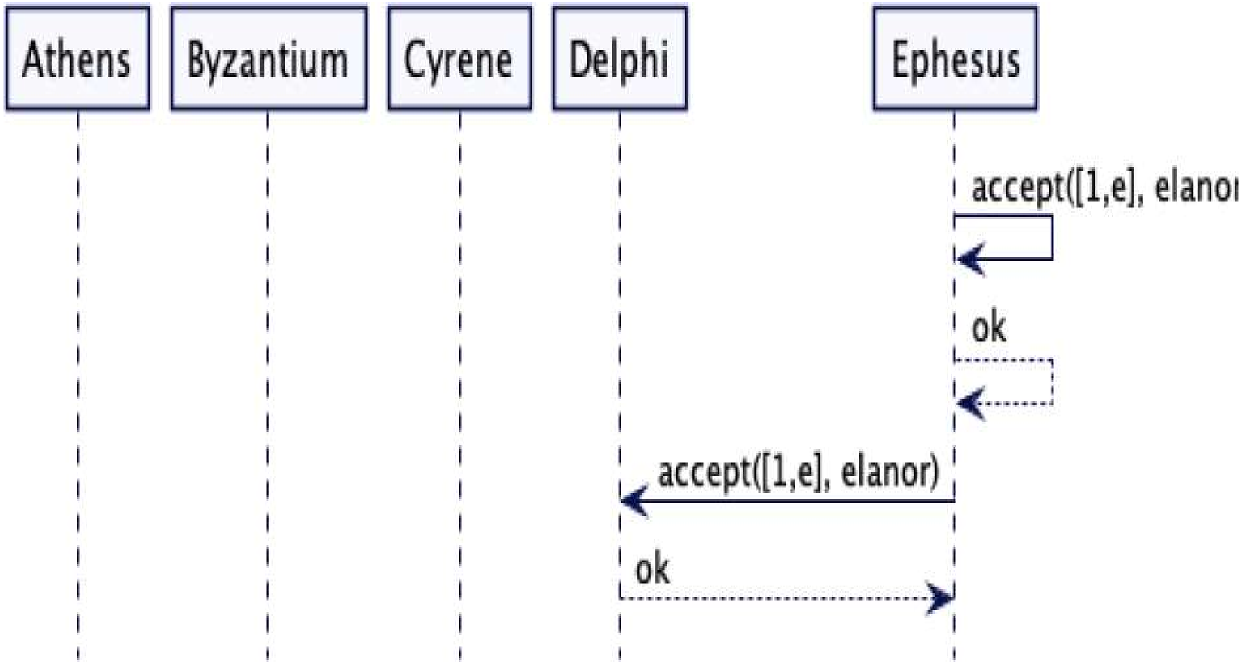


Cyrene now gets an accept request from Athens but rejects it as the generation number is behind its promise to Ephesus.

Table 8.5.

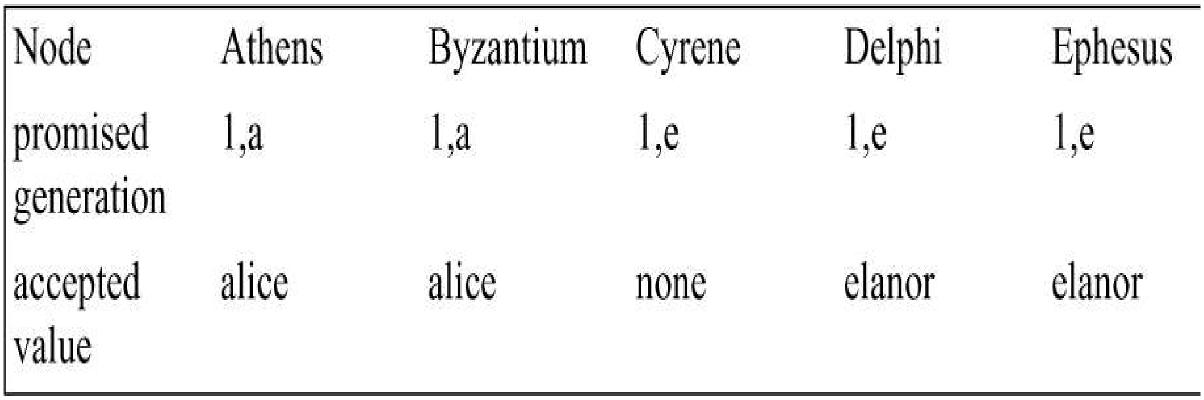


Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,e	1,e	1,e
accepted value	alice	alice	none	none	none



Ephesus has now got a quorum from its prepare messages, so can move on to sending accepts. It sends accepts to itself and to Delphi but then crashes before it can send any more accepts.

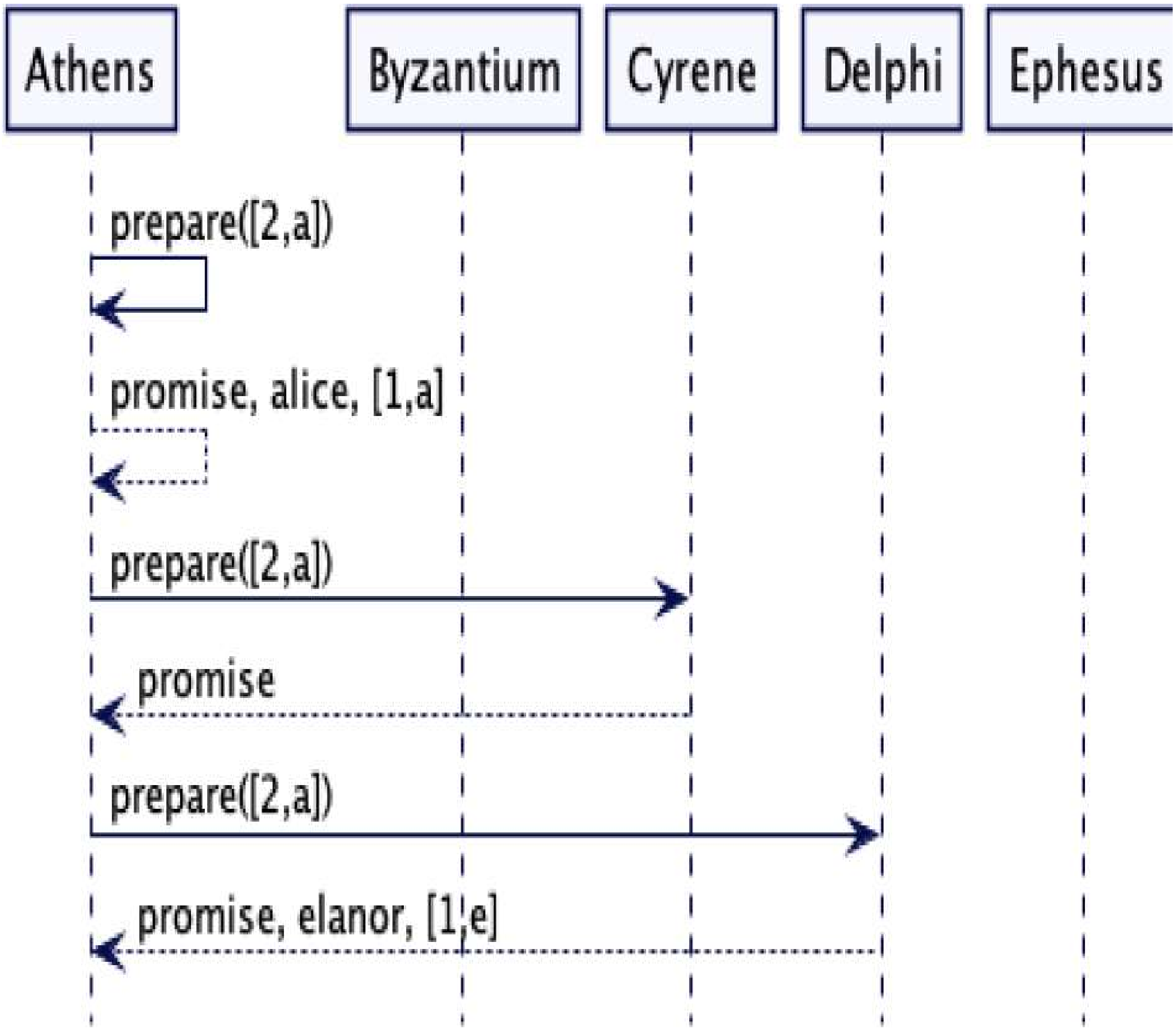
Table 8.6.



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	1,a	1,a	1,e	1,e	1,e
accepted value	alice	alice	none	elanor	elanor

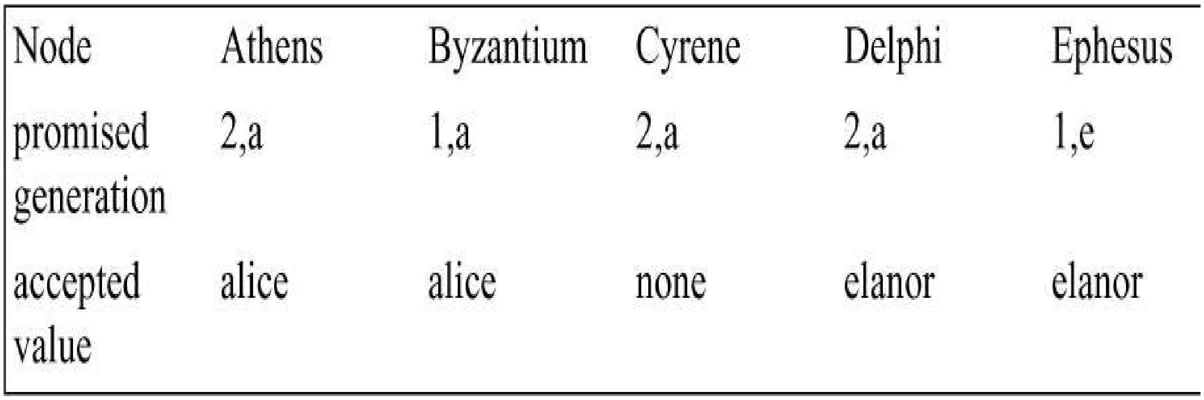
Meanwhile, Athens has to deal with the rejection of its accept request from Cyrene. This indicates that its quorum is no longer promised to it and thus its proposal will fail. This will always happen to a proposer who loses its initial quorum like this; for another proposer to achieve quorum at least one member of the first proposer’s quorum will defect.

In a situation with a simple two phase commit, we would then expect Ephesus to just go on and get its value chosen, but such a scheme would now be in trouble since Ephesus has crashed. If it had a lock on a quorum of acceptors, its crash would deadlock the whole proposal process. Paxos, however, expects this kind of thing to happen, so Athens will make another try, this time with a higher generation.



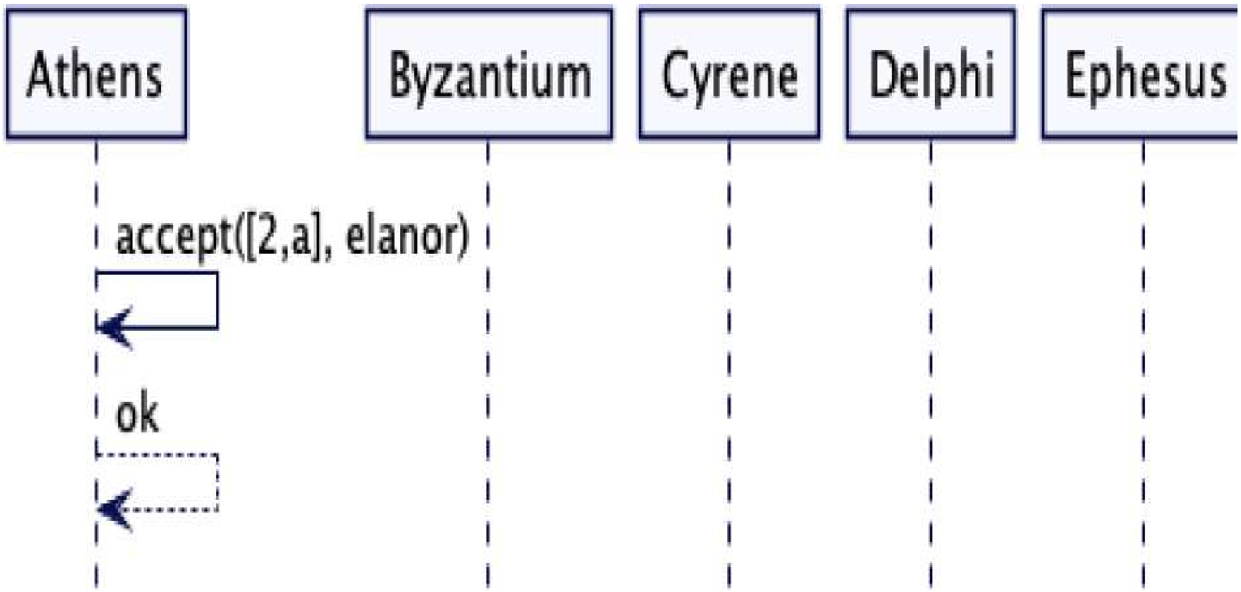
It sends prepare messages again, but this time with a higher generation number. As with the first round, it gets back a trio of promises, but with an important difference. Athens already accepted "alice" earlier, and Delphi had accepted "elanor". Both of these acceptors return a promise, but also the value that they already accepted, together with the generation number of that accepted proposal. When they return that value, they update their promised generation to [2,a] to reflect the promise they made to Athens.

Table 8.7.



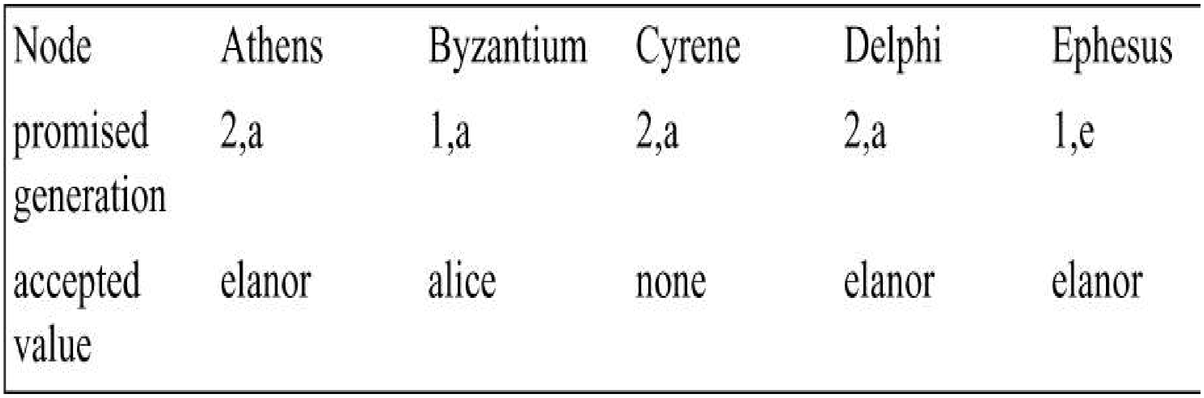
Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	1,a	2,a	2,a	1,e
accepted value	alice	alice	none	elanor	elanor

Athens, with a quorum, must now move onto the accept phase, but it must propose the already-accepted value with the highest generation, which is "elanor", who was accepted by Delphi with a generation of [1,e], which is greater than Athens's acceptance of "alice" with [1,a].



Athens starts to send out accept requests, but now with "elanor" and its current generation. Athens sends an accept request to itself, which is accepted. This is a crucial acceptance because now there are three nodes accepting "elanor", which is a quorum for "elanor", therefore we can consider "elanor" to be the chosen value.

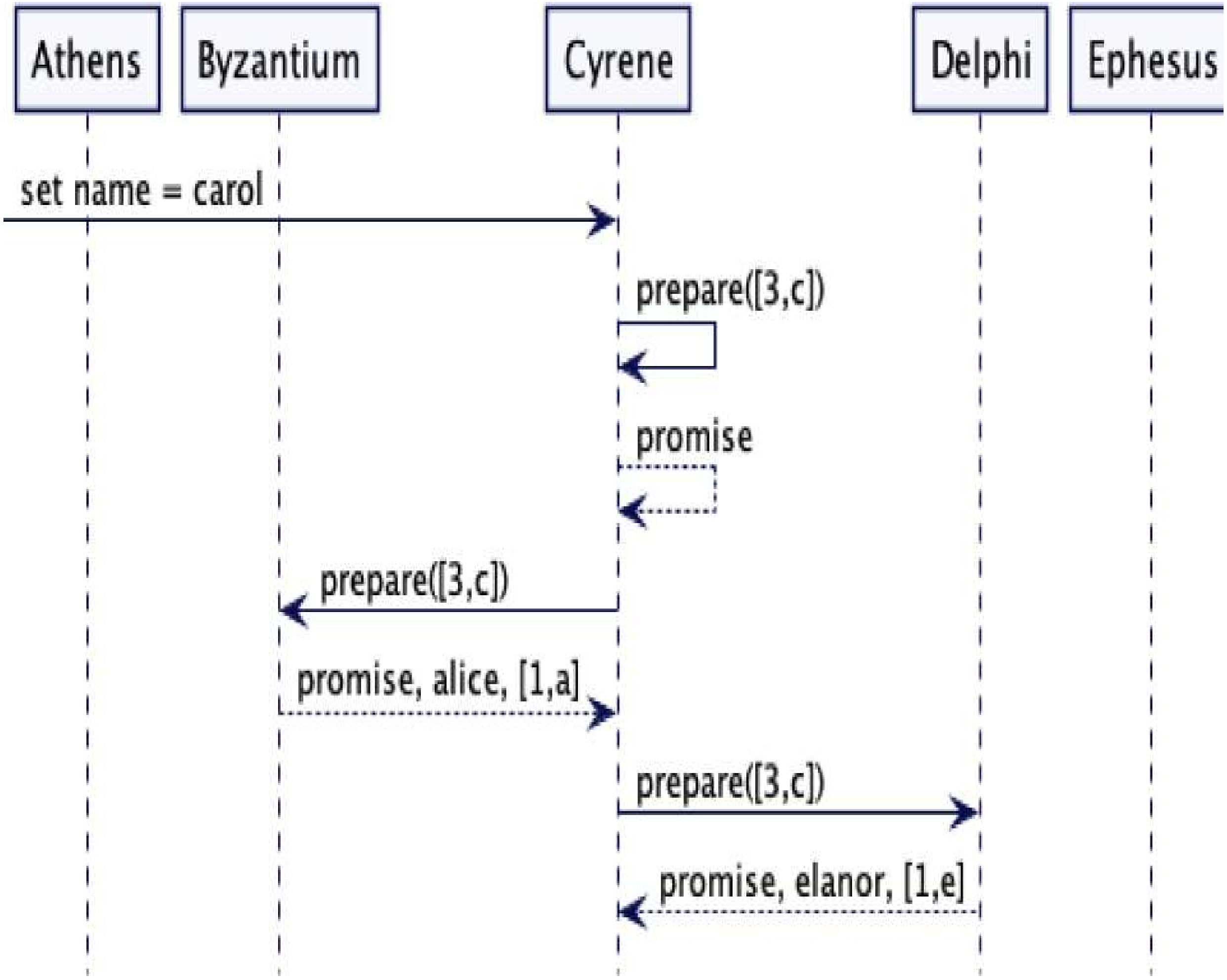
Table 8.8.



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	1,a	2,a	2,a	1,e
accepted value	elanor	alice	none	elanor	elanor

But although "elanor" is now the chosen value, nobody is yet aware of it. Within the accept stage Athens only knows itself having "elanor" as the value, which isn’t a quorum and Ephesus is offline. All Athens needs to do is have a couple more accept requests accepted and it will be able to commit. But now Athens crashes.

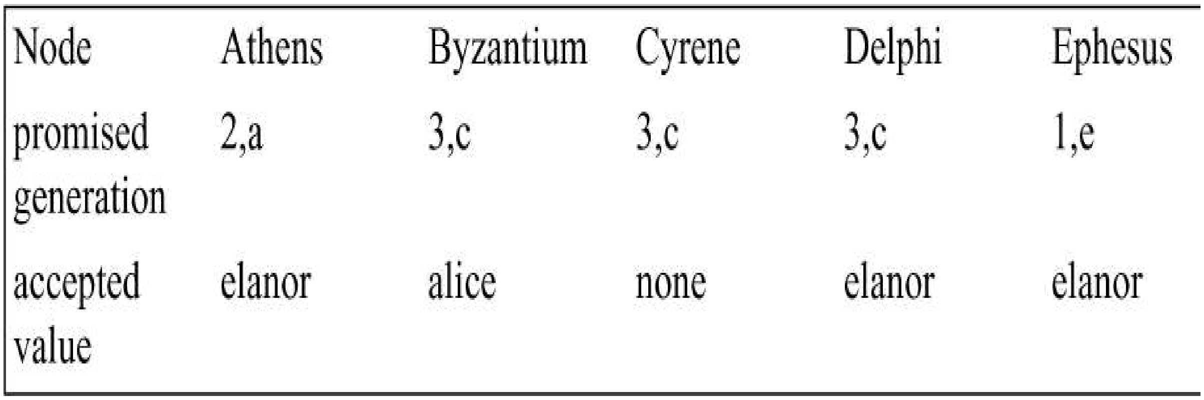
At this point Athens and Ephesus have now crashed. But the cluster still has a quorum of nodes operating, so they should be able to keep working, and indeed by following the protocol they can discover that "elanor" is the chosen value.



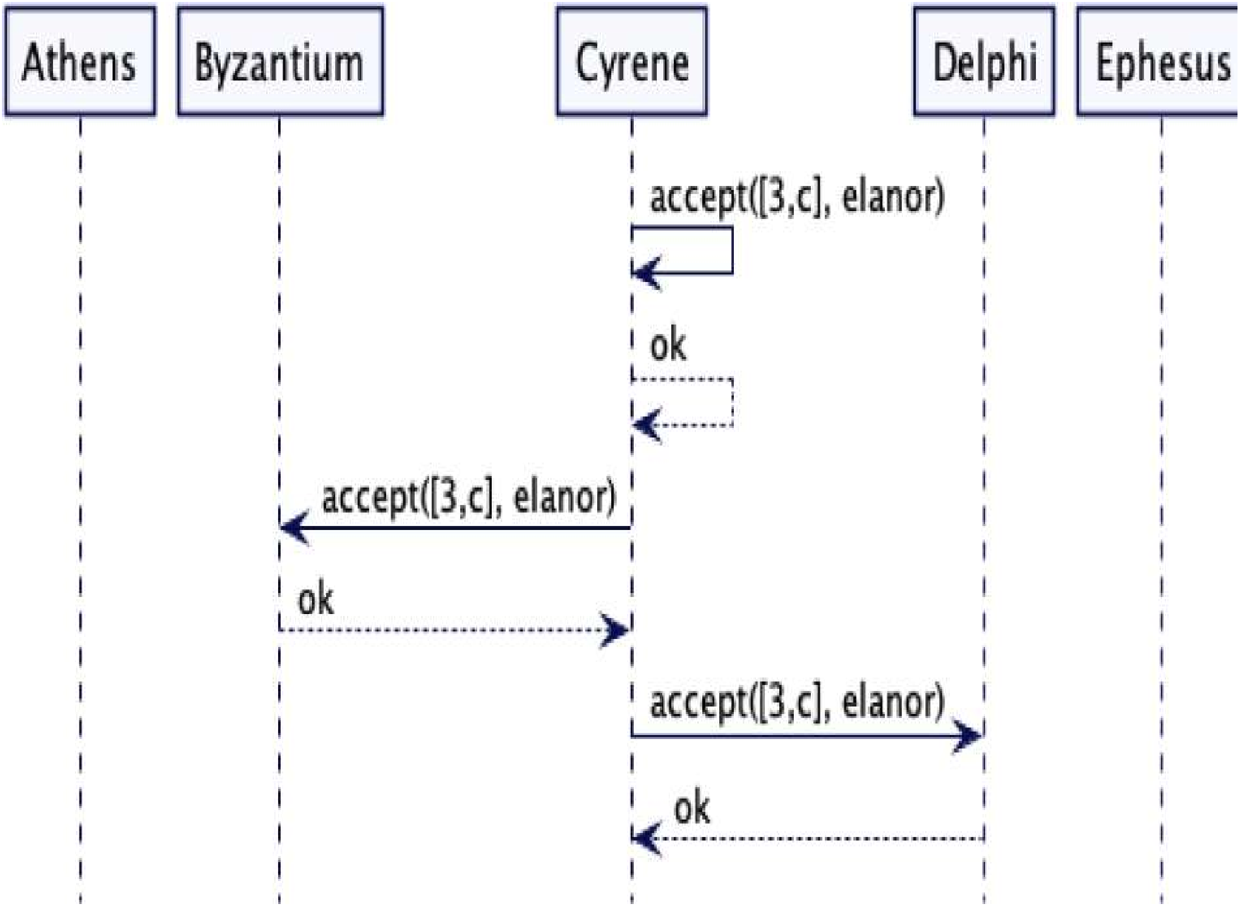
Cyrene gets a request to set the name to "carol", so it becomes a proposer. It's seen generation [2,a] so it kicks off a prepare phase with generation [3,c]. While it wishes to propose "carol" as the name, for the moment it's just issuing prepare requests.

Cyrene sends prepare messages to the remaining nodes in the cluster. As with Athens's earlier prepare phase, Cyrene gets accepted values back, so "carol" never gets proposed as a value. As before, Delphi's "elanor" is later than Byzantium's "alice", so Cyrene starts an accept phase with "elanor" and [3,c].

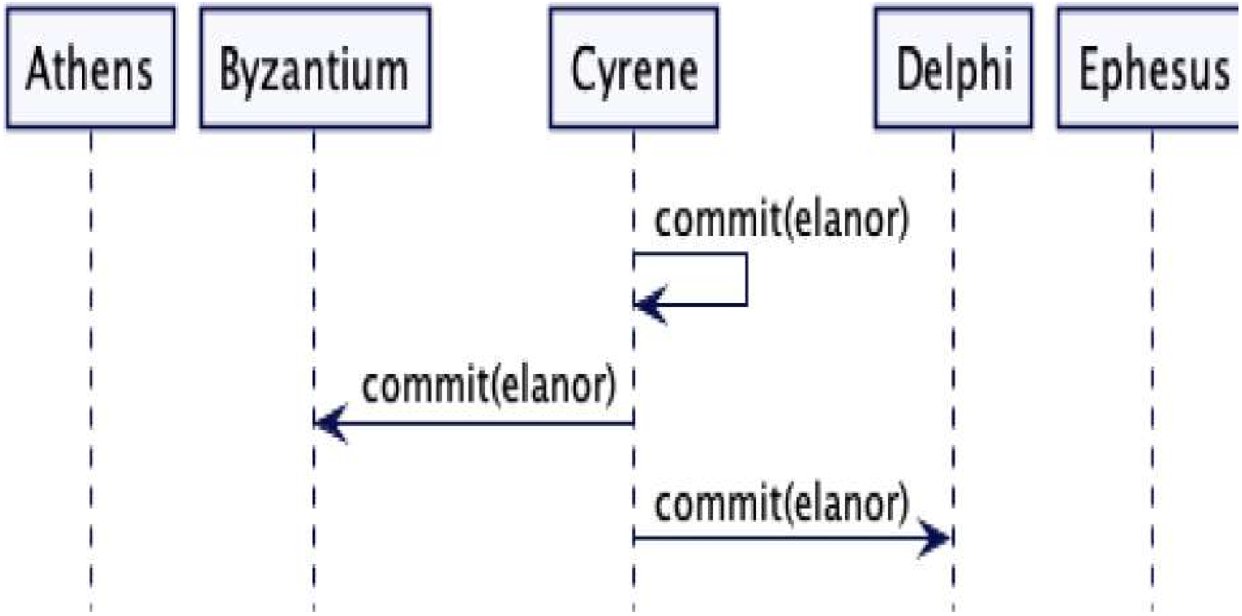
Table 8.9.



Node	Athens	Byzantium	Cyrene	Delphi	Ephesus
promised generation	2,a	3,c	3,c	3,c	1,e
accepted value	elanor	alice	none	elanor	elanor



While I could continue to crash and wake up nodes, it’s clear now that "elanor" will win out. As long as a quorum of nodes are up, at least one of them will have "elanor" as its value, and any node attempting a prepare will have to contact one node that’s accepted "elanor" in order to get a quorum for its prepare phase. So we’ll finish with Cyrene sending out commits.



At some point Athens and Ephesus will come back online and they will discover what the quorum has chosen.

Requests don’t need to be rejected

In the example above, we saw acceptors rejecting requests with an aged generation. But the protocol does not require an explicit rejection like this. As formulated, an acceptor may just ignore an out-of-date request. If this is the case, then the protocol will still converge on a single consensus value. This is an important feature of the protocol because, as this is a distributed system, connections can be lost at any time, so it’s beneficial to not be dependent on rejections to ensure the safety of the protocol. (*Safety* here meaning that the protocol will choose only one value, and once chosen, it won’t be overwritten.)

Sending rejections, however, is still useful as it improves performance. The quicker proposers find out they are old, the sooner they can start another round with a higher generation.

Competing proposers may fail to choose

One way this protocol can go wrong is if two (or more) proposers get into a cycle.

- alice is accepted by athens and byzantium

- elanor is prepared by all nodes, preventing alice from gaining quorum
- elanor is accepted by delphi and ephesus
- alice is prepared by all nodes, preventing elanor from gaining quorum.
- alice is accepted by athens and byzantium
- ... and so on, a situation called a livelock

The FLP Impossibility Result [bib-flp-impossibility] shows that even a single faulty node can stop a cluster from ever choosing a value.

We can reduce the chances of this livelock happening by ensuring that whenever a proposer needs to choose a new generation, it must wait a random period of time. This randomness makes it likely that one proposer will be able to get a quorum accepted before the other sends a prepare request to the full quorum.

But we can never ensure that livelock can't happen. This is a fundamental trade-off: we can either ensure safety or liveness, but not both. Paxos ensures safety first.

An example key-value store

The Paxos protocol explained here, builds consensus on a single value (often called as single-decree Paxos). Most practical implementations used in mainstream products like Cosmos DB [bib-cosmosdb] or Spanner [bib-spanner] use a modification of paxos called multi-paxos which is implemented as a *Replicated Log*.

But a simple key-value store can be built using basic Paxos. Cassandra [bib-cassandra] uses basic Paxos in a similar way to implement it's light-weight transactions.

The key-value store maintains Paxos instance per key.

class PaxosPerKeyStore…

```
int serverId;
public PaxosPerKeyStore(int serverId) {
  this.serverId = serverId;
}
```

```
Map<String, Acceptor> key2Acceptors = new HashMap<String, Acceptor>()
List<PaxosPerKeyStore> peers;
```

The Acceptor stores the promisedGeneration, acceptedGeneration and acceptedValue.

class Acceptor…

```
public class Acceptor {
  MonotonicId promisedGeneration = MonotonicId.empty();

  Optional<MonotonicId> acceptedGeneration = Optional.empty();
  Optional<Command> acceptedValue = Optional.empty();

  Optional<Command> committedValue = Optional.empty();
  Optional<MonotonicId> committedGeneration = Optional.empty();

  public AcceptorState state = AcceptorState.NEW;
  private BiConsumer<Acceptor, Command> kvStore;
```

When the key and value is put in the kv store, it runs the Paxos protocol.

class PaxosPerKeyStore…

```
int maxKnownPaxosRoundId = 1;
int maxAttempts = 4;
public void put(String key, String defaultProposal) {
  int attempts = 0;
  while(attempts <= maxAttempts) {
    attempts++;
    MonotonicId requestId = new MonotonicId(maxKnownPaxosRoundId++, s
    SetValueCommand setValueCommand = new SetValueCommand(key, defaul

    if (runPaxos(key, requestId, setValueCommand)) {
       return;
    }

    Uninterruptibles.sleepUninterruptibly(ThreadLocalRandom.current()
    logger.warn("Experienced Paxos contention. Attempting with higher
  }
  throw new WriteTimeoutException(attempts);
}

private boolean runPaxos(String key, MonotonicId generation, Command
  List<Acceptor> allAcceptors = getAcceptorInstancesFor(key);
  List<PrepareResponse> prepareResponses = sendPrepare(generation, al
  if (isQuorumPrepared(prepareResponses)) {
     Command proposedValue = getValue(prepareResponses, initialValue)
     if (sendAccept(generation, proposedValue, allAcceptors)) {
        sendCommit(generation, proposedValue, allAcceptors);
     }
     if (proposedValue == initialValue) {
        return true;
     }
   }
   return false;
 }

 public Command getValue(List<PrepareResponse> prepareResponses, Comm
   PrepareResponse mostRecentAcceptedValue = getMostRecentAcceptedVal
   Command proposedValue
      = mostRecentAcceptedValue.acceptedValue.isEmpty() ?
      initialValue : mostRecentAcceptedValue.acceptedValue.get();

   return proposedValue;
 }

 private PrepareResponse getMostRecentAcceptedValue(List<PrepareRespo
   return prepareResponses.stream().max(Comparator.comparing(r -> r.a
}
```

class Acceptor…

```
public PrepareResponse prepare(MonotonicId generation) {

  if (promisedGeneration.isAfter(generation)) {
     return new PrepareResponse(false, acceptedValue, acceptedGenerat
  }
  promisedGeneration = generation;
  state = AcceptorState.PROMISED;
  return new PrepareResponse(true, acceptedValue, acceptedGeneration,

}
```

class Acceptor…

```
public boolean accept(MonotonicId generation, Command value) {
  if (generation.equals(promisedGeneration) || generation.isAfter(pro
     this.promisedGeneration = generation;
     this.acceptedGeneration = Optional.of(generation);
     this.acceptedValue = Optional.of(value);
     return true;
  }
  state = AcceptorState.ACCEPTED;
  return false;
}
```

The value is stored in the kvstore only when it can be successfully committed.

class Acceptor…

```
public void commit(MonotonicId generation, Command value) {
  committedGeneration = Optional.of(generation);
  committedValue = Optional.of(value);
  state = AcceptorState.COMMITTED;
  kvStore.accept(this, value);
}
```

class PaxosPerKeyStore…

```
private void accept(Acceptor acceptor, Command command) {
  if (command instanceof SetValueCommand) {
     SetValueCommand setValueCommand = (SetValueCommand) command;
     kv.put(setValueCommand.getKey(), setValueCommand.getValue());
  }
  acceptor.resetPaxosState();
}
```

The paxos state needs to be persisted. It can be easily done by using a *Write-Ahead Log*.

Handling multiple values.

It is important to note that Paxos is specified and proven to work on single value. So handling multiple values with the single value Paxos protocol needs to be done outside of the protocol specification. One alternative is to reset the state, and store committed values separately to make sure they are not lost.

class Acceptor…

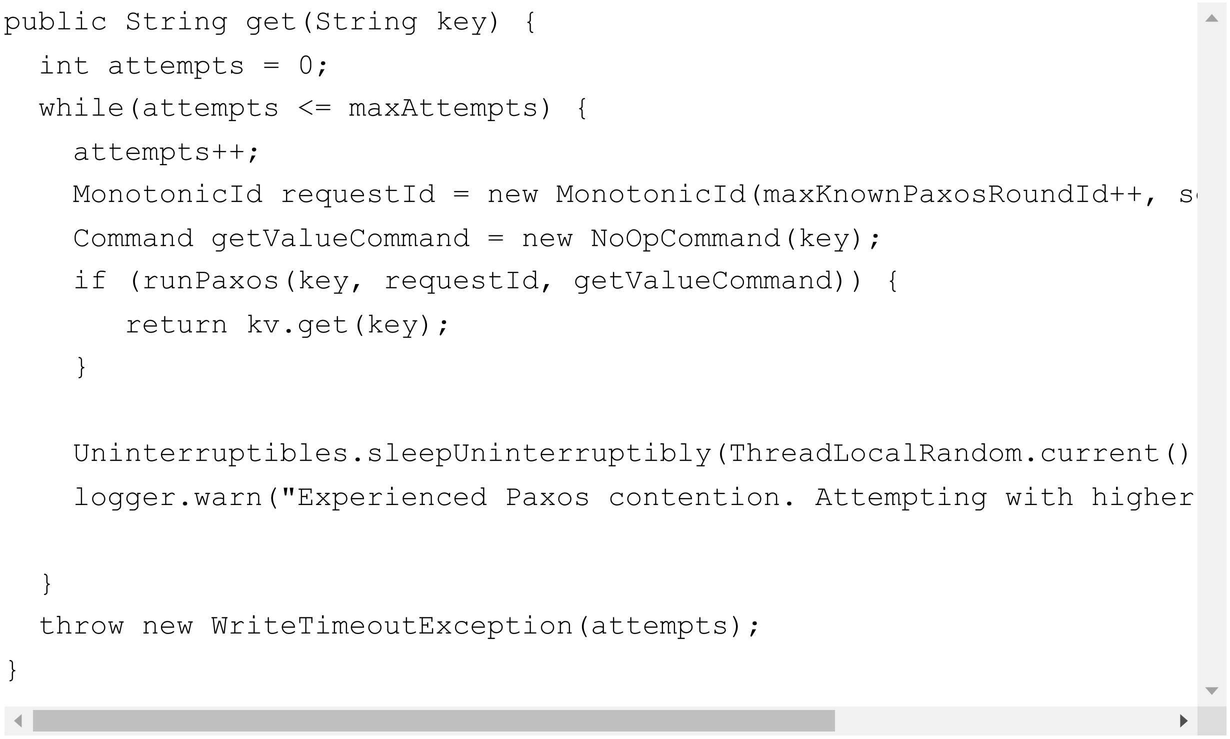
```
public void resetPaxosState() {
  //This implementation has issues if committed values are not stored
  //and handled separately in the prepare phase.
  //See Cassandra implementation for details.
  //https://github.com/apache/cassandra/blob/trunk/src/java/org/apach
  promisedGeneration = MonotonicId.empty();
  acceptedGeneration = Optional.empty();
  acceptedValue = Optional.empty();
}
```

There is an alternative, as suggested in [gryadka] [bib-gryadka], which slightly modifies the basic Paxos to allow setting multiple values. This need for executing steps beyond the basic algorithm is the reason that in practice *Replicated Log* is preferred.

Reading the values

Paxos relies on the prepare phase to detect any uncommitted values. So if basic Paxos is used to implement a key-value store as shown above, the read operation also needs to run the full Paxos algorithm.

class PaxosPerKeyStore…



```
public String get(String key) {
  int attempts = 0;
  while(attempts <= maxAttempts) {
    attempts++;
    MonotonicId requestId = new MonotonicId(maxKnownPaxosRoundId++, s
    Command getValueCommand = new NoOpCommand(key);
    if (runPaxos(key, requestId, getValueCommand)) {
       return kv.get(key);
    }

    Uninterruptibles.sleepUninterruptibly(ThreadLocalRandom.current()
    logger.warn("Experienced Paxos contention. Attempting with higher

  }
  throw new WriteTimeoutException(attempts);
}
```

Examples

Cassandra [bib-cassandra] uses Paxos to implement light-weight transaction.

All the consensus algorithms like Raft [bib-raft] use basic concepts similar to the basic Paxos. The use of *Two Phase Commit*, *Quorum* and *Generation Clock* is used in a similar manner.

Chapter 9. Replicated Log

Keep the state of multiple nodes synchronized by using a write-ahead log that is replicated to all the cluster nodes.

Problem

When multiple nodes share a state, the state needs to be synchronized. All cluster nodes need to agree on the same state, even when some nodes crash or get disconnected. This requires achieving consensus for each state change request.

But achieving consensus on individual requests is not enough. Each replica also needs to execute requests in the same order, otherwise different replicas can get into a different final state, even if they have consensus on an individual request.

Solution

Failure Assumptions

Different algorithms are used to build consensus over log entries depending on the failure assumptions. The most commonly used assumption is that of crash fault [bib-crash-fault]. With crash fault, when a cluster node is faulty, it stops working. A more complex failure assumption is that of byzantine fault [bib-byzantine-fault]. With byzantine faults, faulty cluster nodes can behave arbitrarily. They might be controlled by an adversary which keeps the node functional but

> deliberately sends requests or responses with wrong data, such as a fraudulent transaction to steal money.
>
> Most enterprise systems such as databases, message brokers or even enterprise blockchain products such as hyperledger fabric [bib-hyperledger-fabric] assume crash faults. So consensus algorithms like Raft [bib-raft] and *Paxos*, which are built with crash fault assumptions, are almost always used.
>
> Algorithms like pbft [bib-pbft] are used for systems which need to allow byzantine failures. While the pbft algorithm uses log in a similar way, to tolerate byzantine failure, it needs three phased execution and a quorum of $3f + 1$, where f is the number of tolerated failures.

Cluster nodes maintain a *Write-Ahead Log*. Each log entry stores the state required for consensus along with the user request. They coordinate to build consensus over log entries, so that all cluster nodes have exactly the same Write-Ahead log. The requests are then executed sequentially following the log. Because all cluster nodes agree on each log entry, they execute the same requests in the same order. This ensures that all the cluster nodes share the same state.

A fault tolerant consensus building mechanism using *Quorum* needs two phases.

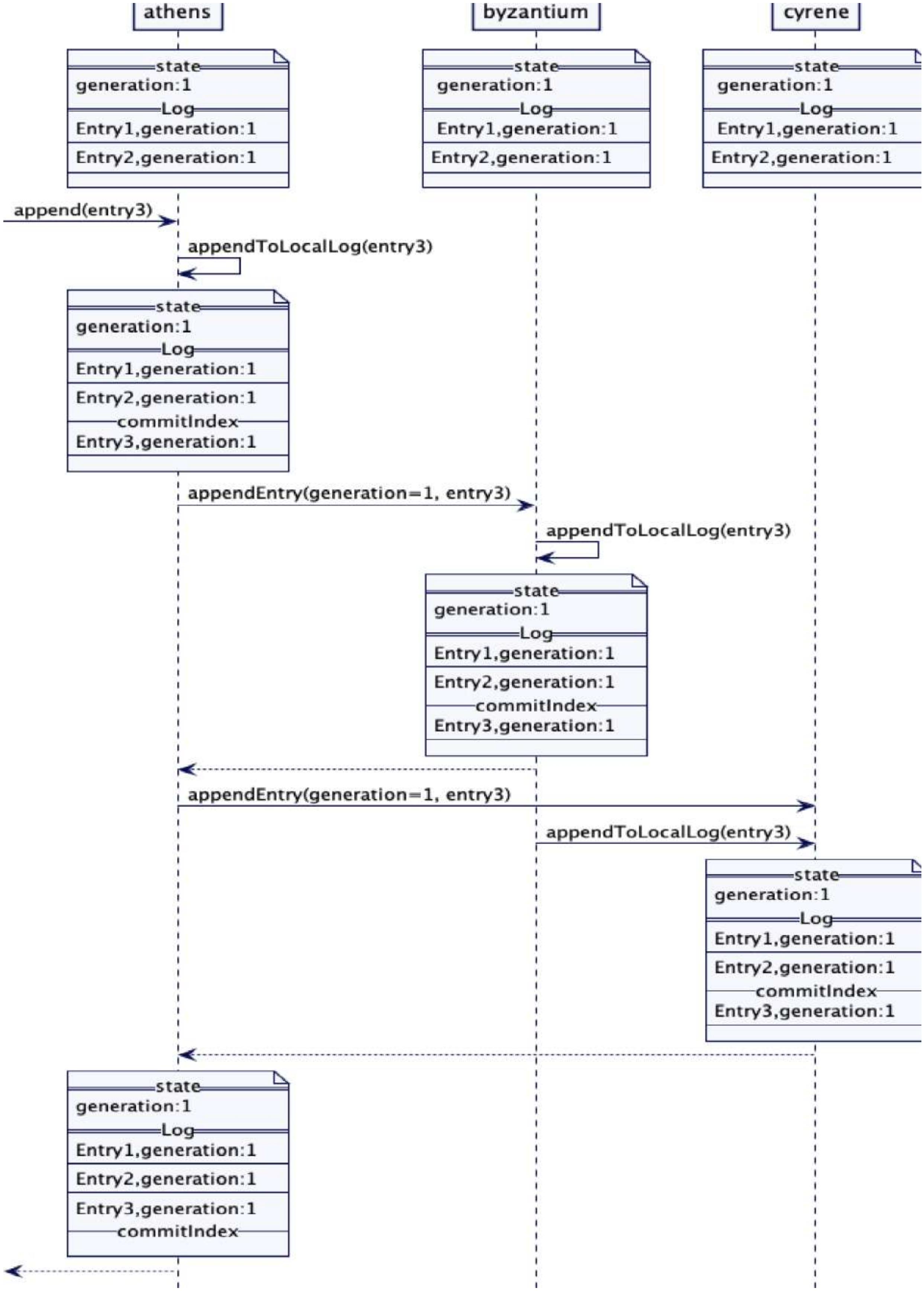
- A phase to establish a *Generation Clock* and to know about the log entries replicated in the previous Quorum.
- A phase to replicate requests on all the cluster nodes.

Executing two phases for each state change request is not efficient. So cluster nodes select a leader at startup. The leader election phase establishes the Generation Clock and detects all log entries in the previous Quorum. (The entries the previous leader might have copied to the majority of the cluster nodes.) Once there is a stable leader, only the leader co-ordinates the replication. Clients communicate with the leader. The leader adds each request to the log and makes sure that it’s replicated on all the followers. Consensus is reached once a log entry is successfully replicated to the majority of the followers. This way, only one phase execution to reach consensus is needed for each state change operation when there is a stable leader.

Multi-Paxos and Raft

Multi-Paxos [bib-multi-paxos] and Raft [bib-raft] are the most popular algorithms to implement replicated-log. Multi-Paxos is only vaguely described in academic papers. Cloud databases such as Spanner [bib-spanner] and Cosmos DB [bib-cosmosdb] use Multi-Paxos [bib-multi-paxos], but the implementation details are not well documented. Raft very clearly documents all the implementation details, and is a preferred implementation choice in most open source systems, even though Paxos and its variants are discussed a lot more in academia.

Following sections describe how Raft implements a replicated log.


athens
byzantium
cyrene
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
append(entry3)
appendToLocalLog(entry3)
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
commitIndex
Entry3,generation:1
appendEntry(generation=1, entry3)
appendToLocalLog(entry3)
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
commitIndex
Entry3,generation:1
appendEntry(generation=1, entry3)
appendToLocalLog(entry3)
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
commitIndex
Entry3,generation:1
state
generation:1
Log
Entry1,generation:1
Entry2,generation:1
Entry3,generation:1
commitIndex

For each log entry, the leader appends it to its local Write-Ahead log and then sends it to all the followers.

leader (class ReplicatedLog...)

```
private Long appendAndReplicate(byte[] data) {
   Long lastLogEntryIndex = appendToLocalLog(data);
   replicateOnFollowers(lastLogEntryIndex);
   return lastLogEntryIndex;
}



private void replicateOnFollowers(Long entryAtIndex) {
   //FIXME: factorout as a separate method.
   oldLeaderLeaseRemainingTime = System.nanoTime() + leaderLeaseTim

   for (final FollowerHandler follower : followers) {
     replicateOn(follower, entryAtIndex); //send replication reques
   }
}
```

The followers handle the replication request and append the log entries to their local logs. After successfully appending the log entries, they respond to the leader with the index of the latest log entry they have. The response also includes the current *Generation Clock* of the server.

The followers also check if the entries already exist or there are entries beyond the ones which are being replicated. It ignores entries which are already present. But if there are entries which are from different generations, they remove the conflicting entries.

follower (class ReplicatedLog...)

```
void maybeTruncate(ReplicationRequest replicationRequest) {
  replicationRequest.getEntries().stream()
       .filter(entry -> wal.getLastLogIndex() >= entry.getEntryInde
              entry.getGeneration() != wal.readAt(entry.getEntryInd
       .forEach(entry -> wal.truncate(entry.getEntryIndex()));
}
```

follower (class ReplicatedLog...)

```
private ReplicationResponse appendEntries(ReplicationRequest replic
   List<WALEntry> entries = replicationRequest.getEntries();
   entries.stream()
        .filter(e -> !wal.exists(e))
        .forEach(e -> wal.writeEntry(e));
   return new ReplicationResponse(SUCCEEDED, serverId(), replicatio
}
```

The follower rejects the replication request when the generation number in the request is lower than the latest generation the server knows about. This notifies the leader to step down and become a follower.

follower (class ReplicatedLog...)

```
Long currentGeneration = replicationState.getGeneration();
if (currentGeneration > request.getGeneration()) {
   return new ReplicationResponse(FAILED, serverId(), currentGenera
}
```

The Leader keeps track of log indexes replicated at each server, when responses are received. It uses it to track the log entries which are successfully copied to the *Quorum* and tracks the index as a commitIndex. commitIndex is the *High-Water Mark* in the log.

leader (class ReplicatedLog...)

```
logger.info("Updating matchIndex for " + response.getServerId() + "
updateMatchingLogIndex(response.getServerId(), response.getReplicat
var logIndexAtQuorum = computeHighwaterMark(logIndexesAtAllServers(
var currentHighWaterMark = replicationState.getHighWaterMark();
if (logIndexAtQuorum > currentHighWaterMark && logIndexAtQuorum !=
   applyLogEntries(currentHighWaterMark, logIndexAtQuorum);
   replicationState.setHighWaterMark(logIndexAtQuorum);
}
```

leader (class ReplicatedLog...)

```
Long computeHighwaterMark(List<Long> serverLogIndexes, int noOfServ
  serverLogIndexes.sort(Long::compareTo);
  return serverLogIndexes.get(noOfServers / 2);
}
```

leader (class ReplicatedLog...)

```
private void updateMatchingLogIndex(int serverId, long replicatedLo
   FollowerHandler follower = getFollowerHandler(serverId);
   follower.updateLastReplicationIndex(replicatedLogIndex);
}
```

leader (class ReplicatedLog...)

```
public void updateLastReplicationIndex(long lastReplicatedLogIndex)
  this.matchIndex = lastReplicatedLogIndex;
}
```

Full replication

It is important to ensure that all the cluster nodes receive all the log entries from the leader, even when they are disconnected or they crash and come back up. Raft has a mechanism to make sure all the cluster nodes receive all the log entries from the leader.

With every replication request in Raft, the leader also sends the log index and generation of the log entries which immediately precede the new entries getting replicated. If the previous log index and term do not match with its local log, the followers reject the request. This indicates to the leader that the follower log needs to be synced for some of the older entries.

follower (class ReplicatedLog...)

```
if (!wal.isEmpty() && request.getPrevLogIndex() >= wal.getLogStartI
      generationAt(request.getPrevLogIndex()) != request.getPrevLog
   return new ReplicationResponse(FAILED, serverId(), replicationSt
}
```

follower (class ReplicatedLog...)

```
private Long generationAt(long prevLogIndex) {
   WALEntry walEntry = wal.readAt(prevLogIndex);

   return walEntry.getGeneration();
}
```

So the leader decrements the matchIndex and tries sending log entries at the lower index. This continues until the followers accept the replication request.

leader (class ReplicatedLog...)

```
//rejected because of conflicting entries, decrement matchIndex
FollowerHandler peer = getFollowerHandler(response.getServerId());
logger.info("decrementing nextIndex for peer " + peer.getId() + " f
peer.decrementNextIndex();
replicateOn(peer, peer.getNextIndex());
```

This check on the previous log index and generation allows the leader to detect two things.

- If the follower log has missing entries. For example, if the follower log has only one entry and the leader starts replicating the third entry, the requests will be rejected until the leader replicates the second entry.
- If the previous entries in the log are from a different generation, higher or lower than the corresponding entries in the leader log. The leader will

try replicating entries from lower indexes until the requests get accepted. The followers truncate the entries for which the generation does not match.

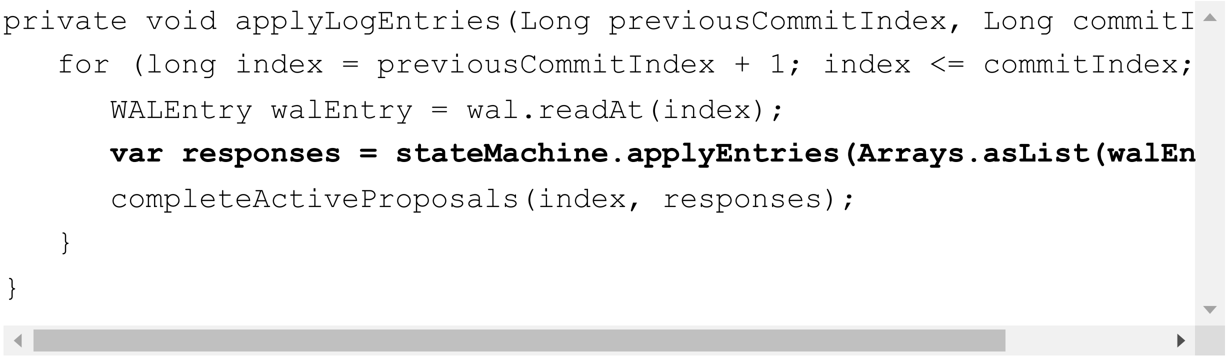
This way, the leader tries to push its own log to all the followers continuously by using the previous index to detect missing entries or conflicting entries. This makes sure that all the cluster nodes eventually receive all the log entries from the leader even when they are disconnected for some time.

Raft does not have a separate commit message, but sends the commitIndex as part of the normal replication requests. The empty replication requests are also sent as heartbeats. So commitIndex is sent to followers as part of the heartbeat requests.

Log entries are executed in the log order

Once the leader updates its commitIndex, it executes the log entries in order, from the last value of the commitIndex to the latest value of the commitIndex. The client requests are completed and the response is returned to the client once the log entries are executed.

class ReplicatedLog…



```
private void applyLogEntries(Long previousCommitIndex, Long commitI
   for (long index = previousCommitIndex + 1; index <= commitIndex;
      WALEntry walEntry = wal.readAt(index);
      var responses = stateMachine.applyEntries(Arrays.asList(walEn
      completeActiveProposals(index, responses);
   }
}
```

The leader also sends the commitIndex with the heartbeat requests it sends to the followers. The followers update the commitIndex and apply the entries the same way.

class ReplicatedLog…

```
private void updateHighWaterMark(ReplicationRequest request) {
   if (request.getHighWaterMark() > replicationState.getHighWaterMa
      var previousHighWaterMark = replicationState.getHighWaterMark
      replicationState.setHighWaterMark(request.getHighWaterMark())
      applyLogEntries(previousHighWaterMark, request.getHighWaterMa
   }
}
```

Leader Election

> It is possible that multiple cluster nodes start leader election at the same time. To reduce the possibility of this happening, each cluster node waits for a random amount of time before triggering the election.So mostly only one cluster node starts the election and wins it.
>
> Leader election is also a problem which needs all of the cluster nodes to reach an agreement. The approach taken by Raft [bib-raft] and other consensus algorithms is to allow not having an agreement in the worst case. The consistency is preferred over availability in such cases. This incident at Cloudflare [bib-cloudflare-outage] is a good example of that. The stale leaders are also tolerated. In such cases, *Generation Clock* makes sure that only one leader succeeds in getting its requests accepted by the followers.

Leader election is the phase where log entries committed in the previous quorum are detected. Every cluster node operates in three states: candidate, leader or follower. The cluster nodes start in a follower state expecting a *HeartBeat* from an existing leader. If a follower doesn’t hear from any leader in a predetermined time period ,it moves to the candidate state and starts leader-election. The leader election algorithm establishes a new Generation Clock value. Raft refers to the Generation Clock as *term*.

The leader election mechanism also makes sure the elected leader has as many up-to-date log entries stipulated by the quorum. This is an optimization done by Raft [bib-raft] which avoids log entries from previous *Quorum* being transferred to the new leader.

New leader election is started by sending each of the peer servers a message requesting a vote.

class ReplicatedLog…

```
private void startLeaderElection() {
   replicationState.setGeneration(replicationState.getGeneration()
   registerSelfVote();
   requestVoteFrom(followers);
}
```

Once a server is voted for in a given Generation Clock, the same vote is returned for that generation always. This ensures that some other server requesting a vote for the same generation is not elected, when a successful election has already happened. The handling of the vote request happens as follows:

class ReplicatedLog…

```
VoteResponse handleVoteRequest(VoteRequest voteRequest) {
  //for higher generation request become follower.
  // But we do not know who the leader is yet.
  if (voteRequest.getGeneration() > replicationState.getGeneration(
     becomeFollower(LEADER_NOT_KNOWN, voteRequest.getGeneration());
  }

  VoteTracker voteTracker = replicationState.getVoteTracker();
  if (voteRequest.getGeneration() == replicationState.getGeneration
     if (isUptoDate(voteRequest) && !voteTracker.alreadyVoted()) {
        voteTracker.registerVote(voteRequest.getServerId());
        return grantVote();
     }
     if (voteTracker.alreadyVoted()) {
        return voteTracker.votedFor == voteRequest.getServerId() ?
             grantVote() : rejectVote();
     }
   }
   return rejectVote();
 }

 private boolean isUptoDate(VoteRequest voteRequest) {
 boolean result = voteRequest.getLastLogEntryGeneration() > wal.get
     || (voteRequest.getLastLogEntryGeneration() == wal.getLastLogE
     voteRequest.getLastLogEntryIndex() >= wal.getLastLogIndex());
 return result;
}
```

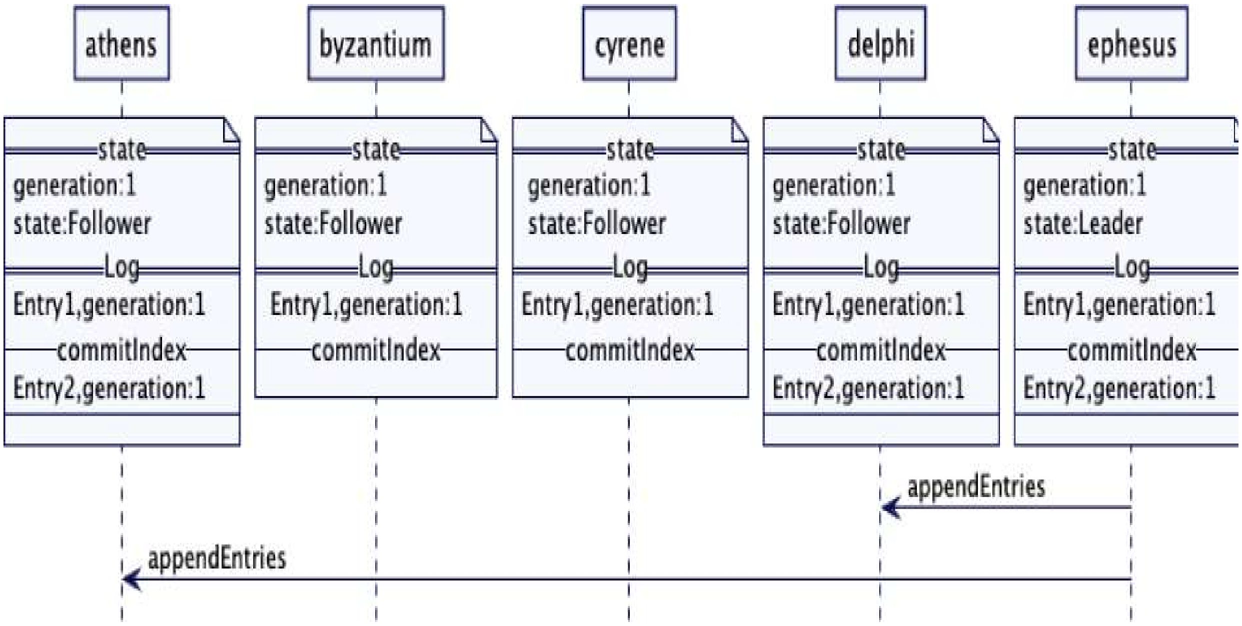
The server which receives votes from the majority of the servers transitions to the leader state. The majority is determined as discussed in Quorum. Once elected, the leader continuously sends a HeartBeat to all of the followers. If the followers don’t receive a HeartBeat in a specified time interval, a new leader election is triggered.

Log entries from previous generation

As discussed in the above section, the first phase of the consensus algorithms detects the existing values, which had been copied on the previous runs of the algorithm. The other key aspect is that these values are proposed as the values with the latest generation of the leader. The second phase decides that the value is committed only if the values are proposed for the current generation. Raft never updates generation numbers for the existing entries in the log. So if the leader has log entries from the older generation which are missing from some of the followers, it can not mark those entries as committed just based on the majority quorum. That is because some other server which may not be available now, can have an entry at the same index with higher generation. If the leader goes down without replicating an entry from its current generation, those entries can get overwritten by the new leader. So in Raft, the new leader must commit at least one entry in its term. It can then safely commit all the previous entries. Most practical implementations of Raft try to commit a no-op entry immediately after a leader election, before the leader is considered ready to serve client requests. Refer to [raft-phd] [bib-raft-phd] section 3.6.1 for details.

An example leader-election

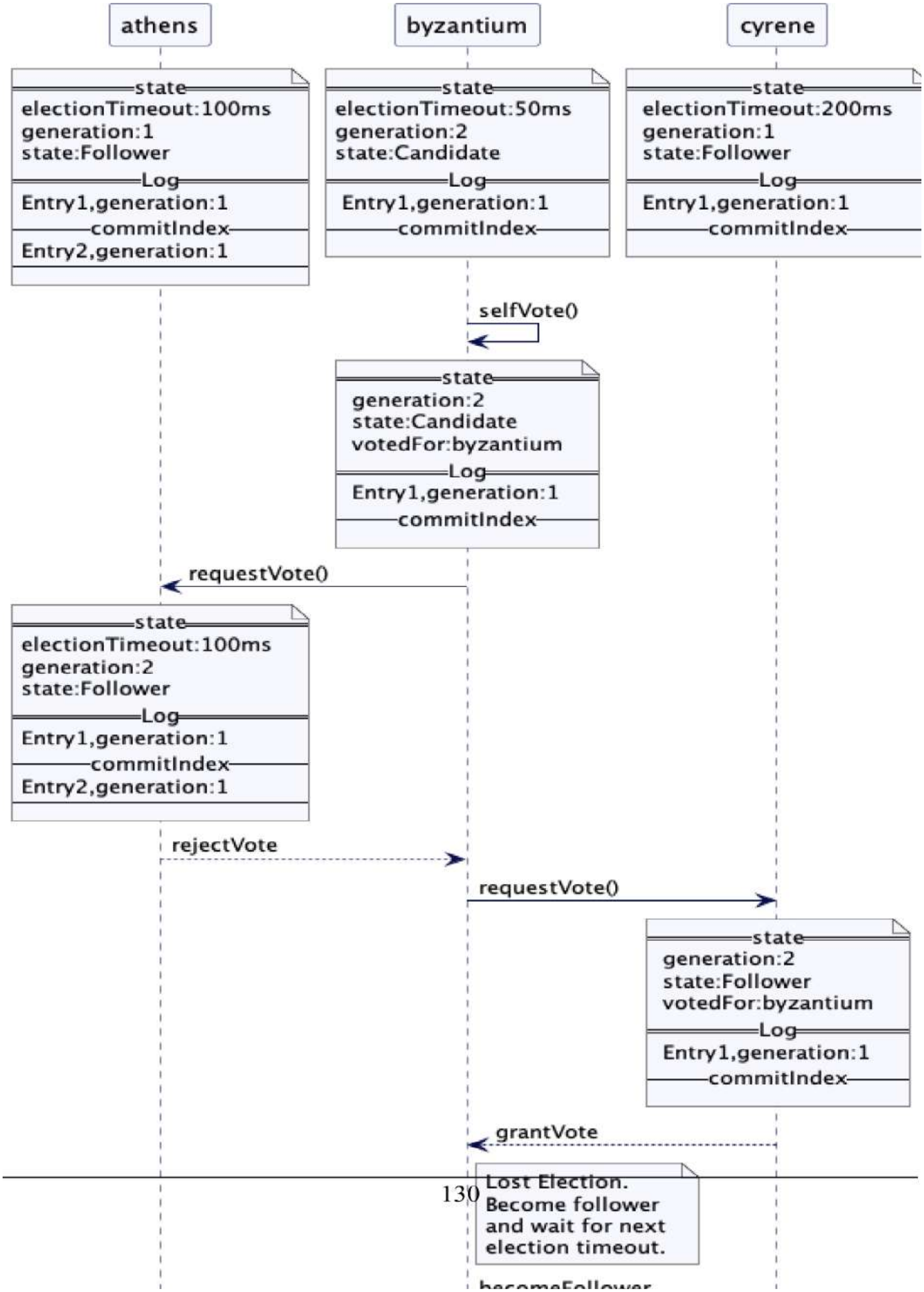
Consider five servers, athens, byzantium, cyrene, delphi and ephesus. ephesus is the leader for generation 1. It has replicated entries to itself, delphi and athens.



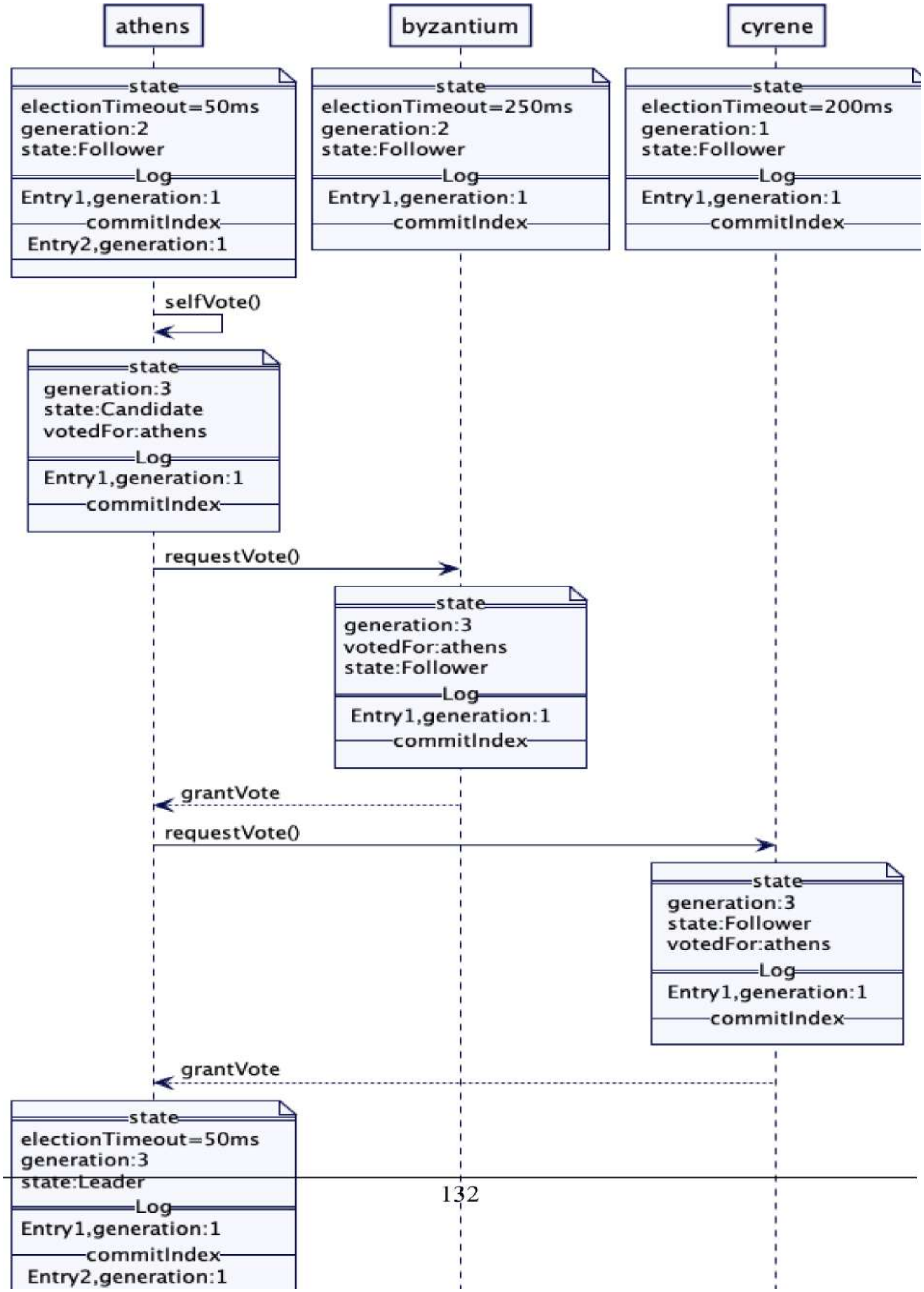
At this point, ephesus and delphi get disconnected from the rest of the cluster.

byzantium has the least election timeout, so it triggers the election by incrementing its *Generation Clock* to 2. cyrene has its generation less than 2 and it also has same log entry as byzantium. So it grants the vote. But athens has an extra entry in its log. So it rejects the vote.

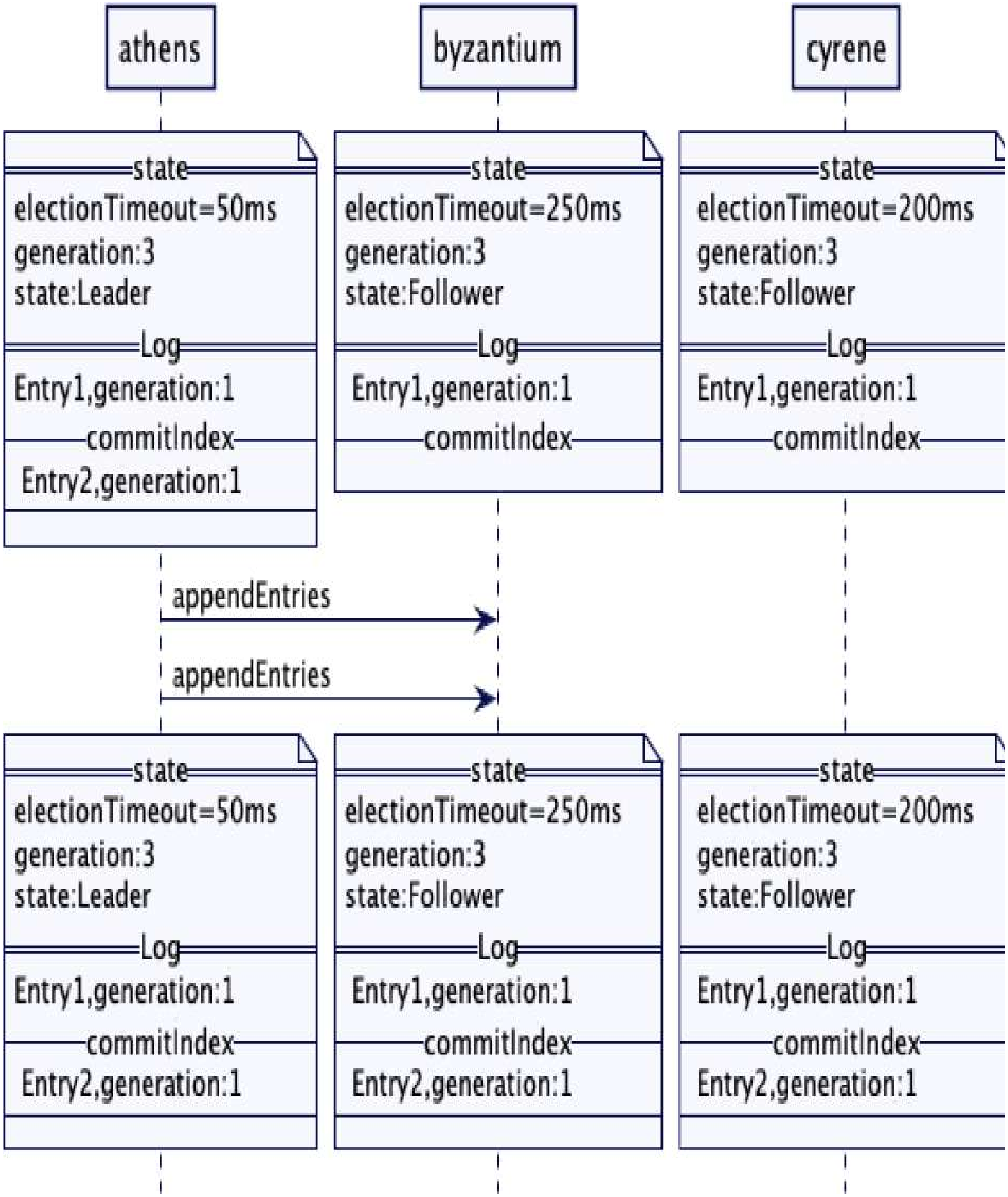
Because byzantium can not get a majority 3 votes, it loses the election and moves back to follower state.


athens
byzantium
cyrene
state
electionTimeout:100ms
generation:1
state:Follower
Log
Entry1,generation:1
commitIndex
Entry2,generation:1
state
electionTimeout:50ms
generation:2
state:Candidate
Log
Entry1,generation:1
commitIndex
state
electionTimeout:200ms
generation:1
state:Follower
Log
Entry1,generation:1
commitIndex
selfVote()
state
generation:2
state:Candidate
votedFor:byzantium
Log
Entry1,generation:1
commitIndex
requestVote()
state
electionTimeout:100ms
generation:2
state:Follower
Log
Entry1,generation:1
commitIndex
Entry2,generation:1
rejectVote
requestVote()
state
generation:2
state:Follower
votedFor:byzantium
Log
Entry1,generation:1
commitIndex
grantVote
Lost Election.
Become follower
and wait for next
election timeout.
becomeFollower

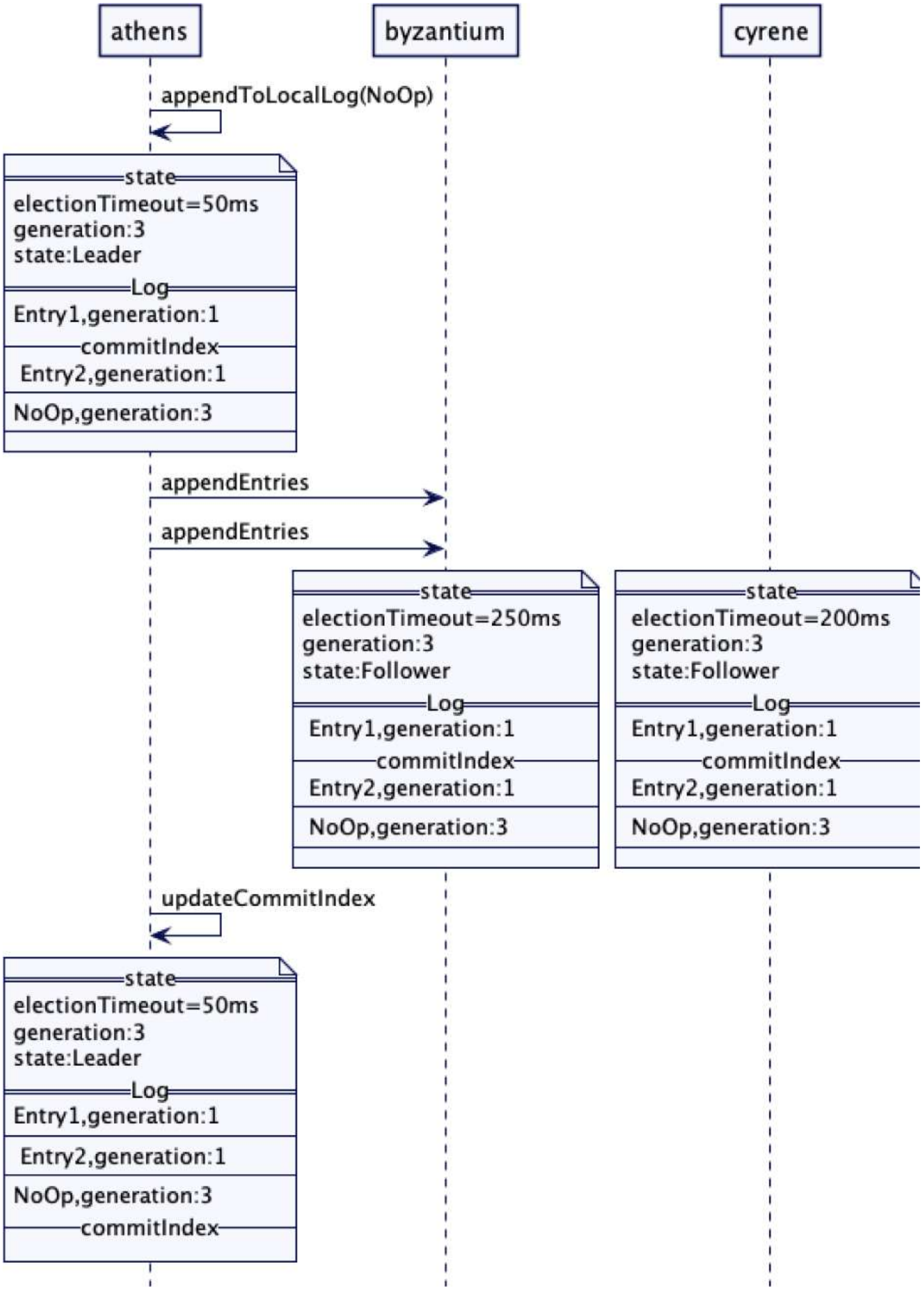
athens times out and triggers the election next. It increments the Generation Clock to 3 and sends vote request to byzantium and cyrene. Because both byzantium and cyrene have lower generation number and less log entries than athens, they both grant the vote to athens. Once athens gets majority of the votes, it becomes the leader and starts sending HeartBeats to byzantium and cyrene. Once byzantium and cyrene receive a heartbeat from the leader at higher generation, they increment their generation. This confirms the leadership of athens. athens then replicates its own log to byzantium and cyrene.


athens
byzantium
cyrene
state
electionTimeout=50ms
generation:2
state:Follower
Log
Entry1,generation:1
commitIndex
Entry2,generation:1
state
electionTimeout=250ms
generation:2
state:Follower
Log
Entry1,generation:1
commitIndex
state
electionTimeout=200ms
generation:1
state:Follower
Log
Entry1,generation:1
commitIndex
selfVote()
state
generation:3
state:Candidate
votedFor:athens
Log
Entry1,generation:1
commitIndex
requestVote()
state
generation:3
votedFor:athens
state:Follower
Log
Entry1,generation:1
commitIndex
grantVote
requestVote()
state
generation:3
state:Follower
votedFor:athens
Log
Entry1,generation:1
commitIndex
grantVote
state
electionTimeout=50ms
generation:3
state:Leader
Log
Entry1,generation:1
commitIndex
Entry2,generation:1

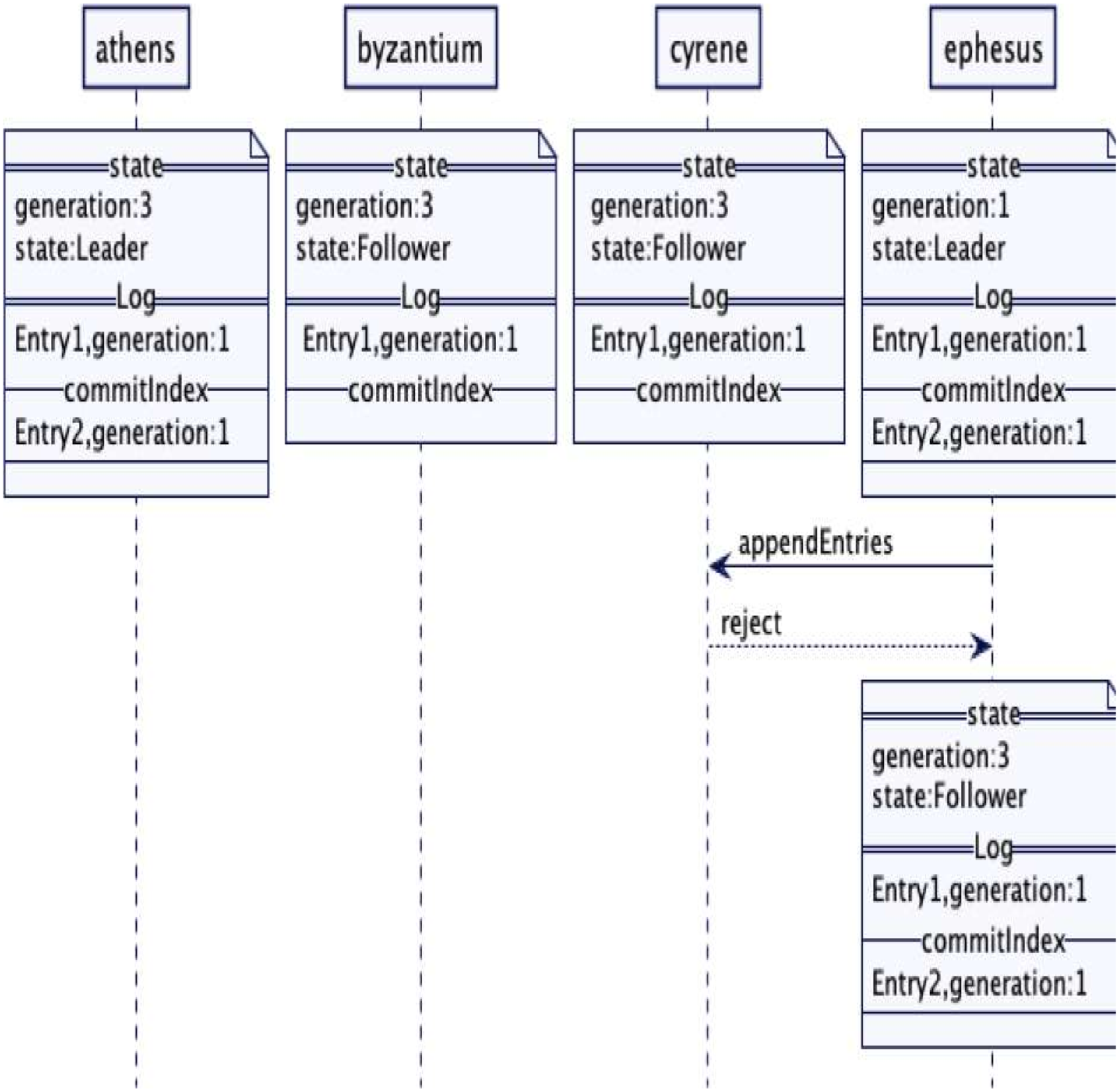
athens now replicates Entry2 from generation 1 to byzantium and cyrene. But because it’s an entry from the previous generation, it does not update the commitIndex even when Entry2 is successfully replicated on the majority quorum.



athens appends a no-op entry to its local log. After this new entry in generation 3 is successfully replicated, it updates the commitIndex


athens
byzantium
cyrene
appendToLocalLog(NoOp)
state
electionTimeout=50ms
generation:3
state:Leader
Log
Entry1,generation:1
commitIndex
Entry2,generation:1
NoOp,generation:3
appendEntries
appendEntries
state
electionTimeout=250ms
generation:3
state:Follower
Log
Entry1,generation:1
commitIndex
Entry2,generation:1
NoOp,generation:3
state
electionTimeout=200ms
generation:3
state:Follower
Log
Entry1,generation:1
commitIndex
Entry2,generation:1
NoOp,generation:3
updateCommitIndex
state
electionTimeout=50ms
generation:3
state:Leader
Log
Entry1,generation:1
Entry2,generation:1
NoOp,generation:3
commitIndex

If ephesus comes back up or restores network connectivity and sends request to cyrene. Because cyrene is now at generation 3, it rejects the requests. ephesus gets the new term in the rejection response, and steps down to be a follower.



Technical Considerations

Following are some of the important technical considerations for any replicated log mechanism.

- The first phase of any consensus building mechanism needs to know about the log entries which might be replicated on the previous *Quorum*. The leader needs to know about all such log entries and make sure that they are replicated on each cluster node.

  Raft makes sure that the cluster node with as up-to-date a log as the quorum of the servers is elected as a leader, so log entries do not need to be passed from other cluster nodes to the new leader.

  It is possible that some entries are conflicting. In this case, the conflicting entries from the follower log are overwritten.

- It is possible that some cluster nodes in the cluster lag behind, either because they crash and restart or get disconnected from the leader. The leader needs to track each cluster node and make sure that it sends all the missing entries.

  Raft maintains a state per cluster node to know the log index to which the log entries are successfully copied to each node. The replication requests to each cluster node are sent with all the entries from that log index, making sure that each cluster node gets all the log entries.

- How the client interacts with the replicated log to find the leader is implemented is discussed in the *Consistent Core*. The cluster detecting duplicate requests in case of client retries is handled by the *Idempotent Receiver*.

- The logs are generally compacted by using *Low-Water Mark*. A snapshot of the store backed by replicated log is taken periodically, say after a few thousand entries are applied. The log is then discarded to the index at which the snapshot is taken. The slow followers or the newly added servers, which need the full log to be sent, are sent the snapshot instead of individual log entries.

- One of the key assumptions here is that all the requests need to be strictly ordered. This might not be the requirement always. For example a key value store might not require ordering across requests for different keys. In such situations, it is possible to run a different consensus instance per key. It then also removes the need to have a single leader for all the requests.

EPaxos [bib-epaxos] is an algorithm which does not rely on a single leader for ordering of requests.

In partitioned databases like MongoDB [bib-mongodb], a replicated log is maintained per partition. So requests are ordered per partition, but not across partitions.

Push vs Pull

In the Raft [bib-raft] replication mechanism explained here, the leader pushes all the log entries to followers. It is also possible to have followers pull the log entries. The Raft implementation [bib-kafka-raft] in Kafka [bib-kafka] follows pull-based replication.

What goes in the log?

The replicated log mechanism is used for a wide variety of applications, from a key-value store to blockchain [bib-blockchain].

For a key-value store, the entries in the log are about setting key-values. For a *Lease* the entries are about setting up named leases. For a blockchain, the entries are the blocks in a blockchain which need to be served to all the peers in the same order. For databases like MongoDB [bib-mongodb], the entries are the data that needs to be consistently replicated.

More generally, the requests which make state changes are placed in the log.

Bypassing the log for read requests

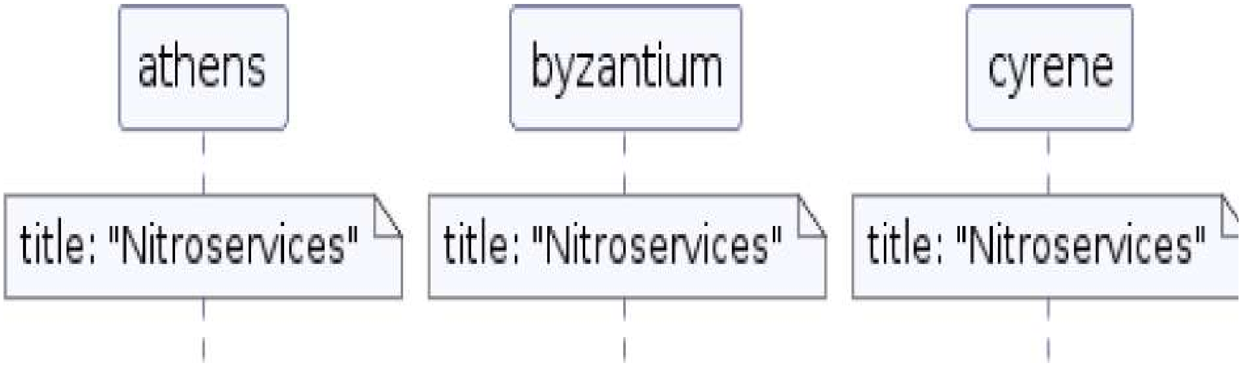
Replicated Log generally acts as a *Write-Ahead Log* for a data store. Datastores are expected to handle lot more read requests than write requests. Read requests are more latency sensitive. So most replicated-log implementations like etcd [bib-etcd] which uses Raft [bib-raft] or Zookeeper [bib-zookeeper] serve read requests directly from the key-value stores without going through the replicated-log, thus, avoid log replication overhead.

One of the key issues because of this is that the read requests can return older values if the leader is disconnected from the rest of the cluster. With the standard log replication, the leader won't be able to execute any request which are put in the log unilatery. This provides the safety because of following things.

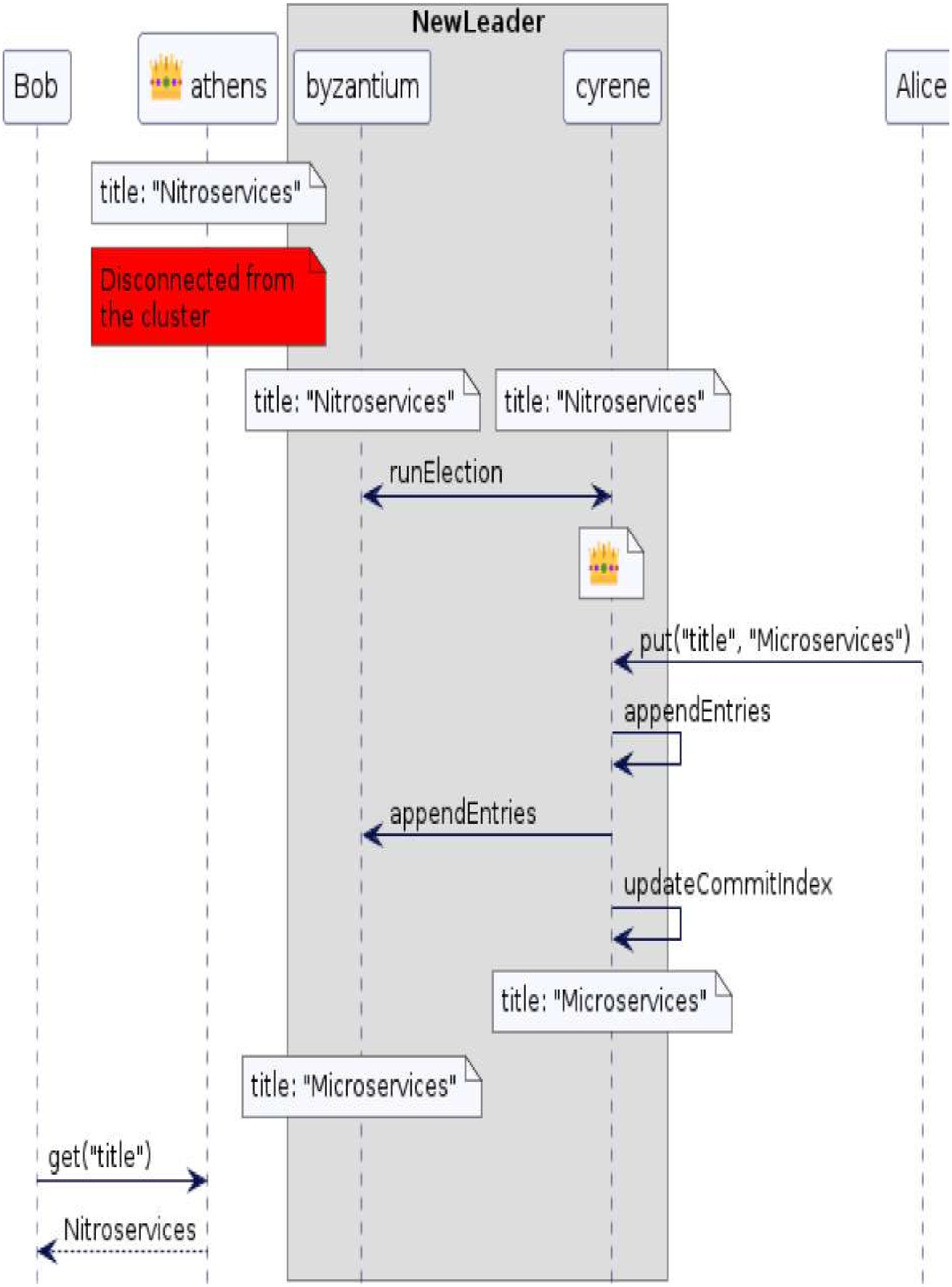
- Leader won't be able to execute requests unless it can replicate it on the *Quorum* nodes.
- Quorum makes sure that requests which reach quorum, are not lost
- Log provides strict ordering of the requests. So when a request is executed, it is guaranteed to see the results of the immediately previous request.

When this mechanism is avoided, it can run into subtle issues. For read requests on a key-value store, this can result in getting older values and missing on latest updates even when clients read from the node which it thinks as the leader, expecting to always get the latest values.

Consider three nodes as before, athens, byzantium and cyrene. athens is the leader. All three nodes have an existing value for title as 'Nitroservices'



Lets say, athens gets disconnected from the rest of the cluster. byzantium and cyrene then run an election and cyrene is the new leader. athens wont know that it is disconnected, unless it receives some communication, either a heartbeat or a response from the new leader asking it to step down. Alice communicates with cyrene and updates the title to 'Microservices'. After some time, Bob communicates with athens to read the latest value of 'title'. cyrene still think that its the leader.So returns what it thinks as the latest value. Bob ends up reading the old value, after Alice has successfully updated the value, even when Bob thinks that it has read the value from a legitimate leader.

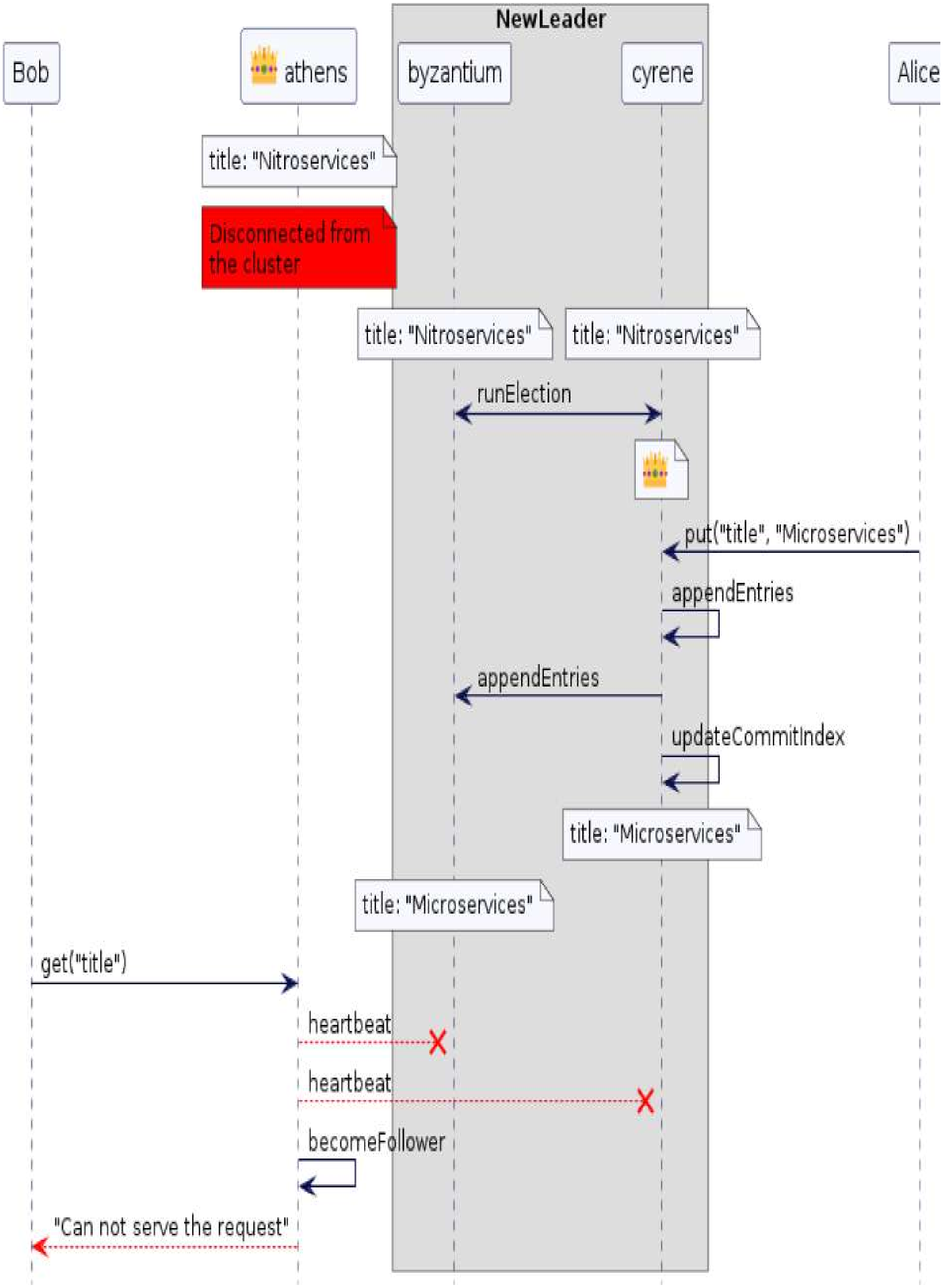

NewLeader
Bob
athens
byzantium
cyrene
Alice
title: "Nitroservices"
Disconnected from the cluster
title: "Nitroservices"
title: "Nitroservices"
runElection
put("title", "Microservices")
appendEntries
appendEntries
updateCommitIndex
title: "Microservices"
title: "Microservices"
get("title")
Nitroservices

Products like etcd [bib-etcd] and Consul [bib-consul] were known to have these issues [bib-etcd-read-issue], which were later fixed. Zookeeper [bib-zookeeper], specifically documents this limitation, and provides no guarantee about getting the latest values in the read requests.

There are two solutions to work around this issue.

- Before serving the read request, the leader can send the heartbeat message to followers. If serves the read request only if it gets a successful response from the Quorum of the followers. This guarantees that there is this leader is still the valid leader. Raft [bib-raft] documents a mechanism which works the same way.

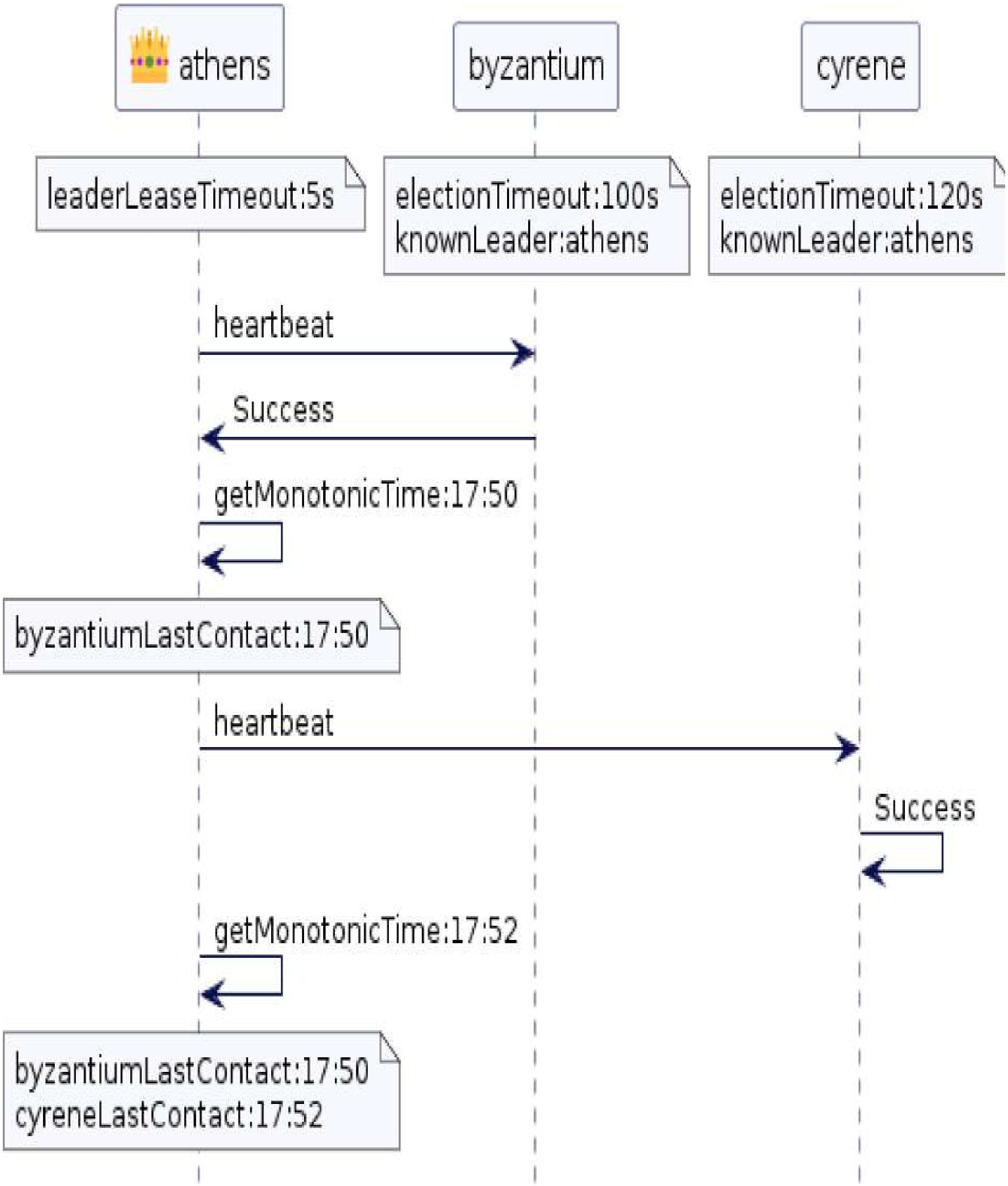
  In the above example, when Bob’s request is handled by athens, it sends the *HeartBeat* to other nodes. If it can not reach the Quorum, then it steps down and returns error to Bob.



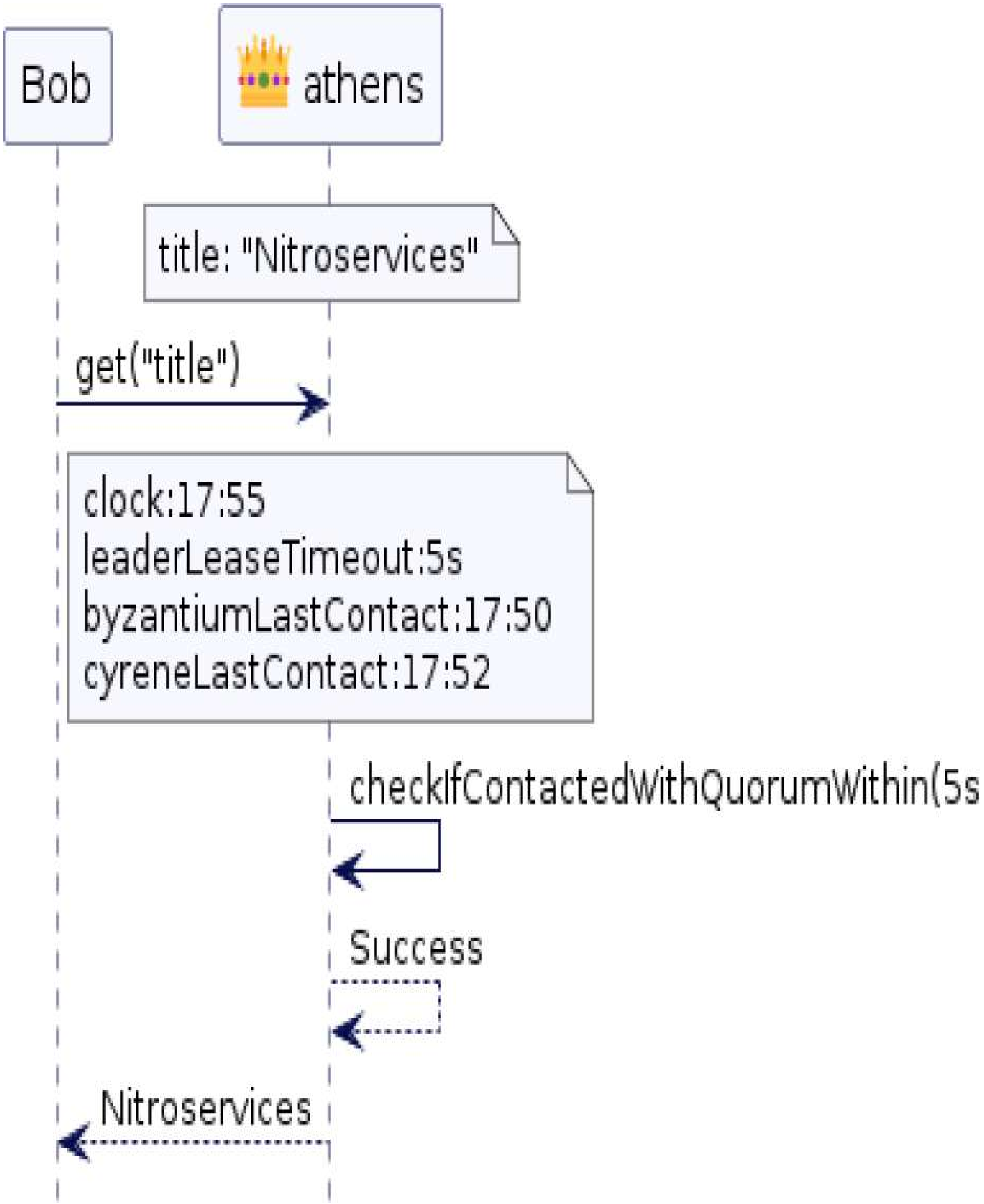
NewLeader
Bob
athens
byzantium
cyrene
Alice
title: "Nitroservices"
Disconnected from the cluster
title: "Nitroservices"
title: "Nitroservices"
runElection
put("title", "Microservices")
appendEntries
appendEntries
updateCommitIndex
title: "Microservices"
title: "Microservices"
get("title")
heartbeat
heartbeat
becomeFollower
"Can not serve the request"

- The network round trip for heartbeats per read request, can be too much of a cost to pay, particularly if the cluster is geo-distributed, with servers placed in distant geographic regions. So often, another solution is used where the leader implements a leader lease [bib-yugabytedb-leader-lease]. This solution depends on the monotonic clocks [bib-linux-clock-gettime]. This makes sure that the old leader steps down if it detects that it is disconnected from the cluster. No other leader is allowed to serve the requests while there is a possibility of older leader still being around.

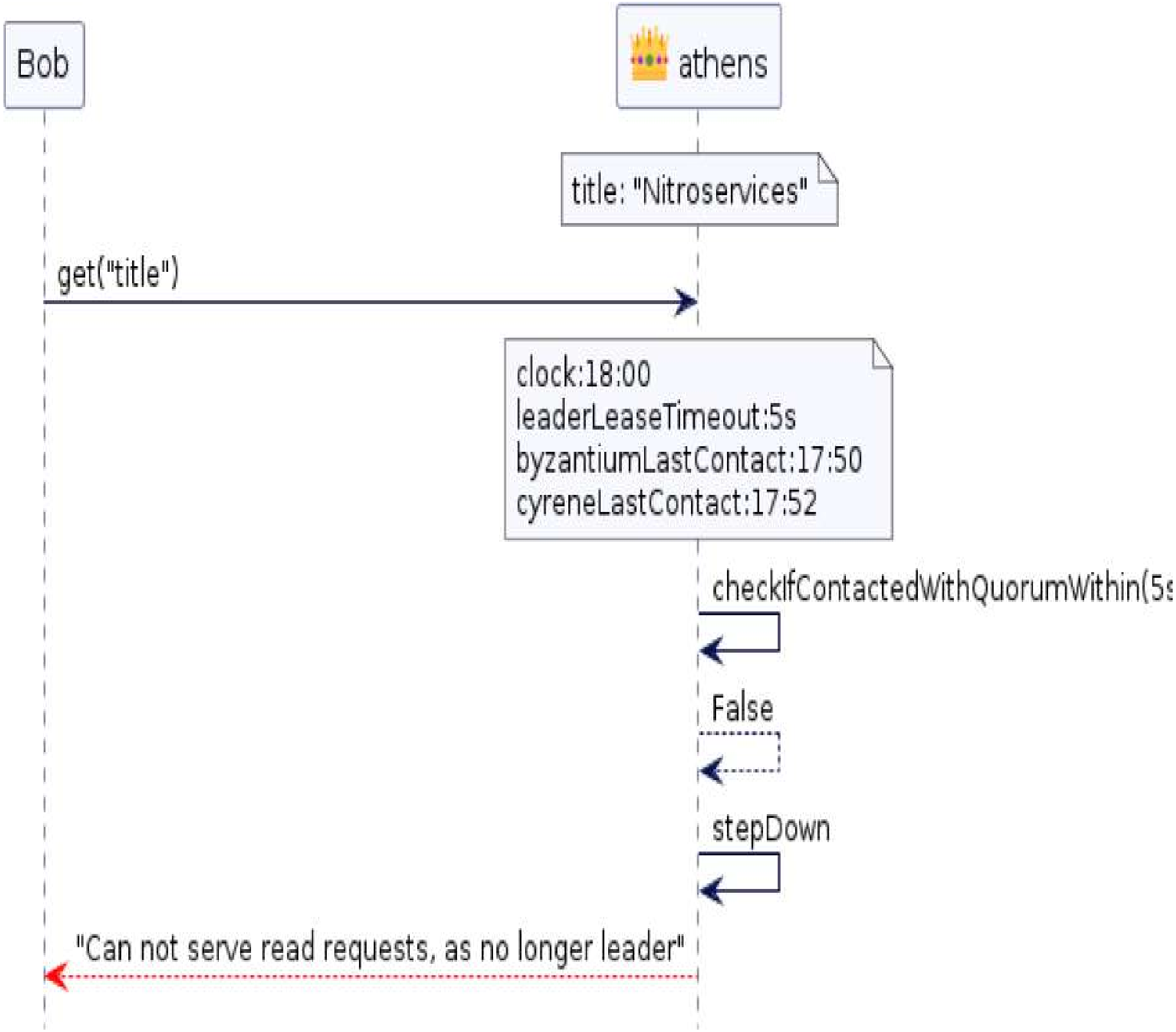
  The leader maintains a time interval called leaderLeaseTimeout within which it expects to get a successful response from the followers. When the leader gets a successful response from the followers for its requests, it marks the time at which it got the successful response from each follower.



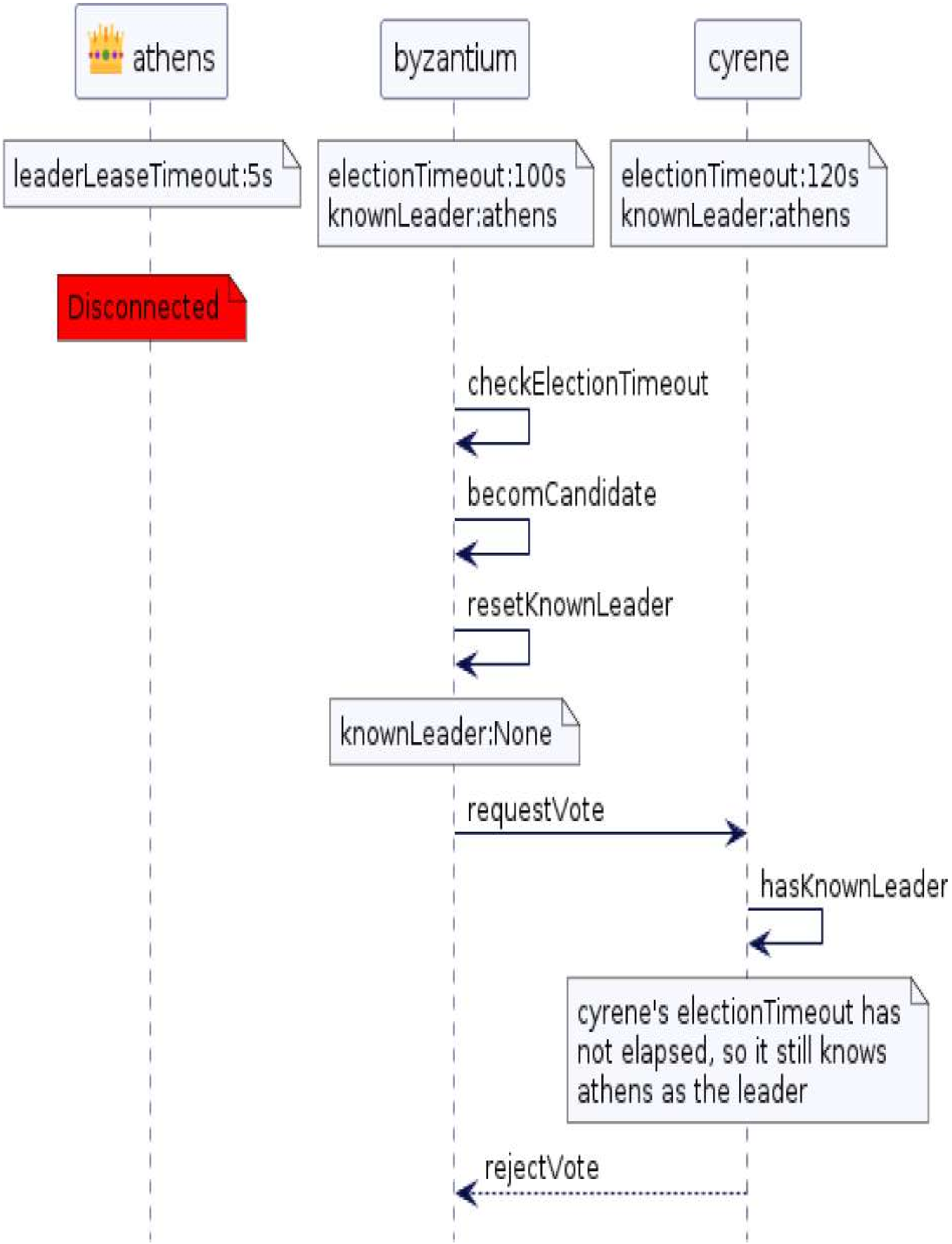
Before serving the read request, it checks if it could contact with the Quorum of the followers within the leaderLeaseTimeout. It serves the read request if it had received successful response from enough followers. This proves that the cluster does not yet have another leader. Let say, leaderLeaseTimeout is 5 seconds. Bob’s request is received by athens at 17:55. It had received responses from cyrene at 17:52. For a three node cluster, athens and cyrene form a quorum. So athens confirms that it could reach quorum within last 5 seconds.



Now, athens did not receive any response after the response from cyrene at 17:52. If Bob sends a read request at 18:00, athens detects that it has not received responses from Quorum of the servers, it steps down, and rejects the read requests.


Bob
athens
title: "Nitroservices"
get("title")
clock:18:00
leaderLeaseTimeout:5s
byzantiumLastContact:17:50
cyreneLastContact:17:52
checkIfContactedWithQuorumWithin(5s
False
stepDown
"Can not serve read requests, as no longer leader"

As discussed in this consul post [bib-consul-leader-lease], the electionTimeout is kept higher than the leaderLeaseTimeout. Followers also keep store the known leader address, which they reset only when electionTimeout period elapses without a HeartBeat from the leader. The followers do not grant vote to any of the vote requests, if they have a known leader.


athens
byzantium
cyrene
leaderLeaseTimeout:5s
electionTimeout:100s
knownLeader:athens
electionTimeout:120s
knownLeader:athens
Disconnected
checkElectionTimeout
becomCandidate
resetKnownLeader
knownLeader:None
requestVote
hasKnownLeader
cyrene's electionTimeout has not elapsed, so it still knows athens as the leader
rejectVote

These two things make sure that as long as the existing leader thinks that it has a leader lease, not other node can win an election, and there can be no other leader.

This implementation assumes that [clock-drift] [bib-clock-drift] across monotonic clocks [bib-linux-clock-gettime] in a cluster is bounded. And electionTimeout on followers will not elapse faster than the leaderLeasetimeout elapses on the leader.

Products like YugabyteDB [bib-yugabyte], etcd [bib-etcd], and Consul [bib-consul] implement a leader lease [bib-yugabytedb-leader-lease] to ensure that there are never two leaders serving the read and write requests.

Examples

Replicated log is the mechanism used by Raft [bib-raft], Multi-Paxos [bib-multi-paxos], Zab [bib-zab] and viewstamped replication [bib-view-stamp-replication] protocols. The technique is called state machine replication [bib-state-machine-replication] where replicas execute the same commands in the same order. *Consistent Core* is often built with state machine replication.

Blockchain implementations like hyperledger fabric [bib-hyperledger-fabric] have an ordering component which is based on a replicated log mechanism. Previous versions of hyperledger fabric used Kafka [bib-kafka] for ordering of the blocks in the blockchain. Recent versions use Raft [bib-raft] for the same purpose.

Chapter 10. Quorum

Avoid two groups of servers making independent decisions, by requiring majority for taking every decision.

Problem

Safety and Liveness

Liveness is the property of the system which says that system always makes progress. Safety is the property which says that the system is always in the correct state. If we focus only on safety, then the system as a whole might not make progress. If we focus only on liveness, then safety might be compromised.

In a distributed system, whenever a server takes any action, it needs to ensure that in the event of a crash the results of the actions are available to the clients. This can be achieved by replicating the result to other servers in the cluster. But that leads to the question: how many other servers need to confirm the replication before the original server can be confident that the update is fully recognized. If the original server waits for too many replications, then it will respond slowly - reducing liveness. But if it doesn't have enough replications, then the update could be lost - a failure of safety. It's critical to balance between the overall system performance and system continuity.

Solution

A cluster agrees that it's received an update when a majority of the nodes in the cluster have acknowledged the update. We call this number a quorum. So if we have a cluster of five nodes, we need a quorum of three. (For a cluster of n nodes, the quorum is n/2 + 1.)

The need for a quorum indicates how many failures can be tolerated - which is the size of the cluster minus the quorum. A cluster of five nodes can tolerate two of them failing. In general, if we want to tolerate 'f' failures we need a cluster size of 2f + 1

Consider following two examples that need a quorum:

- **Updating data in a cluster of servers.** *High-Water Mark* is used to ensure only data which is guaranteed to be available on the majority of servers is visible to clients.
- **Leader election.** In *Leader and Followers*, a leader is selected only if it gets votes from a majority of the servers.

Deciding on number of servers in a cluster

- In his book Guide to Reliable and Scalable Distributed Systems [bib-birman] Dr. Kenneth Birman builds on the analysis done by Dr. Jim Gray for the world of relational databases. Dr. Birman states that the throughput of quorum-based systems can go down as O(1 / n ** 2), where 'n' is the number of servers in a cluster.
- Zookeeper [bib-zookeeper-wait-free] and other consensus based systems are known to have lower write throughput when number of servers in a cluster go beyond five
- In his talk Applying The Universal Scalability Law to Distributed Systems [bib-usl-to-dist-sys], Dr. Neil Gunther shows, how the throughput of the system goes down with the number of coordinating servers in a cluster

The cluster can function only if majority of servers are up and running. In systems doing data replication, there are two things to consider:

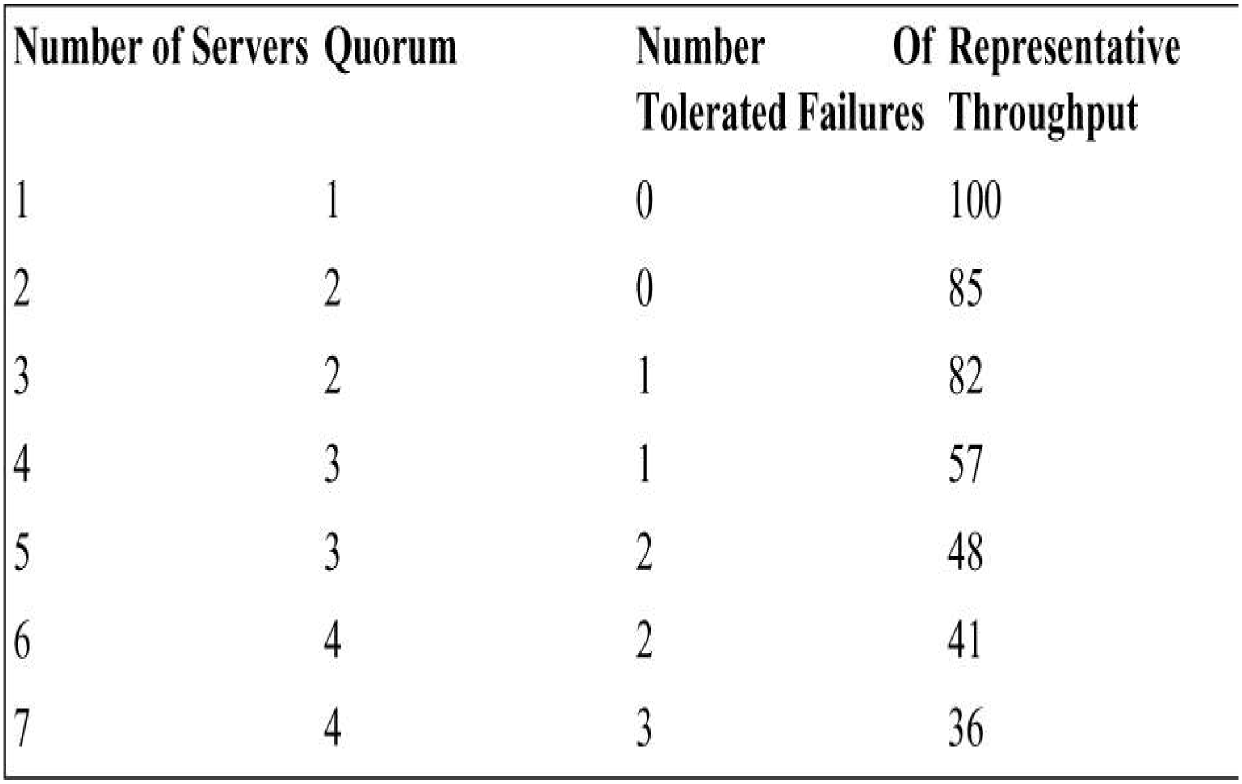
Every time data is written to the cluster, it needs to be copied to multiple servers. Every additional server adds some overhead to complete this write. The latency of data write is directly proportional to the number of servers forming the quorum. As we will see below, doubling the number of servers in a cluster will reduce throughput to half of the value for the original cluster.

The number of server failures tolerated is dependent on the size of the cluster. But just adding one more server to an existing cluster doesn’t always give more fault tolerance: adding one server to a three server cluster doesn’t increase failure tolerance.

- The throughput of write operations.
- The number of failures which need to be tolerated.

Considering these two factors, most practical quorum-based systems have cluster sizes of three or five. A five-server cluster tolerates two server failures and has tolerable data write throughput of few thousand requests per second.

Here is an example of how to choose the number of servers, based on the number of tolerated failures and approximate impact on the throughput. The throughput column shows approximate relative throughput to highlight how throughput degrades with the number of servers. The number will vary from system to system. As an example, readers can refer to the actual throughput data published in Raft Thesis [bib-raft-phd] and the original Zookeeper paper [bib-zookeeper-hunt-paper].



Number of Servers	Quorum	Number Of Tolerated Failures	Representative Throughput
1	1	0	100
2	2	0	85
3	2	1	82
4	3	1	57
5	3	2	48
6	4	2	41
7	4	3	36

Examples

- All the consensus implementations like Zab [bib-zab], Raft [bib-raft], Paxos [bib-paxos] are quorum based.
- Even in systems which don’t use consensus, quorum is used to make sure the latest update is available to at least one server in case of failures or network partition. For instance, in databases like Cassandra [bib-cassandra], a database update can be configured to return success only after a majority of the servers have updated the record successfully.

Chapter 11. Generation Clock

A monotonically increasing number indicating the generation of the server.

Also known as: Term

Also known as: Epoch

Also known as: Generation

Problem

In *Leader and Followers* setup, there is a possibility of the leader being temporarily disconnected from the followers. There might be a garbage collection pause in the leader process, or a temporary network disruption which disconnects the leader from the follower. In this case the leader process is still running, and after the pause or the network disruption is over, it will try sending replication requests to the followers. This is dangerous, as meanwhile the rest of the cluster might have selected a new leader and accepted requests from the client. It is important for the rest of the cluster to detect any requests from the old leader. The old leader itself should also be able to detect that it was temporarily disconnected from the cluster and take necessary corrective action to step down from leadership.

Solution

> Generation Clock pattern is an example of a Lamport timestamp [bib-lamport-timestamp]: a simple technique used to determine ordering of events across a set of processes, without depending on a system clock. Each process maintains an integer counter, which is incremented after

> every action the process performs. Each process also sends this integer to other processes along with the messages processes exchange. The process receiving the message sets its own integer counter by picking up the maximum between its own counter and the integer value of the message. This way, any process can figure out which action happened before the other by comparing the associated integers. The comparison is possible for actions across multiple processes as well, if the messages were exchanged between the processes. Actions which can be compared this way are said to be ‘causally related’.

Maintain a monotonically increasing number indicating the generation of the server. Every time a new leader election happens, it should be marked by increasing the generation. The generation needs to be available beyond a server reboot, so it is stored with every entry in the *Write-Ahead Log*. As discussed in *High-Water Mark*, followers use this information to find conflicting entries in their log.

At startup, the server reads the last known generation from the log.

class ReplicatedLog…

```
this.replicationState = new ReplicationState(config, wal.getLastLog
```

With Leader and Followers servers increment the generation every time there’s a new leader election.

class ReplicatedLog…

```
private void startLeaderElection() {
   replicationState.setGeneration(replicationState.getGeneration()
   registerSelfVote();
   requestVoteFrom(followers);
}
```

The servers send the generation to other servers as part of the vote requests. This way, after a successful leader election, all the servers have the same generation. Once the leader is elected, followers are told about the new generation

follower (class ReplicatedLog...)

```
private void becomeFollower(int leaderId, Long generation) {
   replicationState.reset();
   replicationState.setGeneration(generation);
   replicationState.setLeaderId(leaderId);
   transitionTo(ServerRole.FOLLOWING);
}
```

Thereafter, the leader includes the generation in each request it sends to the followers. It includes it in every *HeartBeat* message as well as the replication requests sent to followers.

Leader persists the generation along with every entry in its Write-Ahead Log

leader (class ReplicatedLog...)

```
Long appendToLocalLog(byte[] data) {
  Long generation = replicationState.getGeneration();
  return appendToLocalLog(data, generation);
}
Long appendToLocalLog(byte[] data, Long generation) {
  var logEntryId = wal.getLastLogIndex() + 1;
  var logEntry = new WALEntry(logEntryId, data, EntryType.DATA, gen
  return wal.writeEntry(logEntry);
}
```

This way, it is also persisted in the follower log as part of the replication mechanism of Leader and Followers

If a follower gets a message from a deposed leader, the follower can tell because its generation is too low. The follower then replies with a failure response.

follower (class ReplicatedLog...)

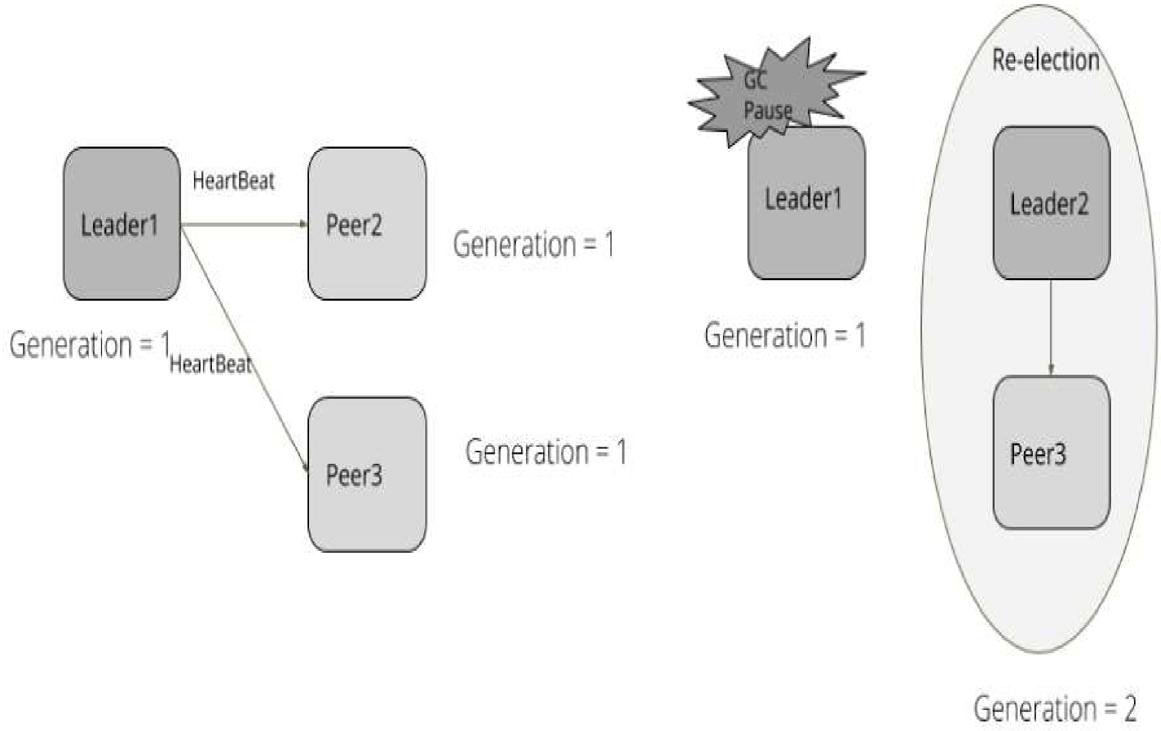
```
Long currentGeneration = replicationState.getGeneration();
if (currentGeneration > request.getGeneration()) {
   return new ReplicationResponse(FAILED, serverId(), currentGenera
}
```

When a leader gets such a failure response, it becomes a follower and expects communication from the new leader.

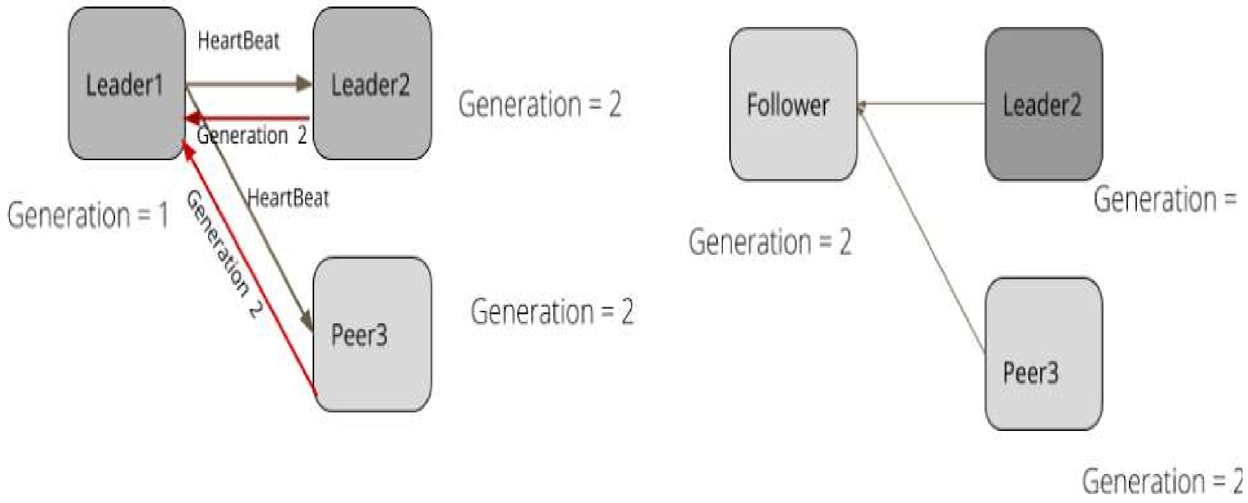
Old leader (class ReplicatedLog...)

```
if (!response.isSucceeded()) {
   if (response.getGeneration() > replicationState.getGeneration())
      becomeFollower(LEADER_NOT_KNOWN, response.getGeneration());
      return;
   }
```

Consider the following example. In the three server cluster, leader1 is the existing leader. All the servers in the cluster have the generation as 1. Leader1 sends continuous heartbeats to the followers. Leader1 has a long garbage collection pause, for say 5 seconds. The followers did not get a heartbeat, and timeout to elect a new leader. The new leader increments the generation to 2. After the garbage collection pause is over, leader1 continues sending the requests to other servers. The followers and the new leader which are at generation 2, reject the request and send a failure response with generation 2. leader1 handles the failure response and steps down to be a follower, with generation updated to 2.



Responds with Failure and
New Generation



Examples

Raft [bib-raft] uses the concept of a Term for marking the leader generation.

In Zookeeper [bib-zab], an epoch number is maintained as part of every transaction id. So every transaction persisted in Zookeeper has a generation marked by epoch.

In Cassandra [bib-cassandra] each server stores a generation number which is incremented every time a server restarts. The generation information is persisted in the system keyspace and propagated as part of the gossip messages to other servers. The servers receiving the gossip message can then compare the generation value it knows about and the generation value in the gossip message. If the generation in the gossip message is higher, it knows that the server was restarted and then discards all the state it has maintained for that server and asks for the new state.

In Kafka [bib-kafka] an epoch number is created and stored in Zookeeper every time a new Controller is elected for a kafka cluster. The epoch is included in every request that is sent from controller to other servers in the cluster. Another epoch called LeaderEpoch [bib-kafka-leader-epoch] is maintained to know if the followers a partition are lagging behind in their High-Water Mark.

Chapter 12. High-Water Mark

An index in the write ahead log showing the last successful replication.

Also known as: CommitIndex

Problem

The *Write-Ahead Log* pattern is used to recover state after the server crashes and restarts. But a write-ahead log is not enough to provide availability in case of server failure. If a single server fails, then clients won't be able to function until the server restarts. To get a more available system, we can replicate the log on multiple servers. Using *Leader and Followers* the leader replicates all its log entries to a *Quorum* of followers. Now should the leader fail, a new leader can be elected, and clients can mostly continue to work with the cluster as before. But there are still a couple things that can go wrong:

- The leader can fail before sending its log entries to any followers.
- The leader can fail after sending log entries to some followers, but could not send it to the majority of followers.

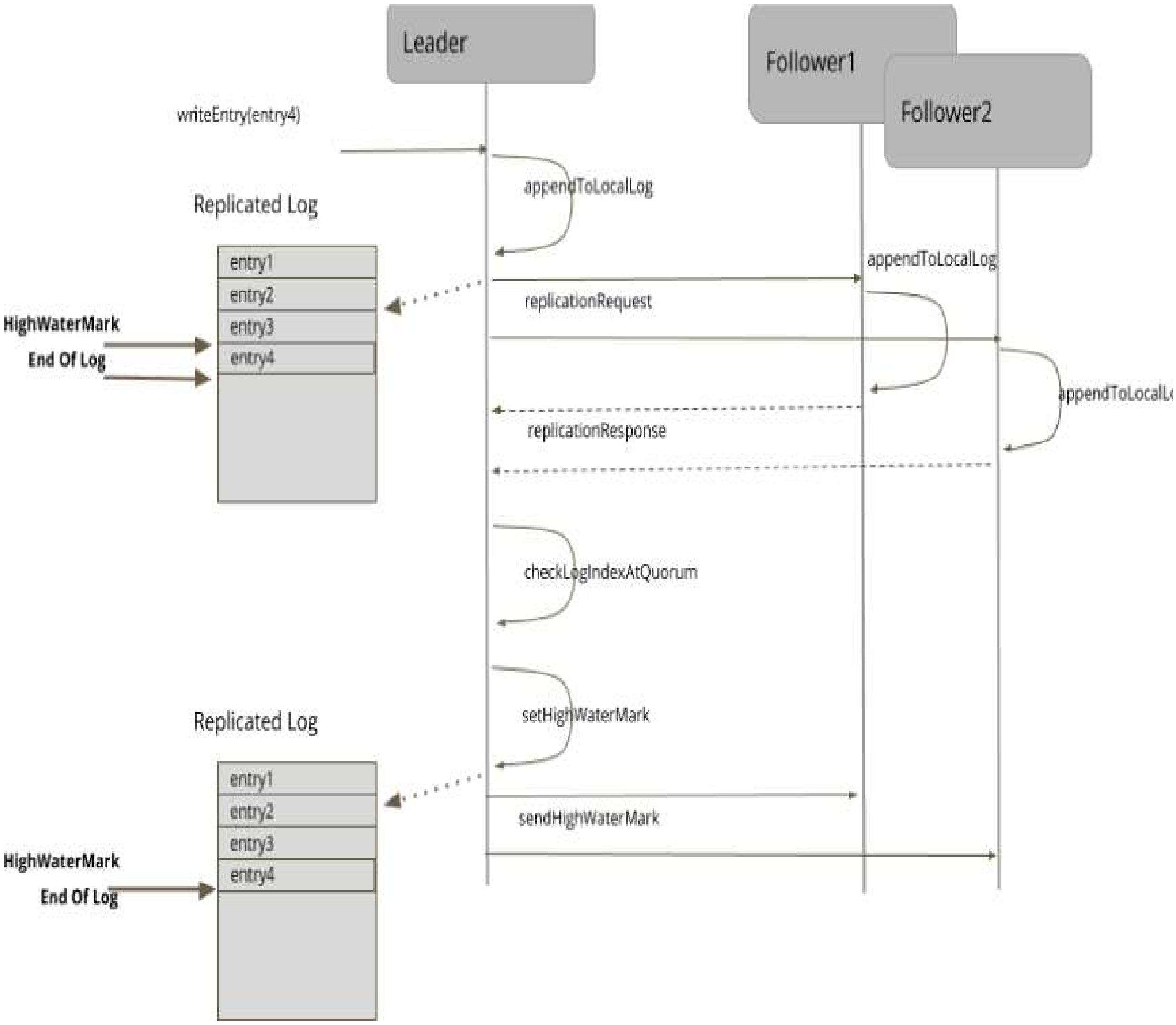
In these error scenarios, some followers can be missing entries in their logs, and some followers can have more entries than others. So it becomes important for each follower to know what part of the log is safe to be made available to the clients.

Solution

The high-water mark is an index into the log file that records the last log entry that is known to have successfully replicated to a Quorum of followers.

The leader also passes on the high-water mark to its followers during its replication. All servers in the cluster should only transmit data to clients that reflects updates that are below the high-water mark.

Here’s the sequence of operations.



For each log entry, the leader appends it to its local write ahead log, and then sends it to all the followers.

leader (class ReplicatedLog...)

```
private Long appendAndReplicate(byte[] data) {
   Long lastLogEntryIndex = appendToLocalLog(data);
   replicateOnFollowers(lastLogEntryIndex);
   return lastLogEntryIndex;
}


private void replicateOnFollowers(Long entryAtIndex) {
   //FIXME: factorout as a separate method.
   oldLeaderLeaseRemainingTime = System.nanoTime() + leaderLeaseTim

   for (final FollowerHandler follower : followers) {
     replicateOn(follower, entryAtIndex); //send replication reques
   }
}
```

The followers handle the replication request and append the log entries to their local logs. After successfully appending the log entries, they respond to the leader with the index of the latest log entry they have. The response also includes the current *Generation Clock* of the server.

follower (class ReplicatedLog...)

```
private ReplicationResponse appendEntries(ReplicationRequest replic
   List<WALEntry> entries = replicationRequest.getEntries();
   entries.stream()
        .filter(e -> !wal.exists(e))
        .forEach(e -> wal.writeEntry(e));
   return new ReplicationResponse(SUCCEEDED, serverId(), replicatio
}
```

The Leader keeps track of log indexes replicated at each server, when responses are received.

class ReplicatedLog…

```
logger.info("Updating matchIndex for " + response.getServerId() + "
updateMatchingLogIndex(response.getServerId(), response.getReplicat
var logIndexAtQuorum = computeHighwaterMark(logIndexesAtAllServers(
var currentHighWaterMark = replicationState.getHighWaterMark();
```

```
if (logIndexAtQuorum > currentHighWaterMark && logIndexAtQuorum !=
   applyLogEntries(currentHighWaterMark, logIndexAtQuorum);
   replicationState.setHighWaterMark(logIndexAtQuorum);
}
```

The high-water mark can be calculated by looking at the log indexes of all the followers and the log of the leader itself, and picking up the index which is available on the majority of the servers.

class ReplicatedLog…

```
Long computeHighwaterMark(List<Long> serverLogIndexes, int noOfServ
  serverLogIndexes.sort(Long::compareTo);
  return serverLogIndexes.get(noOfServers / 2);
}
```

A subtle problem can come up with leader election. We must ensure all the servers in the cluster have an up-to-date log before any server sends data to clients.

There is a subtle issue in the case where the existing leader fails before propagating the high-water mark to all the followers. RAFT does this by appending a no-op entry to the leader's log after a successful leader election, and only serves clients once this is confirmed by its followers. In ZAB, the new leader explicitly tries to push all its entries to all the followers before starting to serve the clients.

The leader propagates the high-water mark to the followers either as part of the regular *HeartBeat* or as separate requests. The followers then set their high-water mark accordingly.

Any client can read the log entries only till the high-water mark. Log entries beyond the high-water mark are not visible to clients as there is no confirmation that the entries are replicated, and so they might not be available if the leader fails, and some other server is elected as a leader.

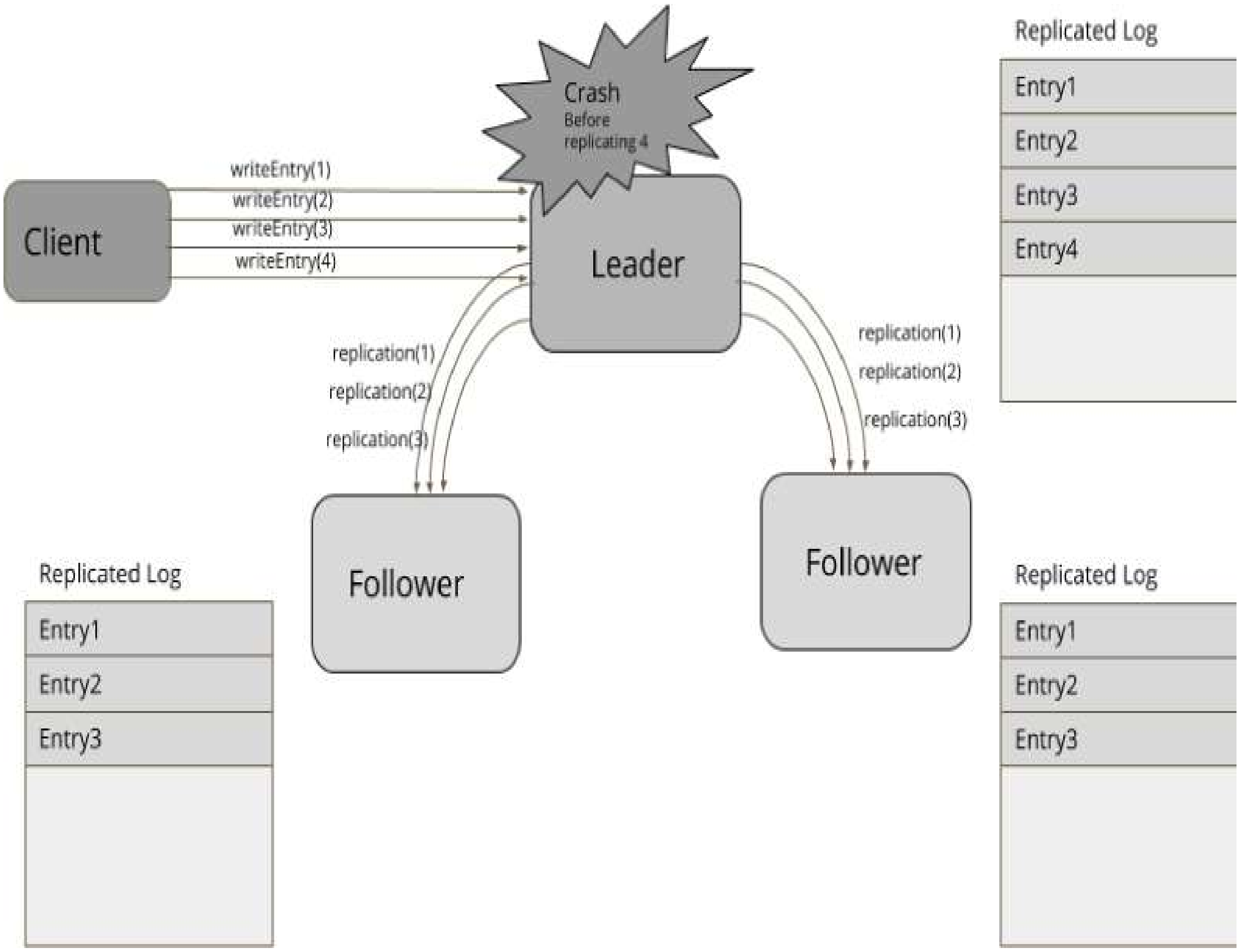
class ReplicatedLog…

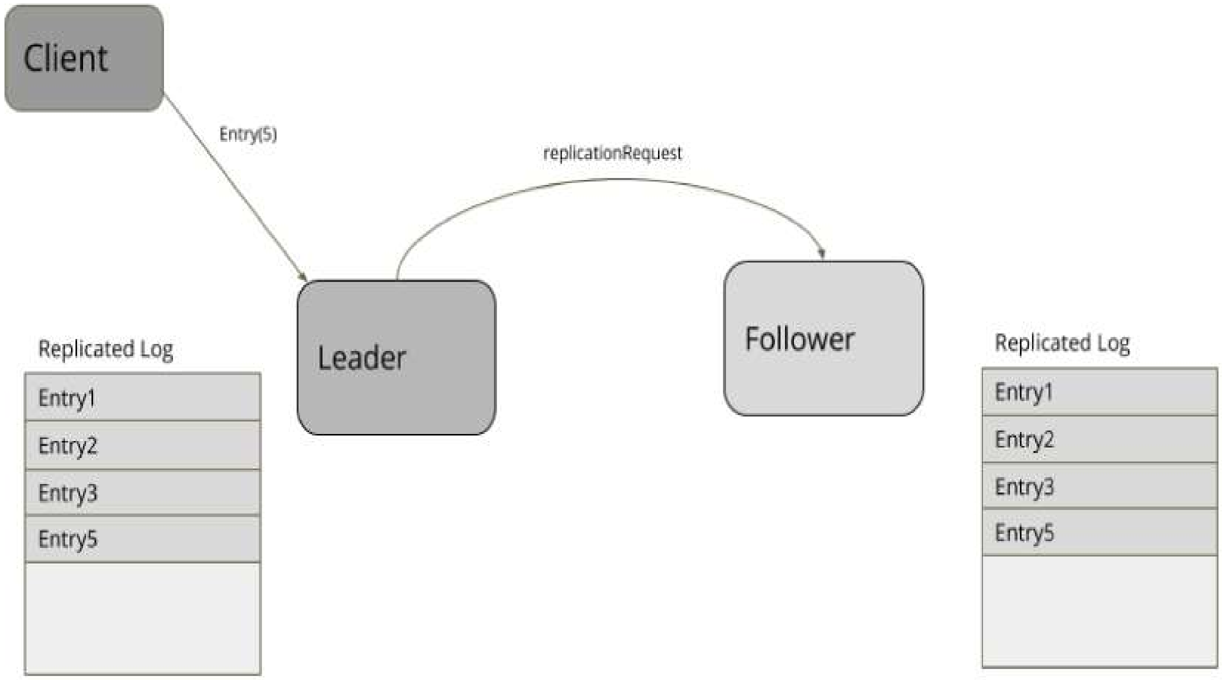
```
public WALEntry readEntry(long index) {
  if (index > replicationState.getHighWaterMark()) {
     throw new IllegalArgumentException("Log entry not available");
  }
  return wal.readAt(index);
}
```

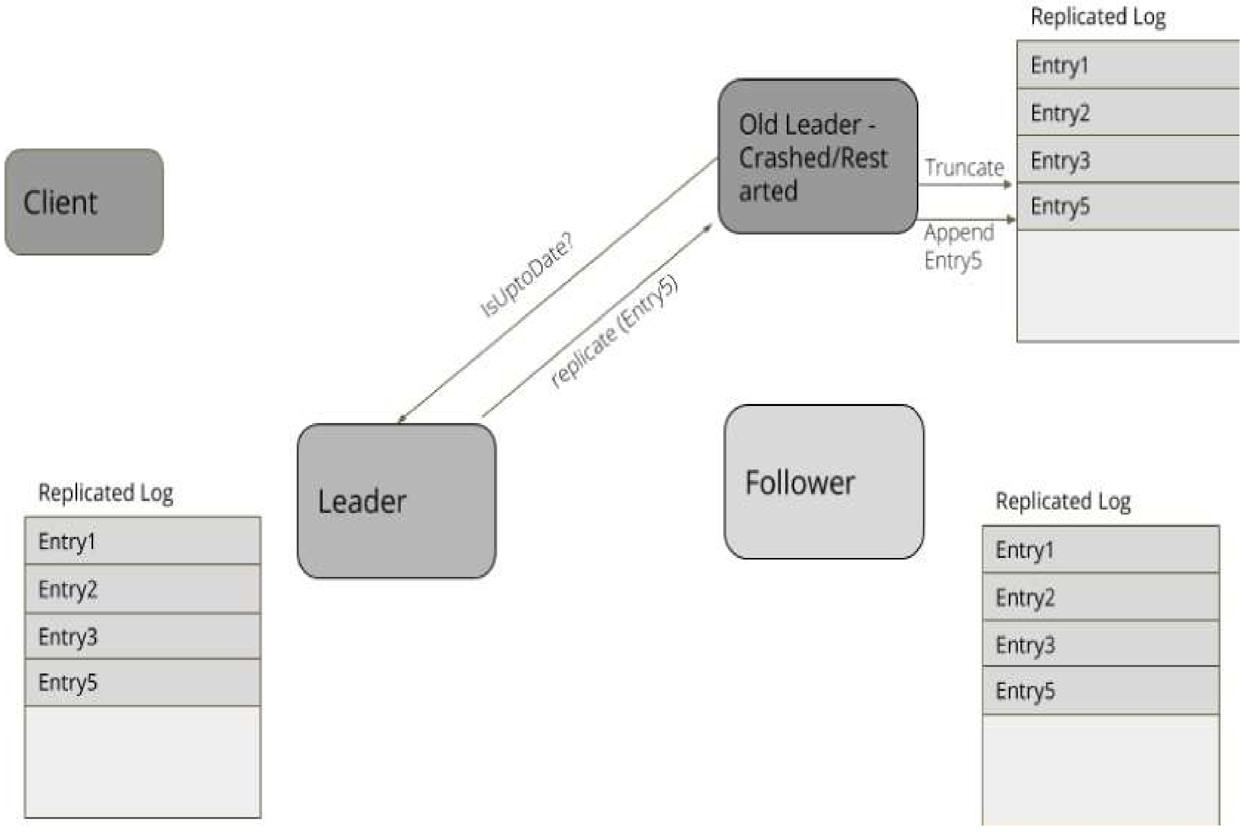
Log Truncation

When a server joins the cluster after crash/restart, there is always a possibility of having some conflicting entries in its log. So whenever a server joins the cluster, it checks with the leader of the cluster to know which entries in the log are potentially conflicting. It then truncates the log to the point where entries match with the leader,and then updates the log with the subsequent entries to ensure its log matches the rest of the cluster.

Consider the following example. The client sends requests to add four entries in the log. The leader successfully replicates three entries, but fails after adding entry4 to its own log. One of the followers is elected as a new leader and accepts more entries from the client. When the failed leader joins the cluster again, it has entry4 which is conflicting. So it needs to truncate its log till entry3, and then add entry5 to match the log with the rest of the cluster.


Crash
Before replicating 4
Client
writeEntry(1)
writeEntry(2)
writeEntry(3)
writeEntry(4)
Leader
replication(1)
replication(2)
replication(3)
replication(1)
replication(2)
replication(3)
Follower
Follower
Replicated Log
Entry1
Entry2
Entry3
Entry4
Replicated Log
Entry1
Entry2
Entry3
Replicated Log
Entry1
Entry2
Entry3


Client
Entry(5)
replicationRequest
Leader
Follower
Replicated Log
Entry1
Entry2
Entry3
Entry5
Replicated Log
Entry1
Entry2
Entry3
Entry5


Replicated Log
Entry1
Entry2
Entry3
Entry5
Old Leader - Crashed/Restarted
Truncate
Append Entry5
Client
IsUptoDate?
replicate (Entry5)
Leader
Follower
Replicated Log
Entry1
Entry2
Entry3
Entry5
Replicated Log
Entry1
Entry2
Entry3
Entry5

Any server which restarts or rejoins the cluster after a pause, finds the new leader. It then explicitly asks for the current high-water mark, truncates its log to high-water mark, and then gets all the entries beyond high-water mark from the leader. Replication algorithms like RAFT have ways to find out conflicting entries by checking log entries in its own log with the log entries in the request. The entries with the same log index, but at lower *Generation Clock*, are removed.

class ReplicatedLog…

```
void maybeTruncate(ReplicationRequest replicationRequest) {
  replicationRequest.getEntries().stream()
       .filter(entry -> wal.getLastLogIndex() >= entry.getEntryInde
              entry.getGeneration() != wal.readAt(entry.getEntryInd
       .forEach(entry -> wal.truncate(entry.getEntryIndex()));
}
```

A simple implementation to support log truncation is to keep a map of log indexes and file position. Then the log can be truncated at a given index, as following:

class WALSegment…

```
public synchronized  void truncate(Long logIndex) throws IOExceptio
  var filePosition = entryOffsets.get(logIndex);
  if (filePosition == null) throw new IllegalArgumentException("No

  fileChannel.truncate(filePosition);
  truncateIndex(logIndex);
}

private void truncateIndex(Long logIndex) {
   entryOffsets.entrySet().removeIf(entry -> entry.getKey() >= logI
}
```

Examples

- All the consensus algorithms use the concept of high-water mark to know when to apply the proposed state mutations. e.g. In the RAFT [bib-raft] consensus algorithm, high-water mark is called ‘CommitIndex’.
- In Kafka replication protocol [bib-kafka-replication-protocol], there is a separate index maintained called ‘high-water mark’. Consumers can see entries only until the high-water mark.
- Apache BookKeeper [bib-bookkeeper] has a concept of ‘last add confirmed [bib-bookkeeper-protocol]’, which is the entry which is successfully replicated on quorum of bookies.

Chapter 13. Singular Update Queue

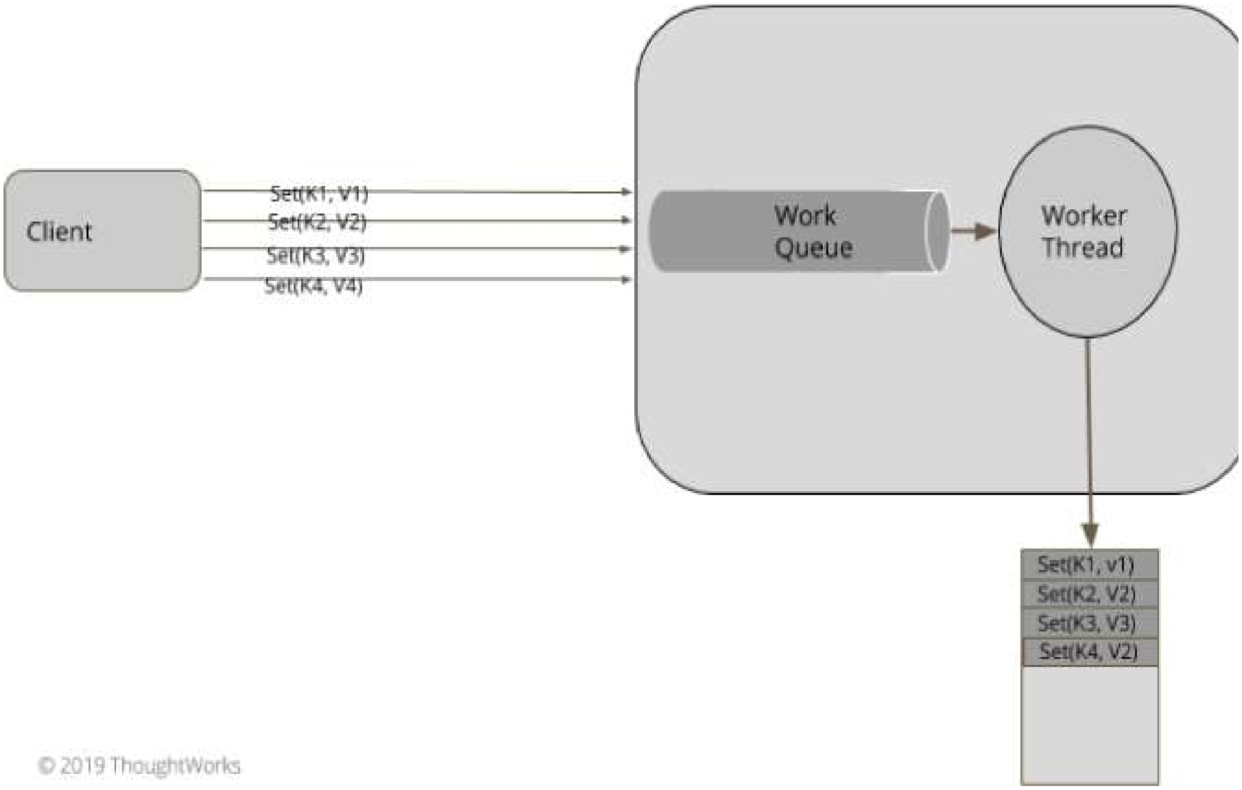
Use a single thread to process requests asynchronously to maintain order without blocking the caller.

Problem

When the state needs to be updated by multiple concurrent clients, we need it to be safely updated with one at a time changes. Consider the example of the *Write-Ahead Log* pattern. We need entries to be processed one at a time, even if several concurrent clients are trying to write. Generally locks are used to protect against concurrent modifications. But if the tasks being performed are time consuming, like writing to a file, blocking all the other calling threads until the task is completed can have severe impact on overall system throughput and latency. It is important to make effective use of compute resources, while still maintaining the guarantee of one at a time execution.

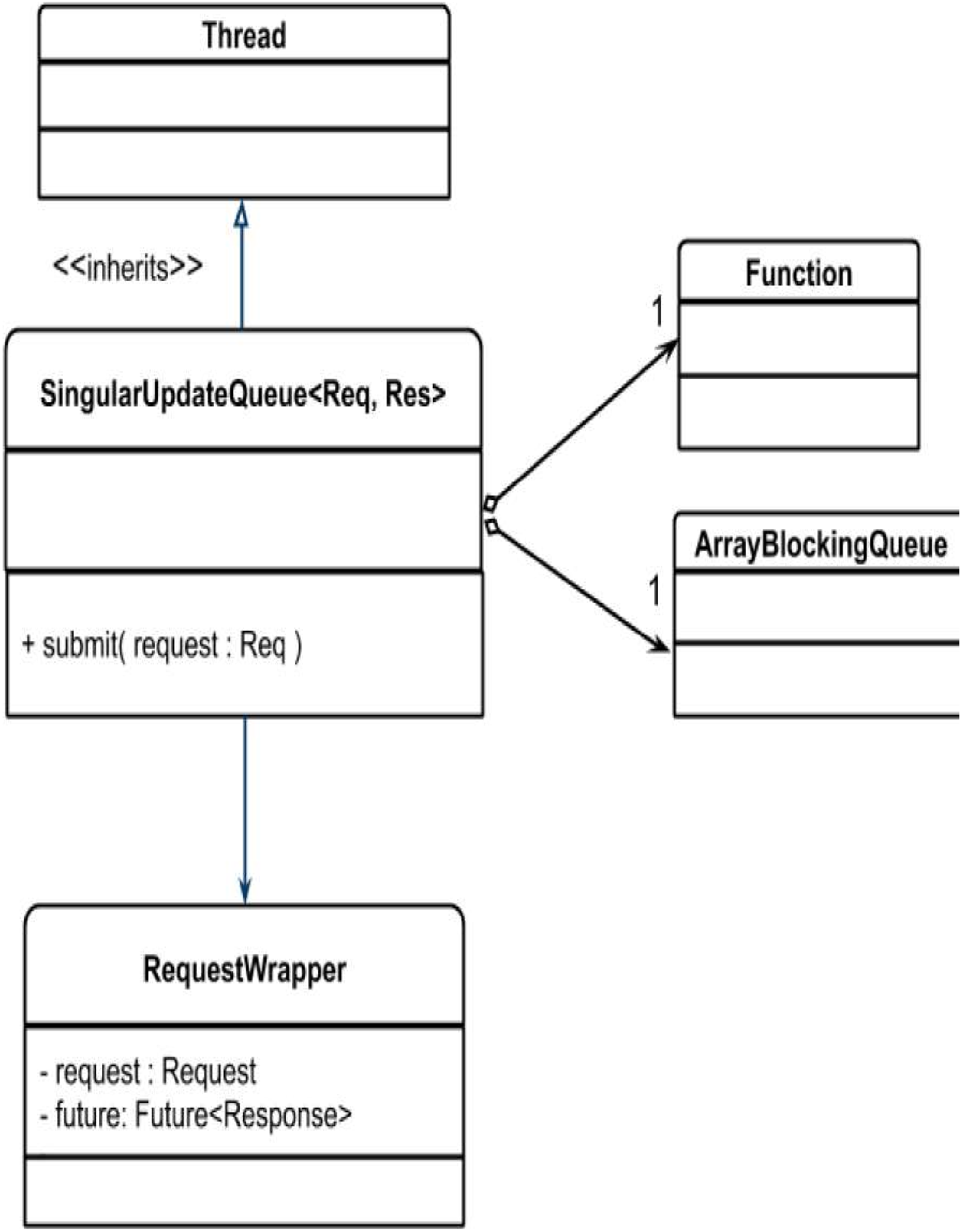
Solution

Implement a workqueue and a single thread working off the queue. Multiple concurrent clients can submit state changes to the queue. But a single thread works on state changes. This can be naturally implemented with goroutines and channels in languages like golang.



A typical Java implementation looks like following:

The implementation shown here is by using Java's Thread class, just to demonstrate basic code structure. It is possible to use Java's ExecutorService with single thread, to achieve the same. You can refer to book Java Concurrency In Practice [bib-java-concurrency-in-practice] to know more about using ExecutorService.


Thread
<<inherits>>
SingularUpdateQueue<Req, Res>
+ submit(request : Req)
Function
1
ArrayBlockingQueue
1
RequestWrapper
- request : Request
- future: Future<Response>

A SingularUpdateQueue has a queue and a function to be applied for work items in the queue. It extends from java.lang.Thread, to make sure that it has its own single thread of execution.

```
public class SingularUpdateQueue<Req, Res> extends Thread implement
  private ArrayBlockingQueue<RequestWrapper<Req, Res>> workQueue
       = new ArrayBlockingQueue<RequestWrapper<Req, Res>>(100);
  private Function<Req, Res> handler;
  private volatile boolean isRunning = false;
```

Clients submit requests to the queue on their own threads. The queue wraps each request in a simple wrapper to combine it with a future, returning the future to the client so that the client can react once the request is eventually completed.

class SingularUpdateQueue…

```
public CompletableFuture<Res> submit(Req request) {
  try {
     var requestWrapper = new RequestWrapper<Req, Res>(request);
     workQueue.put(requestWrapper);
     return requestWrapper.getFuture();
  }
  catch (InterruptedException e) {
     throw new RuntimeException(e);
  }
 }

class RequestWrapper<Req, Res> {
  private final CompletableFuture<Res> future;
  private final Req request;

  public RequestWrapper(Req request) {
    this.request = request;
    this.future = new CompletableFuture<Res>();
  }
  public CompletableFuture<Res> getFuture() { return future; }
  public Req getRequest()          { return request; }
```

The elements in the queue are processed by the single dedicated thread that SingularUpdateQueue inherits from Thread. The queue allows multiple concurrent producers to add the tasks for execution. The queue implementation should be thread safe, and should not add a lot of overhead under contention. The execution thread picks up requests from the queue and process them one at a time. The CompletableFuture is completed with the response of the task execution.

class SingularUpdateQueue…

```
@Override
public void run() {
  isRunning = true;
  while(isRunning) {
    Optional<RequestWrapper<Req, Res>> item = take();
    item.ifPresent(requestWrapper -> {
       try {
          Res response = handler.apply(requestWrapper.getRequest())
          requestWrapper.complete(response);

       } catch (Exception e) {
         requestWrapper.completeExceptionally(e);
       }
     });
   }
}
```

class RequestWrapper…

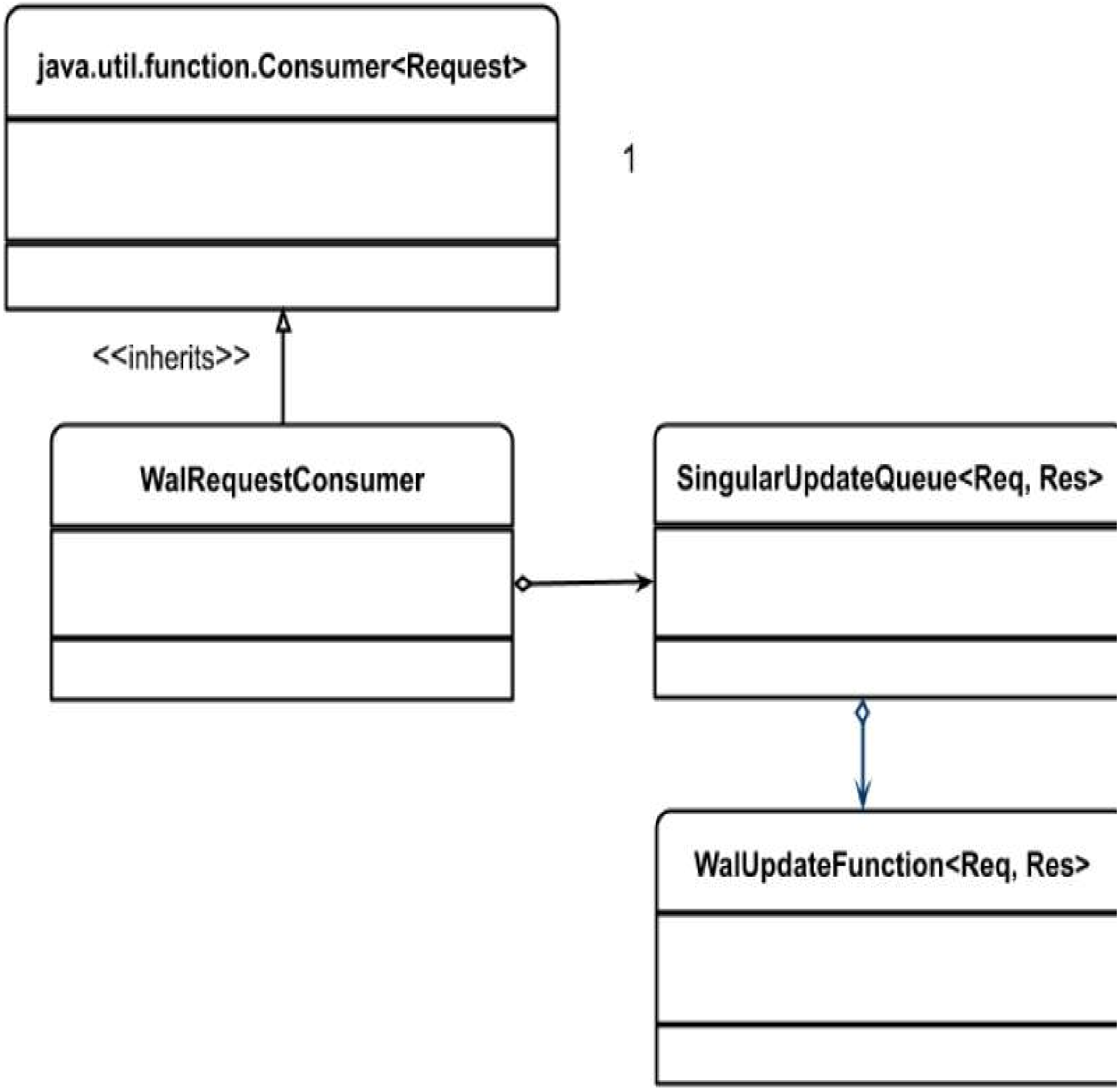
```
public void complete(Res response) {
  future.complete(response);
}

public void completeExceptionally(Exception e) {
  e.printStackTrace();
  getFuture().completeExceptionally(e);
}
```

It’s useful to note that we can put a timeout while reading items from the queue, instead of blocking it indefinitely. It allows us to exit the thread if needed, with isRunning set to false, and the queue will not block indefinitely if it’s empty, blocking the execution thread. So we use the poll method with a timeout, instead of the take method, which blocks indefinitely. This gives us the ability to shutdown the thread of execution cleanly.

class SingularUpdateQueue…

```
private Optional<RequestWrapper<Req, Res>> take() {
  try {
    return Optional.ofNullable(workQueue.poll(2, TimeUnit.MILLISECO

  } catch (InterruptedException e) {
    return Optional.empty();
  }
}



public void shutdown() {
  this.isRunning = false;
}
```

For example, a server processing requests from multiple clients and updating write ahead log, can have use a SingularUpdateQueue as following.


java.util.function.Consumer<Request>
1
<<inherits>>
WalRequestConsumer
SingularUpdateQueue<Req, Res>
WalUpdateFunction<Req, Res>

A client of the SingularUpdateQueue would set it up by specifying its paramaterized types and the function to run when processing the message from the queue. For this example, we're using a consumer of requests for a write ahead log. There is a single instance of this consumer, which will control access to the log data structure. The consumer needs to put each request into a log and then return a response. The response message can only be sent after the message has been put into the log. We use a SingularUpdateQueue to ensure there's a reliable ordering for these actions.

```
public class WalRequestConsumer implements Consumer<Message<Request

  private final SingularUpdateQueue<Message<RequestOrResponse>, Mes
  private final WriteAheadLog wal;

  public WalRequestConsumer(Config config) {
    this.wal = WriteAheadLog.openWAL(config);
  walWriterQueue = new SingularUpdateQueue<>((message) -> {
    wal.writeEntry(serialize(message));
    return responseMessage(message);
  });
  startHandling();
}

private void startHandling() { this.walWriterQueue.start(); }
```

The consumer's accept method takes messages, puts them on the queue and after each message is processed, sends a response. This method is run on the caller's thread, allowing many callers to invoke accept at the same time.

```
@Override
public void accept(Message message) {
  CompletableFuture<Message<RequestOrResponse>> future = walWriterQ
  future.whenComplete((responseMessage, error) -> {
    sendResponse(responseMessage);
  });
}
```

Choice of the queue

The choice of the queue data structure is an important one to be made. On JVM, there are various data structures available to chose from:

As the name suggests, this is an array-backed blocking queue. This is used when a fixed bounded queue needs to be created. Once the queue fills up, the producer will block. This provides blocking backpressure and is helpful when we have slow consumers and fast producers

ConcurrentLinkedQueue can be used when we do not have consumers waiting for the producer, but there is some coordinator which schedules consumers only after tasks are queued onto the ConcurrentLinkedQueue.

This is mostly used when unbounded queuing needs to be done, without blocking the producer. We need to be careful with this choice, as the queue might fill up quickly if no backpressure techniques are implemented and can go on consuming all the memory

As discussed in LMAX Disruptor, sometimes, task processing is latency sensitive. So much so, that copying tasks between processing stages with ArrayBlockingQueue can add to latencies which are not acceptable. RingBuffer [bib-lmax] can be used in these cases to pass tasks between stages.

- ArrayBlockingQueue (Used in Kafka request queue)
- ConcurrentLinkedQueue along with ForkJoinPool (Used in Akka Actors mailbox implementation)
- LinkedBlockingDeque (Used By Zookeeper and Kafka response queue)
- RingBuffer (Used in LMAX Disruptor,)

Using Channels and Lightweight Threads.

This can be a natural fit for languages or libraries which support lightweight threads along with the concept of channels (e.g. golang, kotlin). All the requests are passed to a single channel to be processed. There is a single goroutine which processes all the messages to update state. The response is then written to a separate channel, and processed by separate goroutine to send it back to clients. As seen in the following code, the requests to update key value are passed onto a single shared request channel.

```
func (s *server) putKv(w http.ResponseWriter, r *http.Request)  {
  kv, err := s.readRequest(r, w)
  if err != nil {
   log.Panic(err)
   return
  }
  request := &requestResponse{
   request:  kv,
   responseChannel: make(chan string),
  }

  s.requestChannel <- request
  response := s.waitForResponse(request)
  w.Write([]byte(response))
 }
```

The requests are processed in a single goroutine to update all the state.

```
func (s* server) Start() error {
  go s.serveHttp()

  go s.singularUpdateQueue()

  return nil
}

func (s *server) singularUpdateQueue() {
  for {
   select {
   case e := <-s.requestChannel:
    s.updateState(e)
    e.responseChannel <- buildResponse(e);
  }
 }
}
```

Backpressure

Backpressure can be an important concern when a work queue is used to communicate between threads. In case the consumer is slow and the producer is fast, the queue might fill up fast. Unless some precautions are taken, it might run out of memory with a large number of tasks filling up the queue. Generally, the queue is kept bounded with sender blocking if the queue is full. For example, java.util.concurrent.ArrayBlockingQueue has two methods to add elements. put method blocks the producer if the array is full. add method throws IllegalStateException if queue is full, but doesn’t block the producer. It’s important to know the semantics of the methods available for adding tasks to the queue. In the case of ArrayBlockingQueue, put method should be used to block the sender and provide backpressure by blocking. Frameworks like reactive-streams can help implement a more sophisticated backpressure mechanism from consumer to the producer.

Other Considerations

Most of the time the processing needs to be done with chaining multiple tasks together. The results of a SingularUpdateQueue execution need to be passed to other stages. e.g. As shown in the WalRequestConsumer above, after the records are written to the write ahead log, the response needs to be sent over the socket connection. This can be done by executing the future returned by SingularUpdateQueue on a separate thread. It can submit the task to other SingularUpdateQueue as well.

Sometimes, as part of the task execution in the SingularUpdateQueue, external service calls need to be made and the state of the SingularUpdateQueue is updated by the response of the service call. It’s important in this scenario that no blocking network calls are made or it blocks the only thread which is processing all the tasks. The calls are made asynchronously. Care must be taken to not access the SingularUpdateQueue state in the future callback of the asynchronous service call because this can happen in a separate thread, defeating the purpose of doing all state changes in SingularUpdateQueue by single thread. The result of the call should be added to the work queue similar to other events or requests.

- Task Chaining.
- Making External Service Calls.

Examples

All the consensus implementations like Zookeeper(ZAB) or etcd (RAFT) need requests to be processed in strict order, one at a time. They use a similar code structure.

- The Zookeeper implementation of request processing pipeline [https://github.com/apache/zookeeper/blob/master/zookeeper-server/src/main/java/org/apache/zookeeper/server/SyncRequestProcessor.java] is done with single threaded request processors
- Controller [https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Redesign] in Apache Kafka, which needs to update state based on to multiple concurrent events from zookeeper, handles them in a single thread with all the event handlers submitting the events in a queue
- Cassandra [https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/concurrent/Stage.java], which uses SEDA [bib-seda] architecture, uses single threaded stages to update its Gossip state.
- Etcd [https://github.com/etcd-io/etcd/blob/master/etcdserver/raft.go] and other go-lang based implementation have a single goroutine working off a request channel to update its state
- LMAX-Diruptor architecture [bib-lmax] follows Single Writer Principle [bib-single-writer] to avoid mutual exclusion while updating local state.

Chapter 14. Request Waiting List

Track client requests which require responses after the criteria to respond is met based on responses from other cluster nodes.

Problem

A cluster node needs to communicate with other cluster nodes to replicate data while processing a client request. A response from all other cluster nodes or a *Quorum* is needed before responding to clients.

Communication to other cluster nodes is done asynchronously. Asynchronous communication allows patterns like *Request Pipeline* and *Request Batch* to be used.

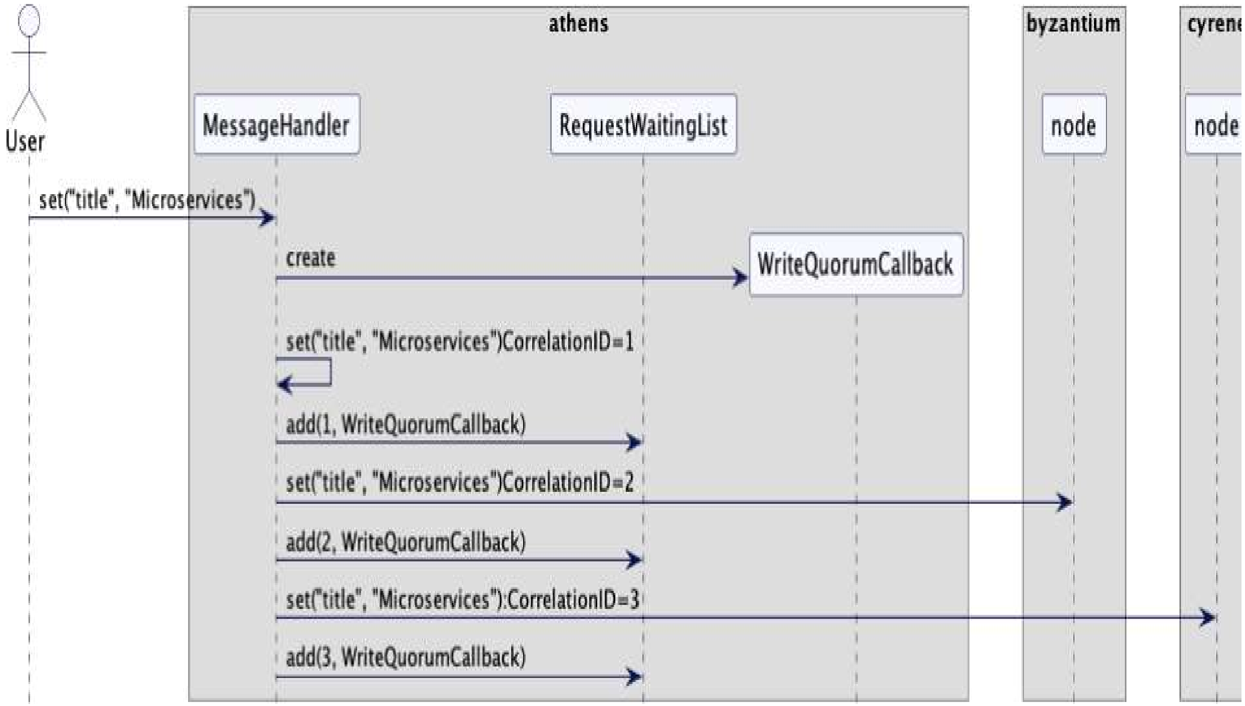
So the cluster node receives and processes responses from multiple other cluster nodes asynchronously. It then needs to correlate them to check if the Quorum for a particular client request is reached.

Solution

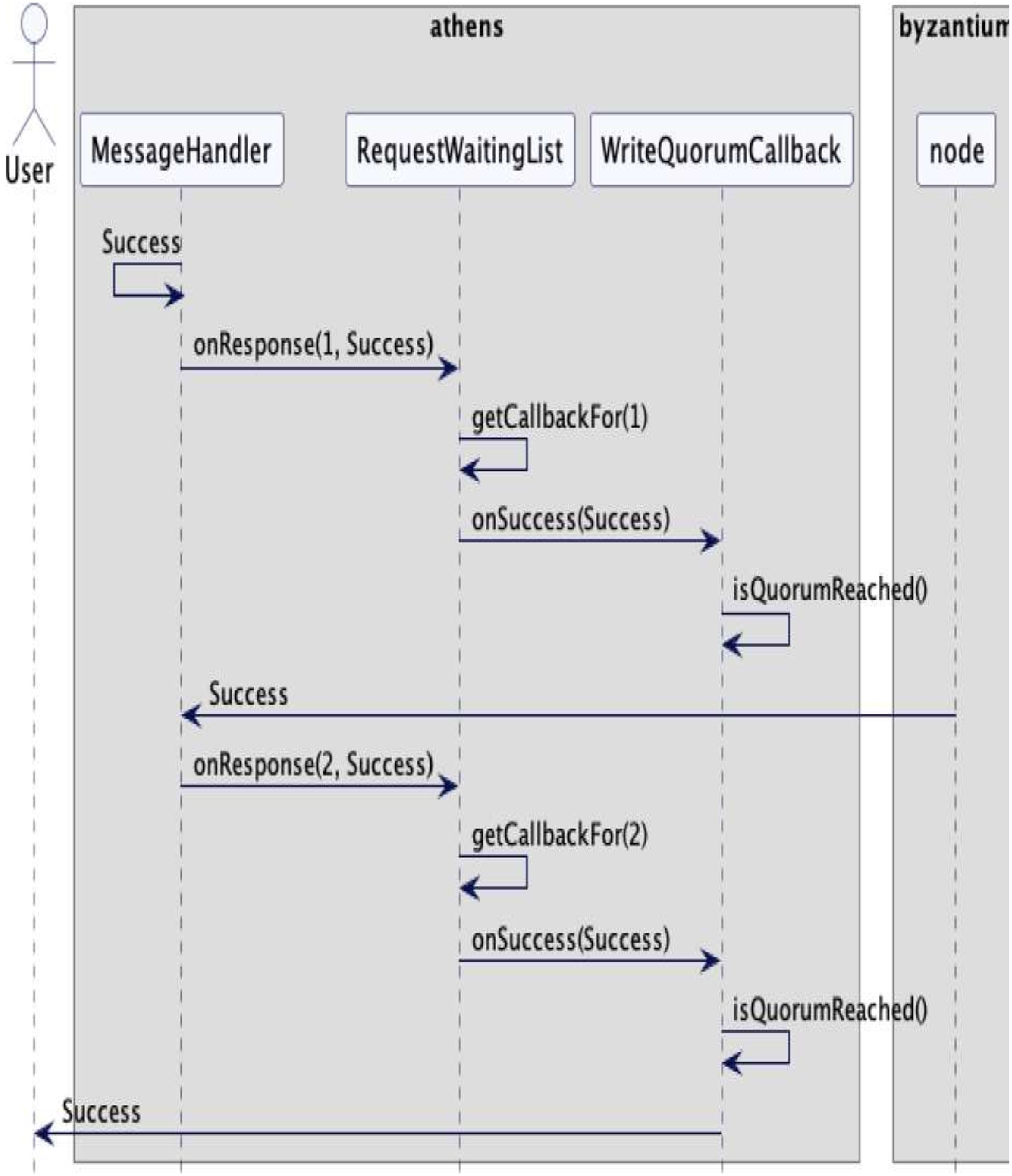
The cluster node maintains a waiting list which maps a key and a callback function. The key is chosen depending on the specific criteria to invoke the callback. For example if it needs to be invoked whenever a message from other cluster node is received, it can be the Correlation Identifer [bib-correlation-id] of the message. In the case of *Replicated Log* it is the *High-Water Mark*. The callback handles the response and decides if the client request can be fulfilled.

Consider the example of a key-value store where, data is replicated on multiple servers. Here, Quorum can be used to decide when a replication can be considered successful to initiate a response to the client. The cluster node then tracks the requests sent to other cluster nodes, and a callback is registered with each request. Each request is marked with a Correlation Identifer [bib-correlation-id], which is used to map response to the request. The waiting list is then notified to invoke the callback when the response from other cluster nodes are received.

For the sake of this example, let’s call our three cluster nodes athens, byzantium and cyrene. The client connects with athens to store "title" as "Microservices". Athens replicates it on byzantium and cyrene; so it sends a request to itself to store the key-value and sends requests to both byzantium and cyrene concurrently. To track responses, athens creates a WriteQuorumResponseCallback and adds it to the waiting list for each of the requests sent.


User
athens
byzantium
cyrene
MessageHandler
RequestWaitingList
node
node
set("title", "Microservices")
create
WriteQuorumCallback
set("title", "Microservices")CorrelationID=1
add(1, WriteQuorumCallback)
set("title", "Microservices")CorrelationID=2
add(2, WriteQuorumCallback)
set("title", "Microservices"):CorrelationID=3
add(3, WriteQuorumCallback)

For every response received, the WriteQuorumResponseCallback is invoked to handle the response. It checks whether the required number of responses have been received. Once the response is received from byzantium, the quorum is reached and the pending client request is completed. Cyrene can respond later, but the response can be sent to the client without waiting for it.



The code looks like the sample below: Note that every cluster node maintains its own instance of a waiting list. The waiting list tracks the key and associated callback and stores the timestamp at which the callback was registered. The timestamp is used to check whether the callbacks need to be expired if responses haven't been received within the expected time.

```
public class RequestWaitingList<Key, Response> {
  private Map<Key, CallbackDetails> pendingRequests = new Concurren
  public void add(Key key, RequestCallback<Response> callback) {
    pendingRequests.put(key, new CallbackDetails(callback, clock.na
  }

class CallbackDetails {
  RequestCallback requestCallback;
  long createTime;

  public CallbackDetails(RequestCallback requestCallback, long crea
    this.requestCallback = requestCallback;
    this.createTime = createTime;
  }

  public RequestCallback getRequestCallback() {
  return requestCallback;
  }

  public long elapsedTime(long now) {
    return now - createTime;
  }
 }

 public interface RequestCallback<T> {
   void onResponse(T r);
   void onError(Throwable e);
 }
```

It is asked to handle the response or error once the response has been received from the other cluster node.

class RequestWaitingList…

```
public void handleResponse(Key key, Response response) {
  if (!pendingRequests.containsKey(key)) {
     return;
  }
  CallbackDetails callbackDetails = pendingRequests.remove(key);
  callbackDetails.getRequestCallback().onResponse(response);
}
```

class RequestWaitingList…

```
public void handleError(int requestId, Throwable e) {
  CallbackDetails callbackDetails = pendingRequests.remove(requestI
  callbackDetails.getRequestCallback().onError(e);
}
```

The waiting list can then be used to handle quorum responses with the implementation looking something like this:

```
static class WriteQuorumCallback implements RequestCallback<Request
  private final int quorum;
  private volatile int expectedNumberOfResponses;
  private volatile int receivedResponses;
  private volatile int receivedErrors;
  private volatile boolean done;

  private final RequestOrResponse request;
  private final ClientConnection clientConnection;

  public WriteQuorumCallback(int totalExpectedResponses, RequestOrR
    this.expectedNumberOfResponses = totalExpectedResponses;
    this.quorum = expectedNumberOfResponses / 2 + 1;
    this.request = clientRequest;
    this.clientConnection = clientConnection;
  }

  @Override
  public void onResponse(RequestOrResponse response) {
    receivedResponses++;
    if (receivedResponses == quorum && !done) {
       respondToClient("Success");
       done = true;
    }
  }

  @Override
  public void onError(Throwable t) {
    receivedErrors++;
    if (receivedErrors == quorum && !done) {
       respondToClient("Error");
       done = true;
    }
  }

  private void respondToClient(String response) {
    clientConnection.write(new RequestOrResponse(RequestId.SetValue
  }
}
```

Whenever a cluster node sends requests to other nodes, it adds a callback to the waiting list mapping with the Correlation Identifer [bib-correlation-id] of the request sent.

class ClusterNode...

```
private void handleSetValueClientRequestRequiringQuorum(List<InetAd
  int totalExpectedResponses = replicas.size();
  RequestCallback requestCallback = new WriteQuorumCallback(totalEx
  for (InetAddressAndPort replica : replicas) {
    int correlationId = nextRequestId();
    requestWaitingList.add(correlationId, requestCallback);
    try {
       SocketClient client = new SocketClient(replica);
       client.sendOneway(new RequestOrResponse(RequestId.SetValueRe
   } catch (IOException e) {
     requestWaitingList.handleError(correlationId, e);
   }
  }
}
```

Once the response is received, the waiting list is asked to handle it:

class ClusterNode…

```
private void handleSetValueResponse(RequestOrResponse response) {
  requestWaitingList.handleResponse(response.getCorrelationId(), re
}
```

The waiting list will then invoke the associated WriteQuorumCallback. The WriteQuorumCallback instance verifies if the quorum responses have been received and invokes the callback to respond to the client.

Expiring Long Pending Requests

Sometimes, responses from the other cluster nodes are delayed. In these instances the waiting list generally has a mechanism to expire requests after a timeout:

class RequestWaitingList…

```
private SystemClock clock;
private ScheduledExecutorService executor = Executors.newSingleThre
private long expirationIntervalMillis = 2000;
public RequestWaitingList(SystemClock clock) {
  this.clock = clock;
  executor.scheduleWithFixedDelay(this::expire, expirationIntervalM
}

private void expire() {
  long now = clock.nanoTime();
  List<Key> expiredRequestKeys = getExpiredRequestKeys(now);
  expiredRequestKeys.stream().forEach(expiredRequestKey -> {
    CallbackDetails request = pendingRequests.remove(expiredRequest
    request.requestCallback.onError(new TimeoutException("Request e
  });
}
```

```
private List<Key> getExpiredRequestKeys(long now) {
  return pendingRequests.entrySet().stream().filter(entry -> entry.
}
```

Examples

Cassandra [bib-cassandra] uses asynchronous message passing for internode communication. It uses *Quorum* and processes response messages asynchronously the same way.

Kafka [bib-kafka] tracks the pending requests using a data structure called [kafka-purgatory] [bib-kafka-purgatory].

etcd [bib-etcd] maintains a wait list to respond to client requests in a similar way [bib-etcd-wait].

Chapter 15. Idempotent Receiver

Identify requests from clients uniquely so they can ignore duplicate requests when client retries

Problem

Clients send requests to servers but might not get a response. It’s impossible for clients to know if the response was lost or the server crashed before processing the request. To make sure that the request is processed, the client has to re-send the request.

If the server had already processed the request and crashed after that servers will get duplicate requests from clients, when the client retries.

Solution

At-most once, At-least once and Exactly Once actions

Depending on how the client interacts with the server, the guarantee of whether the server will do certain action is predetermined. If a client experiences a failure after the request is sent, and before receiving the response, there can be three possibilities.

If the client doesn’t retry the request in case of failure, the server might have processed the request, or might have failed before processing the request. So the request is processed at the most once on the server.

If the client retries the request, and the server had processed it before communication failure, it might process it again. So the request is processed at least once, but can be processed multiple times on the server.

With idempotent receiver, even with multiple client retries, the server processes the request only once. So to achieve ‘exactly once’ actions, its important to have idempotent receivers.

Identify a client uniquely by assigning a unique id to each client. Before sending any requests, the client registers itself with the server.

class ConsistentCoreClient…

```
private void registerWithLeader() {
  RequestOrResponse request
     = new RequestOrResponse(RequestId.RegisterClientRequest.getId
     correlationId.incrementAndGet());

  //blockingSend will attempt to create a new connection if there
  RequestOrResponse response = blockingSend(request);
  RegisterClientResponse registerClientResponse
      = JsonSerDes.deserialize(response.getMessageBodyJson(),
      RegisterClientResponse.class);
 this.clientId = registerClientResponse.getClientId();
}
```

When the server receives a client registration request, it assigns a unique id to the client. If the server is a *Consistent Core*, it can assign the *Write-Ahead Log* index as a client identifier.

class ReplicatedKVStore…

```
private Map<Long, Session> clientSessions = new ConcurrentHashMap<


private RegisterClientResponse registerClient(WALEntry walEntry) {
  Long clientId = walEntry.getEntryIndex();
  //clientId to store client responses.
  clientSessions.put(clientId, new Session(clock.nanoTime()));

  return new RegisterClientResponse(clientId);
}
```

The server creates a session to store responses for the requests for the registered client. It also tracks the time at which the session is created, so that inactive sessions can be discarded as explained in later sections.

```
public class Session {
  long lastAccessTimestamp;
  Queue<Response> clientResponses = new ArrayDeque<>();

  public Session(long lastAccessTimestamp) {
    this.lastAccessTimestamp = lastAccessTimestamp;
  }

  public long getLastAccessTimestamp() {
    return lastAccessTimestamp;
  }

  public Optional<Response> getResponse(int requestNumber) {
    return clientResponses.stream().
         filter(r -> requestNumber == r.getRequestNumber()).findFi

  }

  private static final int MAX_SAVED_RESPONSES = 5;
  public void addResponse(Response response) {
    if (clientResponses.size() == MAX_SAVED_RESPONSES) {
       clientResponses.remove(); //remove the oldest request
    }
    clientResponses.add(response);
  }
  public void refresh(long nanoTime) {
    this.lastAccessTimestamp = nanoTime;
  }
}
```

For a Consistent Core, the client registration request is also replicated as part of the consensus algorithm. So the client registration is available even if the existing leader fails. The server then also stores responses sent to the client for subsequent requests.

Idempotent and Non-Idempotent requests

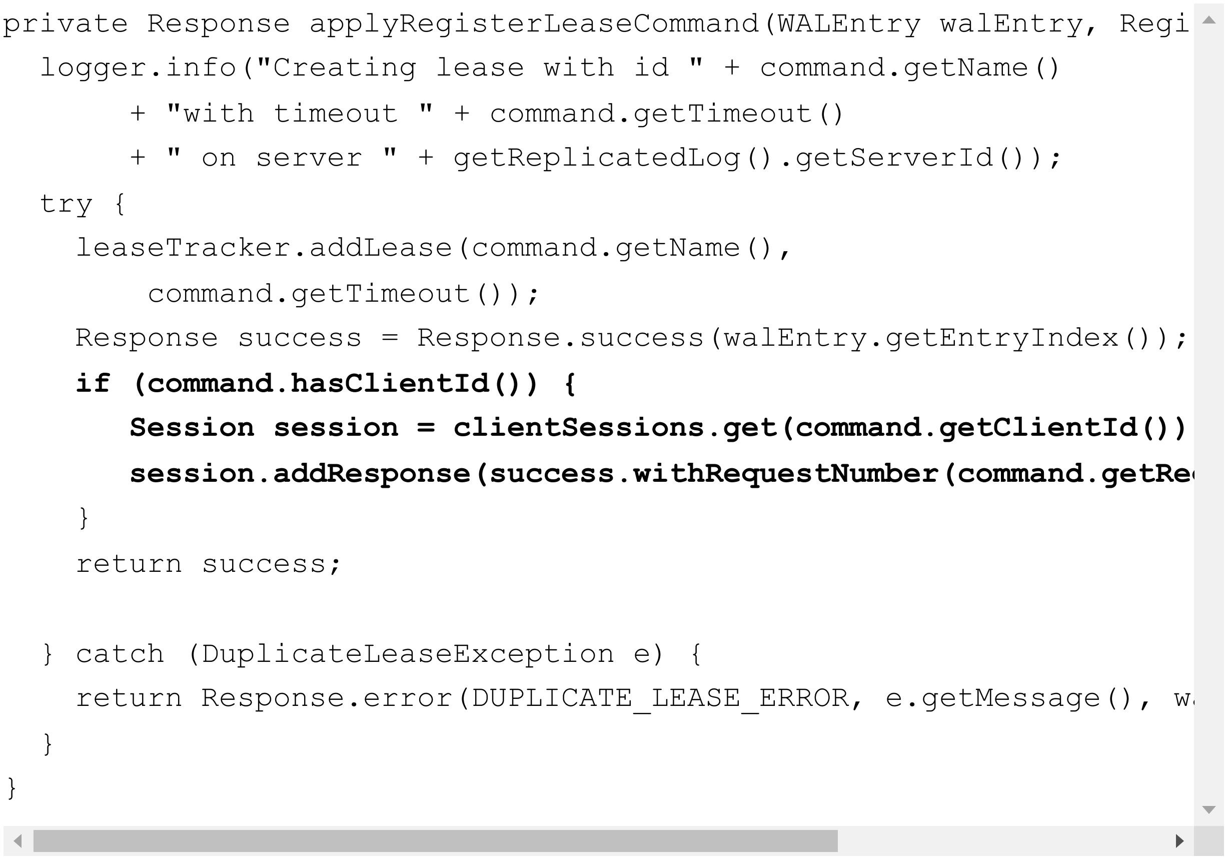
It is important to note that some of the requests are by nature idempotent. For example, setting a key and a value in a key-value store, is naturally idempotent. Even if the same key and value is set multiple times, it doesn’t create a problem.

On the other hand, creating a *Lease* is not idempotent. If a lease is already created, a retried request to create a lease will fail. This is a problem. Consider the following scenario. A client sends a request to create a lease; the server creates a lease successfully, but then crashes, or the connection fails before the response is sent to the client. The client creates the connection again, and retries creating the lease; because the server already has a lease with the given name, it returns an error. So the client thinks that it doesn’t have a lease. This is clearly not the behaviour we expect to have.

With idempotent receiver, the client will send the lease request with the same request number. Because the response from the already processed request is saved on the server, the same response is returned. This way, if the client could successfully create a lease before the connection failed, it will get the response after it retries the same request.

For every non-idempotent request (see sidebar) that the server receives, it stores the response in the client session after successful execution.

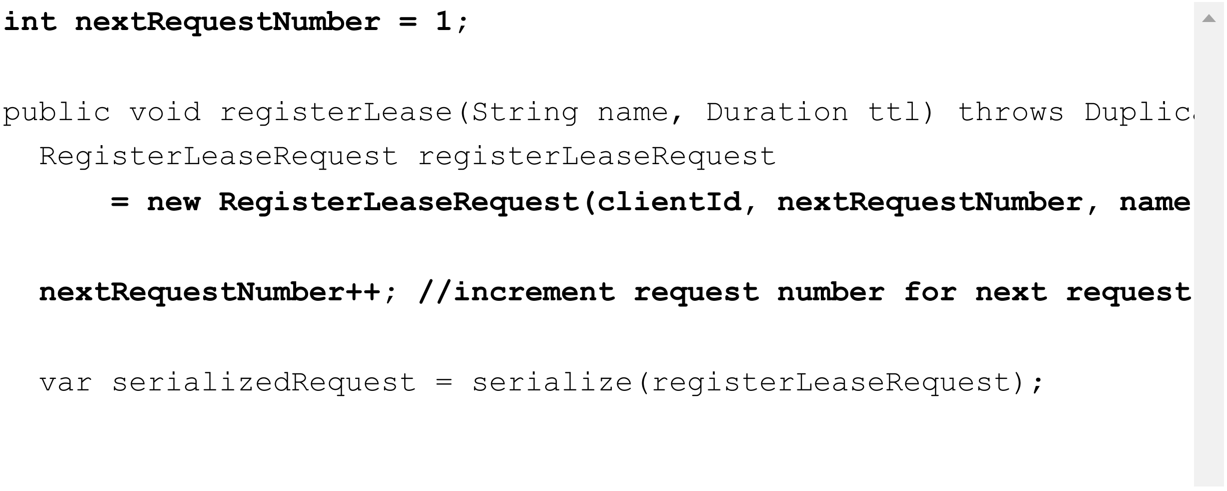
class ReplicatedKVStore…



```
private Response applyRegisterLeaseCommand(WALEntry walEntry, Regi
  logger.info("Creating lease with id " + command.getName()
       + "with timeout " + command.getTimeout()
       + " on server " + getReplicatedLog().getServerId());
  try {
    leaseTracker.addLease(command.getName(),
        command.getTimeout());
    Response success = Response.success(walEntry.getEntryIndex());
    if (command.hasClientId()) {
       Session session = clientSessions.get(command.getClientId())
       session.addResponse(success.withRequestNumber(command.getRe
    }
    return success;

  } catch (DuplicateLeaseException e) {
    return Response.error(DUPLICATE_LEASE_ERROR, e.getMessage(), w
  }
}
```

The client sends the client identifier with each request that is sent to the server. The client also keeps a counter to assign request numbers to each request sent to the server.

class ConsistentCoreClient…



```
int nextRequestNumber = 1;

public void registerLease(String name, Duration ttl) throws Duplic
  RegisterLeaseRequest registerLeaseRequest
      = new RegisterLeaseRequest(clientId, nextRequestNumber, name

  nextRequestNumber++; //increment request number for next request

  var serializedRequest = serialize(registerLeaseRequest);
  logger.info("Sending RegisterLeaseRequest for " + name);
  RequestOrResponse requestOrResponse = blockingSendWithRetries(se
  Response response = JsonSerDes.deserialize(requestOrResponse.getI
  if (response.error == Errors.DUPLICATE_LEASE_ERROR) {
     throw new DuplicateLeaseException(name);
  }

}

private static final int MAX_RETRIES = 3;

private RequestOrResponse blockingSendWithRetries(RequestOrRespons
  for (int i = 0; i <= MAX_RETRIES; i++) {
    try {
      //blockingSend will attempt to create a new connection is th
      return blockingSend(request);

    } catch (NetworkException e) {
      resetConnectionToLeader();
      logger.error("Failed sending request  " + request + ". Try "
    }
  }

  throw new NetworkException("Timed out after " + MAX_RETRIES + "
}
```

When the server receives a request, it checks if the request with the given request number from the same client is already processed. If it finds the saved response, it returns the same response to the client, without processing the request again.

class ReplicatedKVStore…

```
private Response applyWalEntry(WALEntry walEntry) {
  Command command = deserialize(walEntry);
  if (command.hasClientId()) {
     Session session = clientSessions.get(command.getClientId());
```

```
    Optional<Response> savedResponse = session.getResponse(comman
    if(savedResponse.isPresent()) {
      return savedResponse.get();
    } //else continue and execute this command.
  }
```

Expiring the saved client requests

The requests stored per client cannot be stored forever. There are multiple ways the requests can be expired. In the reference implementation [bib-logcabin-raft] for Raft, the client keeps a separate number to note the request number for which the response is successfully received. This number is then sent with each request to the server. The server can safely discard any requests with request number less than this number.

If a client is guaranteed to send the next request only after receiving the response for the previous request, the server can safely remove all previous requests once it gets a new request from the client. There is a problem when *Request Pipeline* is used, as there can be multiple in-flight requests for which client might not have received the response. If the server knows the maximum number of in-flight requests a client can have, it can store only those many responses, and remove all the other responses. For example, Kafka [bib-kafka] can have a maximum of five in-flight requests for its producer, so it stores a maximum of five previous responses.

class Session…

```
private static final int MAX_SAVED_RESPONSES = 5;

public void addResponse(Response response) {
  if (clientResponses.size()== MAX_SAVED_RESPONSES) {
      clientResponses.remove(); //remove the oldest request
  }
  clientResponses.add(response);
}
```

Removing the registered clients

> It is important to note that this mechanism to detect duplicate messages is only applicable for client retries on connection failures. If a client fails and is restarted, it will be registered again, so no deduplication is achieved across client restarts.
>
> It is also not aware of any application level logic. So if an application sends multiple requests, which are considered duplicate at the application-level, there is no way for the storage server implementation to know about it. The application needs to handle it independently.

The client’s session is not kept on the server forever. A server can have maximum time to live for the client sessions it stores. Clients send a *HeartBeat* periodically. If there are no HeartBeats from the client during this time to live, the client’s state on the server can be removed.

The server starts a scheduled task to periodically check for expired sessions and remove the sessions which are expired.

class ReplicatedKVStore…

```
private long heartBeatIntervalMs = TimeUnit.SECONDS.toMillis(10);
private long sessionTimeoutNanos = TimeUnit.MINUTES.toNanos(5);

private void startSessionCheckerTask() {
  scheduledTask = executor.scheduleWithFixedDelay(()->{
    removeExpiredSession();
  }, heartBeatIntervalMs, heartBeatIntervalMs, TimeUnit.MILLISECON
}
private void removeExpiredSession() {
  long now = System.nanoTime();
  for (Long clientId : clientSessions.keySet()) {
    Session session = clientSessions.get(clientId);
    long elapsedNanosSinceLastAccess = now - session.getLastAccess
    if (elapsedNanosSinceLastAccess > sessionTimeoutNanos) {
       clientSessions.remove(clientId);
    }
  }
}
```

Examples

Raft [bib-raft] has reference implementation to have idempotency for providing linearizable actions.

Kafka [bib-kafka] allows Idempotent Producer [bib-kafka-idempotent-producer] which allows clients to retry requests and ignores duplicate requests.

Zookeeper [bib-zookeeper] has the concept of Sessions, and zxid, which allows clients to recover. Hbase has a [hbase-recoverable-zookeeper] [bib-hbase-recoverable-zookeeper] wrapper, which implements idempotent actions following the guidelines of [zookeeper-error-handling] [bib-zookeeper-error-handling]

Chapter 16. Follower Reads

Serve read requests from followers to achieve better throughput and lower latency

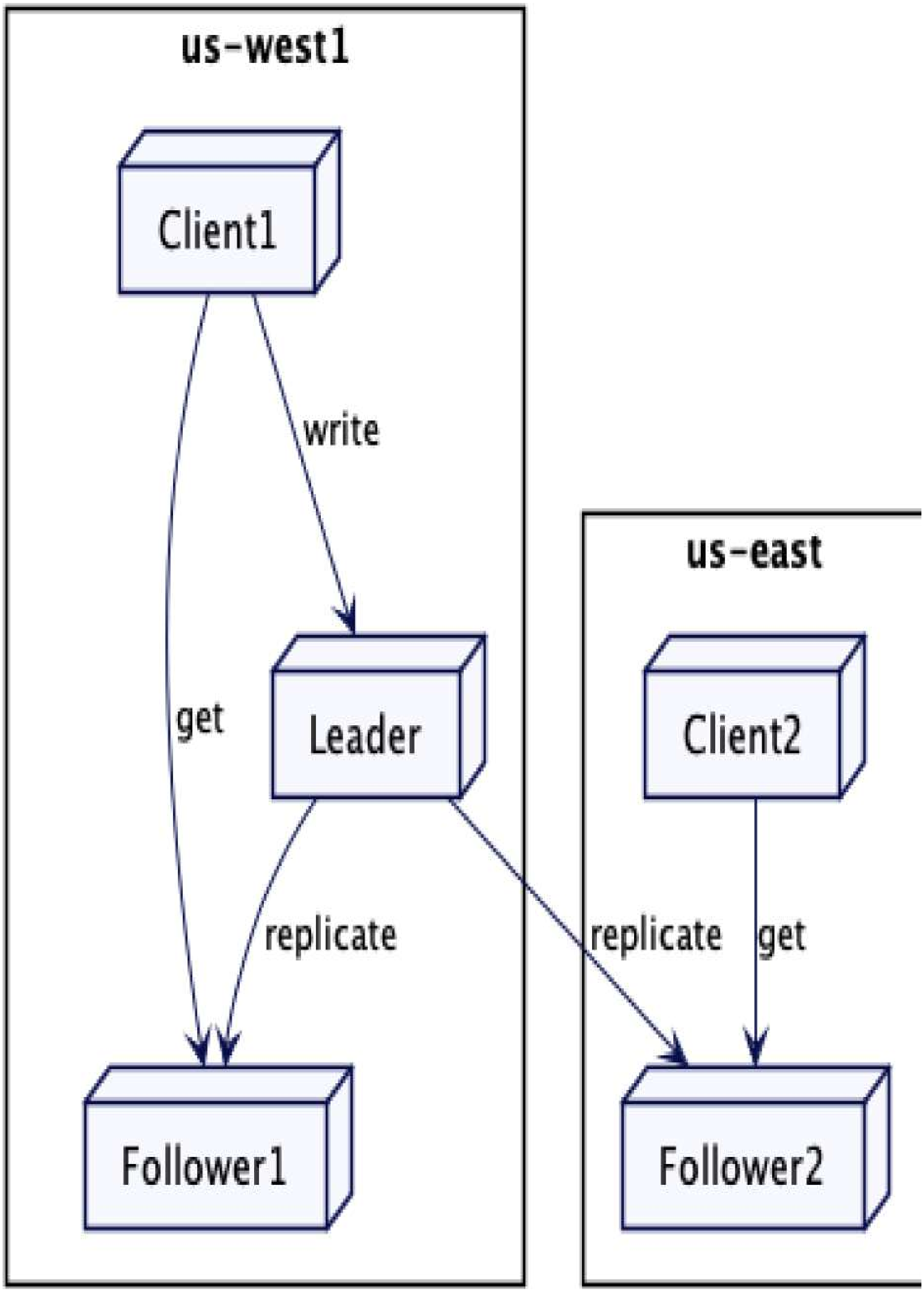
Problem

When using the *Leader and Followers* pattern, it's possible that the leader may get overloaded if too many requests are sent to it. Furthermore in a multi-datacenter setup, where the client is in a remote datacenter, requests to the leader will be subject to additional latency.

Solution

While the write requests need to go to the leader to maintain consistency, the read-only requests can instead go to the nearest follower. This is particularly useful when clients are mostly read-only.

It is important to note that clients reading from followers can get old values. There will always be a replication lag between the leader and the follower, even in the systems which implement consensus algorithms like Raft [bib-raft]. That's because even when the leader knows about which values are committed, it needs another message to communicate it to the follower. So reading from follower servers is used only in situations where slightly older values are tolerated.


us-west1
Client1
write
get
Leader
replicate
Follower1
us-east
Client2
replicate
get
Follower2

Finding The Nearest Replica

Cluster nodes maintain additional metadata about their location.

class ReplicaDescriptor…

```
public class ReplicaDescriptor {
  public ReplicaDescriptor(InetAddressAndPort address, String regio
    this.address = address;
    this.region = region;
  }
  InetAddressAndPort address;
  String region;

  public InetAddressAndPort getAddress() {
    return address;
  }

  public String getRegion() {
    return region;
  }
}
```

The cluster client can then pick up the local replica based its own region.

class ClusterClient…

```
public List<String> get(String key) {
  List<ReplicaDescriptor> allReplicas = allFollowerReplicas(key);
  ReplicaDescriptor nearestFollower = findNearestFollowerBasedOnLoc
  GetValueResponse getValueResponse = sendGetRequest(nearestFollowe
  return getValueResponse.getValue();
}

ReplicaDescriptor findNearestFollowerBasedOnLocality(List<ReplicaDe
  List<ReplicaDescriptor> sameRegionFollowers = matchLocality(follo
  List<ReplicaDescriptor> finalList = sameRegionFollowers.isEmpty()
  return finalList.get(0);
}

private List<ReplicaDescriptor> matchLocality(List<ReplicaDescripto
  return followers.stream().filter(rd -> clientRegion.equals(rd.reg
}
```

For example, if there are two follower replicas, one in the region us-west and the other in the region us-east. The client from us-east region, will be connected to the us-east replica.

class CausalKVStoreTest...

```
@Test
public void getFollowersInSameRegion() {
  List<ReplicaDescriptor> followers = createReplicas("us-west", "us
  ReplicaDescriptor nearestFollower = new ClusterClient(followers,
  assertEquals(nearestFollower.getRegion(), "us-east");

}
```

The cluster client or a co-ordinating cluster node can also track latencies observed with cluster nodes. It can send period heartbeats to capture the latencies, and use that to pick up a follower with minimum latency. To do a more fair selection, products like MongoDB [bib-mongodb] or CockroachDB [bib-cockroachdb] calculate latencies as a moving average [bib-moving-average]. Cluster nodes generally maintain a *Single Socket Channel* to communicate with other cluster nodes. Single Socket Channel needs a *HeartBeat* to keep the connection alive. So capturing latencies and calculating the moving average can be easily implemented.

class WeightedAverage...

```
public class WeightedAverage {
  long averageLatencyMs = 0;
  public void update(long heartbeatRequestLatency) {
    //Example implementation of weighted average as used in Mongodb
    //The running, weighted average round trip time for heartbeat m
    // Weighted 80% to the old round trip time, and 20% to the new
    averageLatencyMs = averageLatencyMs == 0
         ? heartbeatRequestLatency
         : (averageLatencyMs * 4 + heartbeatRequestLatency) / 5;
 }

 public long getAverageLatency() {
   return averageLatencyMs;
 }
}
```

class ClusterClient…

```
private Map<InetAddressAndPort, WeightedAverage> latencyMap = new H
private void sendHeartbeat(InetAddressAndPort clusterNodeAddress) {
  try {
    long startTimeNanos = System.nanoTime();
    sendHeartbeatRequest(clusterNodeAddress);
    long endTimeNanos = System.nanoTime();

    WeightedAverage heartbeatStats = latencyMap.get(clusterNodeAddr
    if (heartbeatStats == null) {
       heartbeatStats = new WeightedAverage();
       latencyMap.put(clusterNodeAddress, new WeightedAverage());
    }
    heartbeatStats.update(endTimeNanos - startTimeNanos);

  } catch (NetworkException e) {
    logger.error(e);
  }
}
```

This latency information can then be used to pick up the follower with the least network latency.

class ClusterClient…

```
ReplicaDescriptor findNearestFollower(List<ReplicaDescriptor> allFo
  List<ReplicaDescriptor> sameRegionFollowers = matchLocality(allFo
  List<ReplicaDescriptor> finalList
        = sameRegionFollowers.isEmpty() ? allFollowers
                        :sameRegionFollowers;
  return finalList.stream().sorted((r1, r2) -> {
    if (!latenciesAvailableFor(r1, r2)) {
       return 0;
    }
    return Long.compare(latencyMap.get(r1).getAverageLatency(),
                latencyMap.get(r2).getAverageLatency());
  }).findFirst().get();
}
private boolean latenciesAvailableFor(ReplicaDescriptor r1, Replica
  return latencyMap.containsKey(r1) && latencyMap.containsKey(r2);
}
```

Disconnected Or Slow Followers

A follower might get disconnected from the leader and stop getting updates. In some cases, followers can suffer with slow disks impeding the overall replication process, which causes the follower to lag behind the leader. Followers can track if it has not heard from the leader in a while, and stop serving user requests.

For example, products like MongoDB [bib-mongodb] allow selecting a replica with a maximum allowed lag time. [bib-mongodb-max-staleness] If the replica lags behind the leader beyond this maximum time, it’s not selected to serve the requests. In Kafka [bib-kafka] if the follower detects the offset asked by the consumer is too large, it responds with OFFSET_OUT_OF_RANGE error. The consumer is then expected to communicate with the leader [bib-kafka-follower-fetch].

Read Your Own Writes

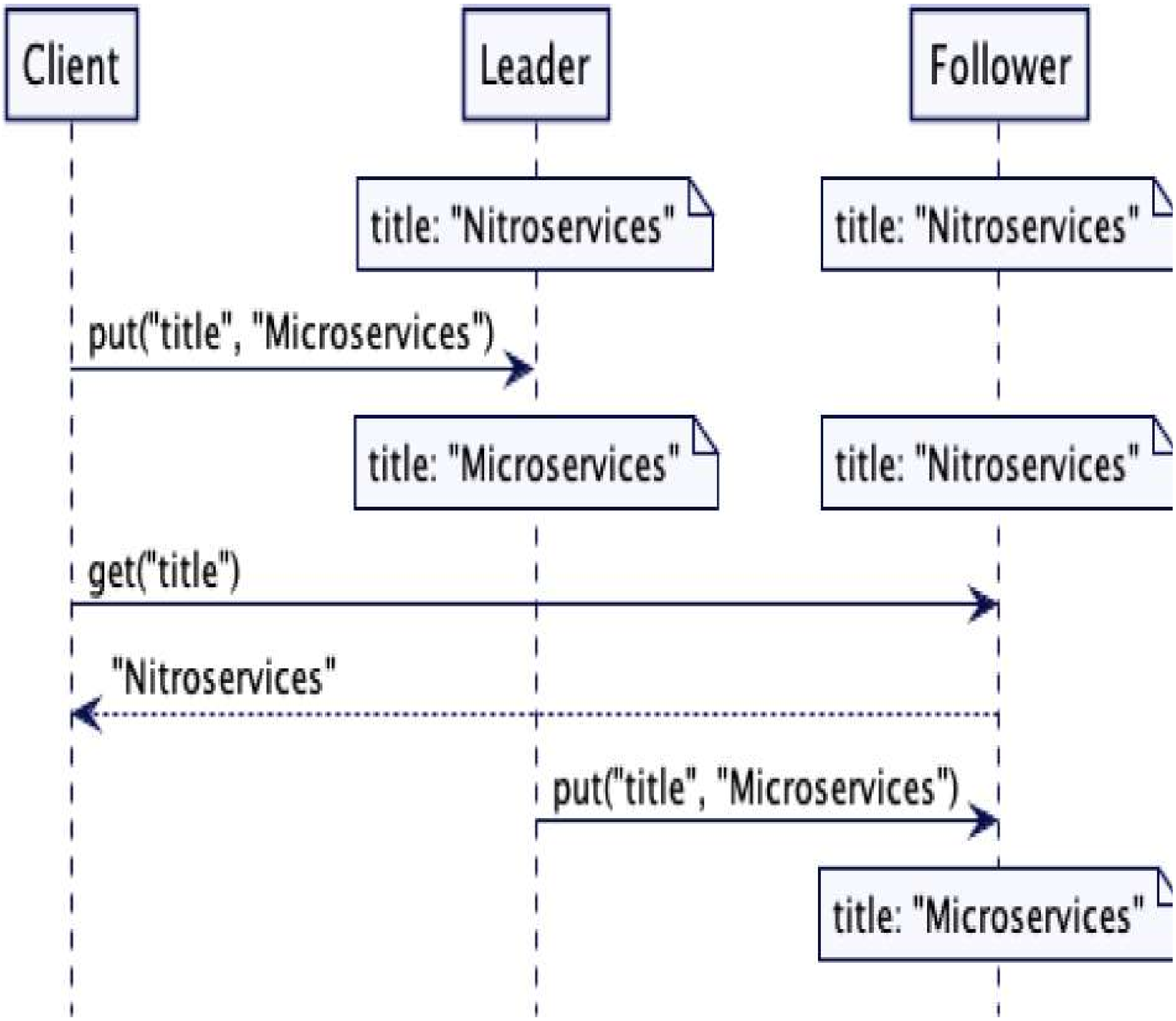
Causal Consistency

When an event A in a system happens before another event B, it is said to have causal relationship. This causal relationship means that A might have some role in causing B.

For a data storage system, the events are about writing and reading values. To provide causal consistency, the storage system needs to track happens-before relationship between read and write events. *Lamport Clock* and its variants are used for this purpose.

Reading from the follower servers can be problematic, as it can give surprising results in common scenarios where a client writes something and then immediately tries to read it.

Consider a client who notices that some book data erroneously has "title": "Nitroservices". It corrects this by a write, "title": "Microservices", which goes to the leader. It then immediately reads back the value but the read request goes to a follower, which may not have been updated yet.

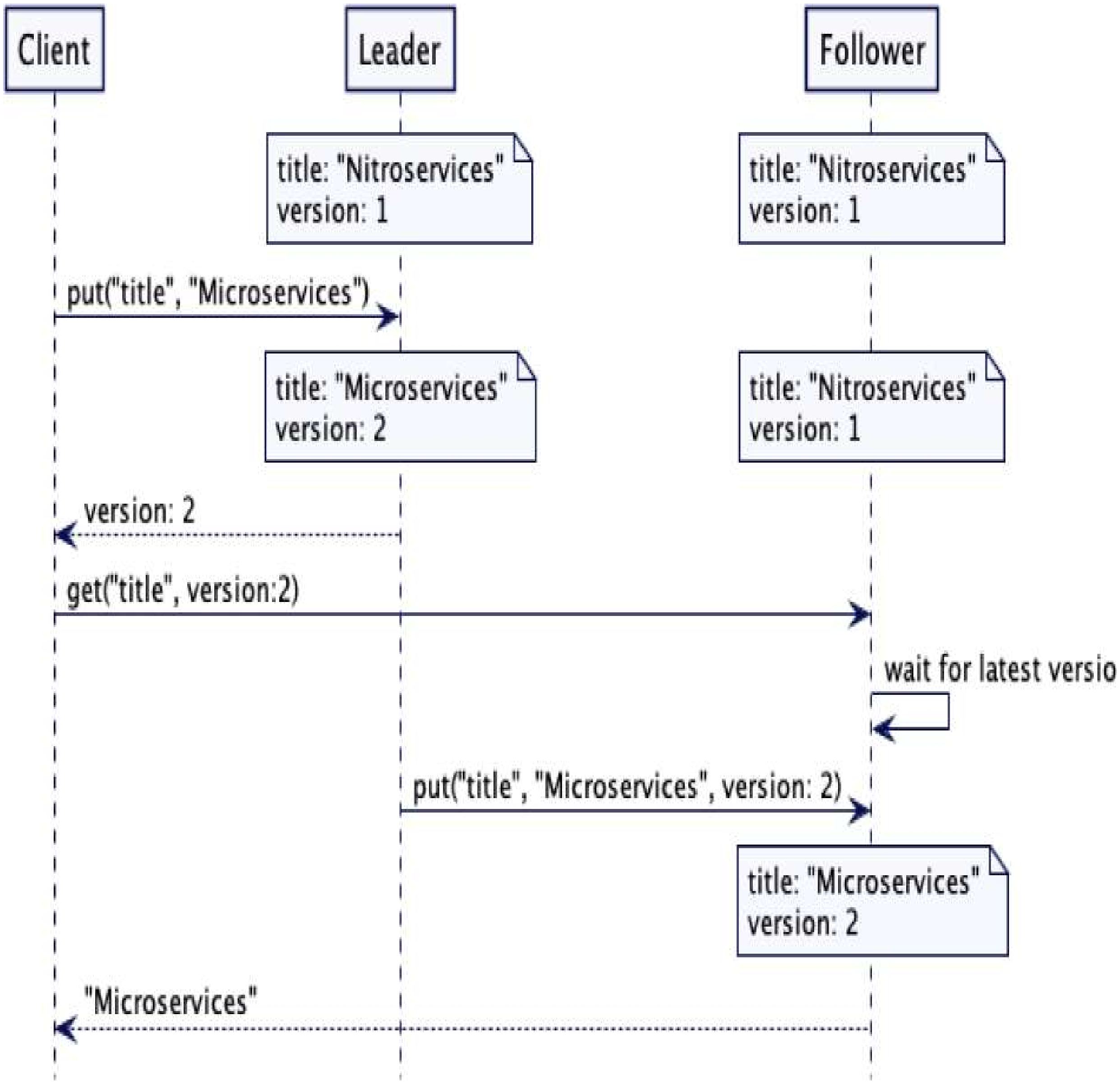


This can be a common problem. For example, untill very recently [bib-aws-strong-consistency] Amazon S3 did not prevent this.

To fix this issue, with each write, the server stores not just the new value, but also a monotonically increasing version stamp. The stamp can be a *High-Water Mark* or a *Hybrid Clock*. The server returns this version stamp of the stored value in the response to the write request. Then, should the client wish to read the value later, it includes the version stamp as part of its read request. Should the read request go to a follower, it checks its stored value to see if it is equal or later than the requested version stamp. If it isn’t, it waits until it has an up-to-date version before returning the value. By doing this clients will always read a value that’s consistent with a value they write - which is often referred to as *read-your-writes consistency*.

(TODO: Is this a usage of ?)

The flow of requests happens as shown below: To correct a wrongly written value, "title": "Microservices" is written to the leader. The leader returns version 2, to the client in the response. When the client tries to read the value for "title", it passes the version number 2 in the request. The follower server which receives the request checks if its own version number is up-to-date. Because the version number at the follower server is still 1, it waits till it gets that version from the leader. Once it has the matching (or later) version, it completes the read request, and returns the value "Microservices".



The code for the key value store looks as follows. It is important to note that the follower can be lagging behind too much or be disconnected from the leader. So it does not wait indefinitely. There is a configured timeout value. If the follower server can not get the updates within timeout, an error response is returned to the client. The client can then retry reading from other followers.

class ReplicatedKVStore…

```
Map<Integer, CompletableFuture> waitingRequests = new ConcurrentHas
public CompletableFuture<Optional<String>> get(String key, int atVe
  if(this.replicatedLog.getRole() == ServerRole.FOLLOWING) {
     //check if we have the version with us;
     if (!isVersionUptoDate(atVersion)) {
       //wait till we get the latest version.
       CompletableFuture<Optional<String>> future = new Completable
       //Timeout if version does not progress to required version
       //before followerWaitTimeout ms.
       future.orTimeout(config.getFollowerWaitTimeoutMs(), TimeUnit
       waitingRequests.put(atVersion, future);
       return future;
    }
  }
  return CompletableFuture.completedFuture(mvccStore.get(key, atVer
}

private boolean isVersionUptoDate(int atVersion) {
  return version >= atVersion;
}
```

Once the key value store progresses to the version the client requested, it can send the response to the client.

class ReplicatedKVStore…

```
private Response applyWalEntry(WALEntry walEntry) {
  Command command = deserialize(walEntry);
  if (command instanceof SetValueCommand) {
     return applySetValueCommandsAndCompleteClientRequests((SetValu
  }
  throw new IllegalArgumentException("Unknown command type " + comm
}

private Response applySetValueCommandsAndCompleteClientRequests(Set
  version = version + 1;
  getLogger().info(replicatedLog.getServerId() + " Setting key valu
  mvccStore.put(new VersionedKey(setValueCommand.getKey(), version)
  completeWaitingFuturesIfFollower(version, setValueCommand.getValu
  Response response = Response.success(version);
  return response;
}

private void completeWaitingFuturesIfFollower(int version, String v
  CompletableFuture completableFuture = waitingRequests.remove(vers
  if (completableFuture != null) {
     logger.info("Completing pending requests for version " + versi
     completableFuture.complete(Optional.of(value));
  }
}
```

Linearizable Reads

Sometimes read requests need to get the latest available data. The replication lag cannot be tolerated. In these cases, the read requests need to be redirected to the leader. This is a common design issue tackled by the Consistent Core [consistentcore.xhtml#SerializabilityAndLinearizability]

Examples

[neo4j] [bib-neo4j] allows causal clusters [bib-neo4j-causal-cluster] to be set up. Every write operation returns a bookmark, which can be passed when executing queries against read replicas. The bookmark ensures that the client will always get the values written at the bookmark

MongoDB [bib-mongodb] maintains causal consistency [bib-mongodb-causal-consistency] in its replica sets. The write operations return an operationTime; this is passed in the subsequent read requests to make sure read requests return the writes which happened before the read request.

CockroachDB [bib-cockroachdb] allows clients to read from follower servers. [bib-cockroachdb-follower-read] The leader servers publish the latest timestamps at which the writes are completed on the leader, called closed timestamps. The followers allow reading the values if it has values at the closed timestamp.

Kafka [bib-kafka] allows consuming the messages from the follower brokers. [bib-kafka-follower-fetch] The followers know about the *High-Water Mark* at the leader. In kafka’s design, instead of waiting for the latest updates, the broker returns a OFFSET_NOT_AVAILABLE error to the consumers and expects consumers to retry.

Chapter 17. Versioned Value

Store every update to a value with a new version, to allow reading historical values.

Problem

In a distributed system, nodes need to be able to tell which value for a key is the most recent. Sometimes they need to know past values so they can react properly to changes in a value

Solution

Store a version number with each value. The version number is incremented for every update. This allows every update to be converted to new write without blocking a read. Clients can read historical values at a specific version number.

Consider a simple example of a replicated key value store. The leader of the cluster handles all the writes to the key value store. It saves the write requests in *Write-Ahead Log*. The Write Ahead Log is replicated using *Leader and Followers*. The Leader applies entries from the Write Ahead Log at *High-Water Mark* to the key value store. This is a standard replication method called as state machine replication [bib-state-machine-replication]. Most data systems backed by consensus algorithm like Raft are implemented this way. In this case, the key value store keeps an integer version counter. It increments the version counter every time the key value write command is applied from the Write Ahead Log. It then constructs the new key with the incremented version counter. This way no existing value is updated, but every write request keeps on appending new values to the backing store.

class ReplicatedKVStore…

```
int version = 0;
MVCCStore mvccStore = new MVCCStore();

@Override
public CompletableFuture<Response> put(String key, String value) {
  return replicatedLog.propose(new SetValueCommand(key, value));
}

private Response applySetValueCommand(SetValueCommand setValueComma
  getLogger().info("Setting key value " + setValueCommand);
  version = version + 1;
  mvccStore.put(new VersionedKey(setValueCommand.getKey(), version)
  Response response = Response.success(version);
  return response;
}
```

(TODO: MF: should there be something here about the scope of the version number? Do we version every field of a record, or the whole record/aggregate?)

Ordering Of Versioned Keys

> Embedded data stores like [rocksdb] [bib-rocksdb] or [boltdb] [bib-boltdb] are commonly used as storage layers of databases. In these storages, all data is logically arranged in sorted order of keys, very similar to the implementation shown here. Because these storages use byte array based keys and values, it's important to have the order maintained when the keys are serialized to byte arrays.

Because quickly navigating to the best matching versions is an important implementation concern, the versioned keys are arranged in such a way as to form a natural ordering by using version number as a suffix to the key. This maintains an order that fits well with the underlying data structure. For example, if there are two versions of a key, key1 and key2, key1 will be ordered before key2.

To store the versioned key values, a data structure, such as skip list, that allows quick navigation to the nearest matching versions is used. In Java the mvcc storage can be built as following:

class MVCCStore…

```
public class MVCCStore {
  NavigableMap<VersionedKey, String> kv = new ConcurrentSkipListMap

  public void put(VersionedKey key, String value) {
    kv.put(key, value);

  }
```

To work with the navigable map, the versioned key is implemented as follows. It implements a comparator to allow natural ordering of keys.

class VersionedKey…

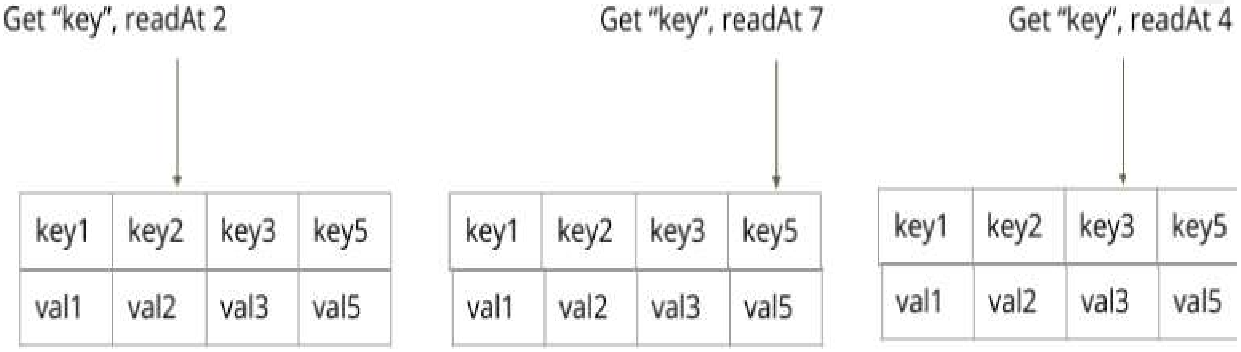
```
public class VersionedKey implements Comparable<VersionedKey> {
  private String key;
  private long version;

  public VersionedKey(String key, long version) {
    this.key = key;
    this.version = version;
  }

  public String getKey() {
    return key;
  }

  public long getVersion() {
    return version;
  }
  @Override
  public int compareTo(VersionedKey other) {
    int keyCompare = this.key.compareTo(other.key);
    if (keyCompare != 0) {
       return keyCompare;
    }
    return Long.compare(this.version, other.version);
  }
}
```

This implementation allows getting values for a specific version using the navigable map API.

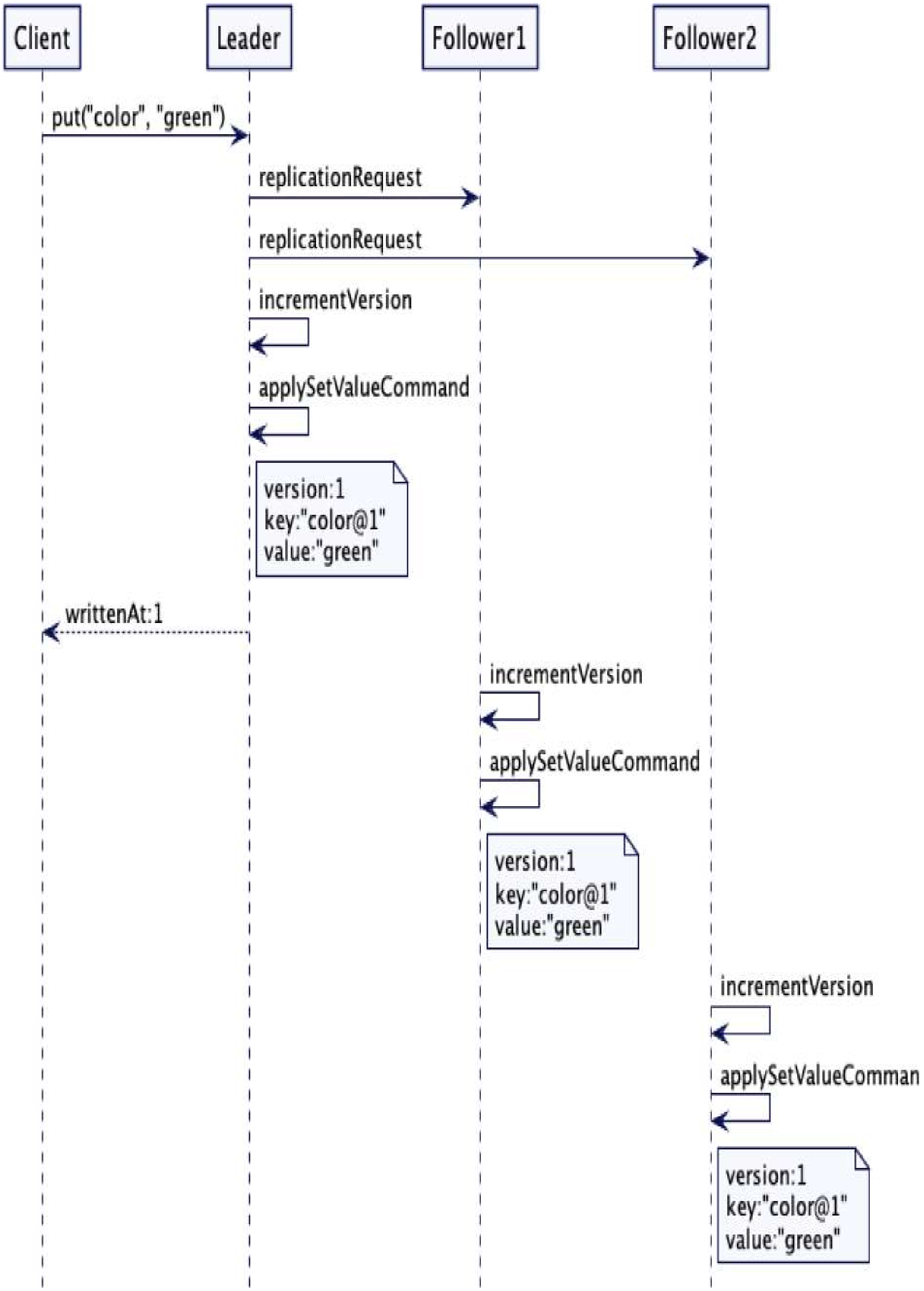
class MVCCStore…

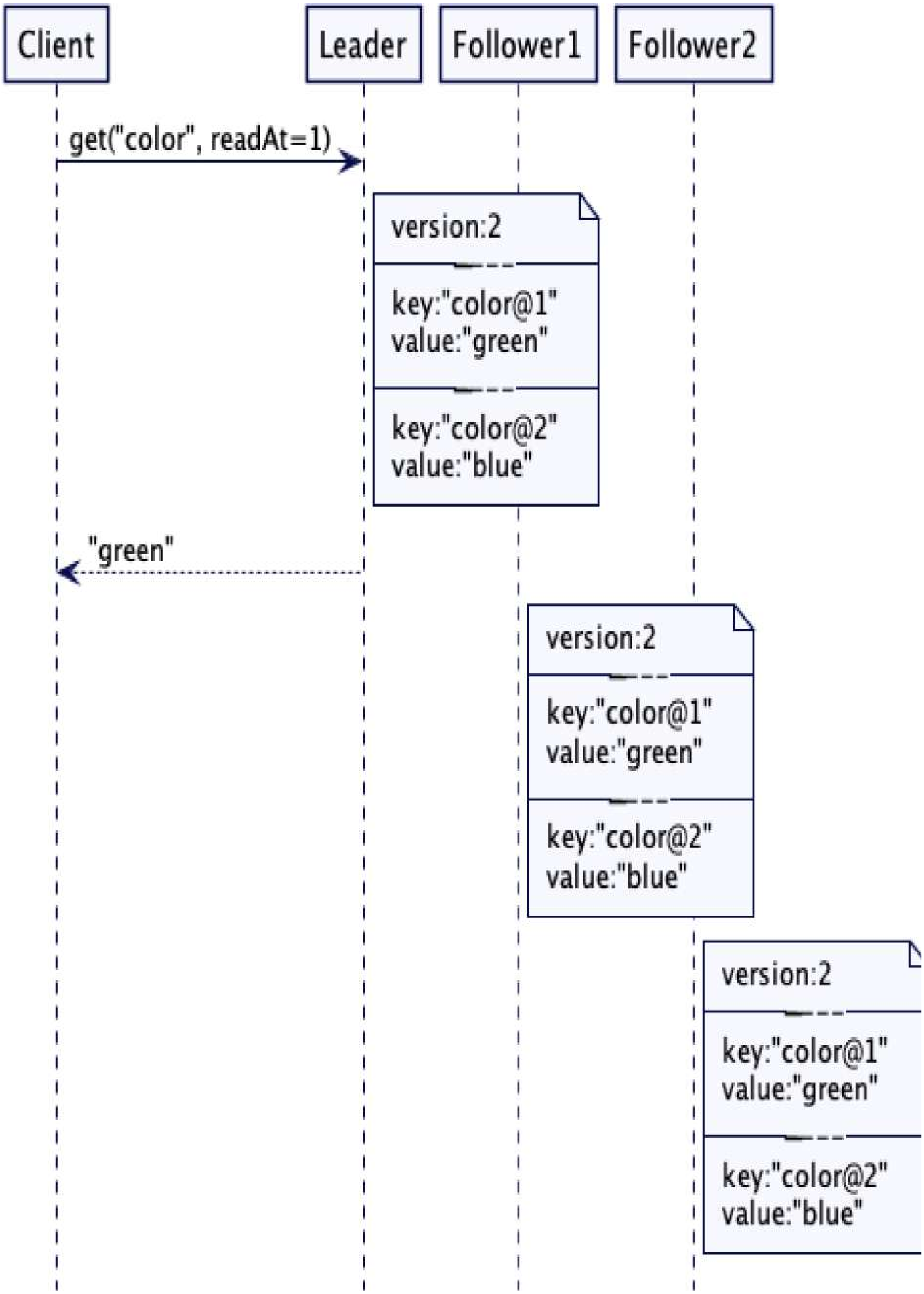
```
public Optional<String> get(final String key, final int readAt) {
  Map.Entry<VersionedKey, String> entry = kv.floorEntry(new Version
  return (entry == null)? Optional.empty(): Optional.of(entry.getVa
}
```

Consider an example where there are four versions of a key stored at version numbers 1, 2, 3 and 5. Depending on the version used by clients to read values, the nearest matching version of the key is returned.



The version at which the specific key value is stored is returned to the client. The client can then use this version to read the values. The overall working is as follows.


Client
Leader
Follower1
Follower2
put("color", "green")
replicationRequest
replicationRequest
incrementVersion
applySetValueCommand
version:1
key:"color@1"
value:"green"
writtenAt:1
incrementVersion
applySetValueCommand
version:1
key:"color@1"
value:"green"
incrementVersion
applySetValueComman
version:1
key:"color@1"
value:"green"


Client
Leader
Follower1
Follower2
get("color", readAt=1)
version:2
key:"color@1"
value:"green"
key:"color@2"
value:"blue"
"green"
version:2
key:"color@1"
value:"green"
key:"color@2"
value:"blue"
version:2
key:"color@1"
value:"green"
key:"color@2"
value:"blue"

Reading multiple versions

Sometimes clients need to get all the versions from a given version number. For example, in *State Watch* the client needs to get all the events from a specific version.

The cluster node can store additional index structures to store all the versions for a key.

class IndexedMVCCStore…

```
public class IndexedMVCCStore {
NavigableMap<String, List<Integer>> keyVersionIndex = new TreeMap<>
NavigableMap<VersionedKey, String> kv = new TreeMap<>();

ReadWriteLock rwLock = new ReentrantReadWriteLock();
int version = 0;

public int put(String key, String value) {
  rwLock.writeLock().lock();
  try {
     version = version + 1;
     kv.put(new VersionedKey(key, version), value);

     updateVersionIndex(key, version);

     return version;
  } finally {
     rwLock.writeLock().unlock();
  }
}

private void updateVersionIndex(String key, int newVersion) {
  List<Integer> versions = getVersions(key);
  versions.add(newVersion);
  keyVersionIndex.put(key, versions);
}
```

```
private List<Integer> getVersions(String key) {
  List<Integer> versions = keyVersionIndex.get(key);
  if (versions == null) {
     versions = new ArrayList<>();
     keyVersionIndex.put(key, versions);
  }
  return versions;
}
```

Then a client API can be provided to read values from a specific version or for a version range.

class IndexedMVCCStore…

```
public List<String> getRange(String key, final int fromRevision, in
  rwLock.readLock().lock();
  try {
     List<Integer> versions = keyVersionIndex.get(key);
     Integer maxRevisionForKey = versions.stream().max(Integer::com
     Integer revisionToRead = maxRevisionForKey > toRevision ? toRe
     SortedMap<VersionedKey, String> versionMap = kv.subMap(new Ver
     getLogger().info("Available version keys " + versionMap + ". R
     return new ArrayList<>(versionMap.values());

  } finally {
    rwLock.readLock().unlock();
  }
}
```

Care must be taken to use appropriate locking while updating and reading from the index.

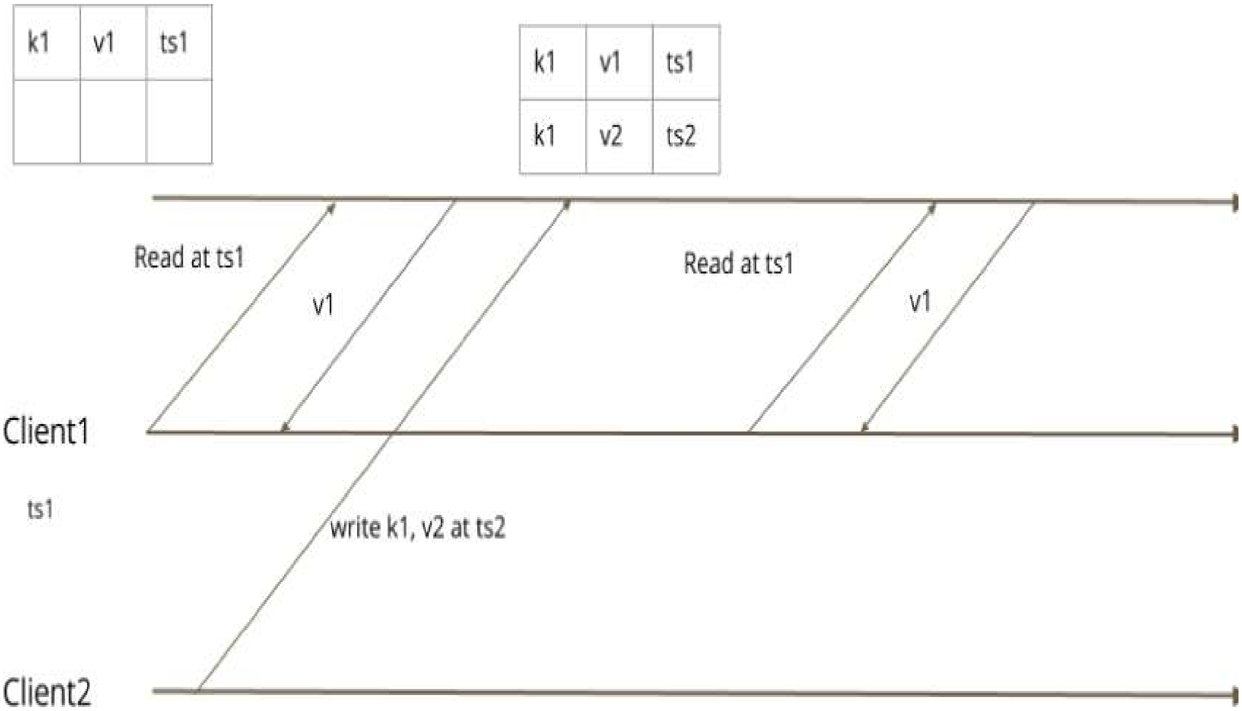
There is an alternate implementation possible to save a list of all the versioned values with the key, as used in *Gossip Dissemination* to avoid unnecessary state exchange. [gossip-dissemination.xhtml#AvoidingUnnecessaryStateExchange]

MVCC and Transaction Isolation

Databases use Versioned Value to implement [mvcc] [bib-mvcc] and [transaction-isolation] [bib-transaction-isolation].

Concurrency Control is about how locking is used when there are multiple concurrent requests accessing the same data. When locks are used to synchronize access, all the other requests are blocked until a request holding the lock is complete and the lock released. With Versioned Value, every write request adds a new record. This allows usage of non-blocking data structures to store the values.

Transaction isolation levels, such as Snapshot Isolation [bib-snapshot-isolation], can be naturally implemented as well. When a client starts reading at a particular version, it’s guaranteed to get the same value every time it reads from the database, even if there are concurrent write transactions which commit a different value between multiple read requests.



Using RocksDb like storage engines

It is very common to use [rocksdb] [bib-rocksdb] or similar embedded storage engines as a storage backend for data stores. For example, etcd [bib-etcd] uses [boltdb] [bib-boltdb], CockroachDB [bib-cockroachdb] earlier used [rocksdb] [bib-rocksdb] and now uses a go-lang clone of RocksDb called [pebble] [bib-pebble].

These storage engines provide implementation suitable for storing versioned values. They internally use skip lists the same way described in the above section and rely on the ordering of keys. There is a way to provide custom comparator for ordering keys.

class VersionedKeyComparator…

```
public class VersionedKeyComparator extends Comparator {
  public VersionedKeyComparator() {
    super(new ComparatorOptions());
  }

  @Override
  public String name() {
    return "VersionedKeyComparator";
  }

  @Override
  public int compare(Slice s1, Slice s2) {
    VersionedKey key1 = VersionedKey.deserialize(ByteBuffer.wrap(s1
    VersionedKey key2 = VersionedKey.deserialize(ByteBuffer.wrap(s2
    return key1.compareTo(key2);
  }
}
```

The implementation using [rocksdb] [bib-rocksdb] can be done as follows:

class RocksDBStore…

```
private final RocksDB db;

public RocksDBStore(File cacheDir) {
  Options options = new Options();
  options.setKeepLogFileNum(30);
  options.setCreateIfMissing(true);
  options.setLogFileTimeToRoll(TimeUnit.DAYS.toSeconds(1));
  options.setComparator(new VersionedKeyComparator());
  try {
     db = RocksDB.open(options, cacheDir.getPath());
  } catch (RocksDBException e) {
     throw new RuntimeException(e);
  }
}

public void put(String key, int version, String value) throws Rocks
  VersionedKey versionKey = new VersionedKey(key, version);
  db.put(versionKey.serialize(), value.getBytes());
}

public String get(String key, int readAtVersion) {
  RocksIterator rocksIterator = db.newIterator();
  rocksIterator.seekForPrev(new VersionedKey(key, readAtVersion).se
  byte[] valueBytes = rocksIterator.value();
  return new String(valueBytes);
}
```

Examples

etc3d [bib-etcd3] uses mvcc backend with a single integer representing a version.

MongoDB [bib-mongodb] and CockroachDB [bib-cockroachdb] use mvcc backend with a hybrid logical clock.

Chapter 18. Version Vector

Maintain a list of counters, one per cluster node, to detect concurrent updates

Problem

If multiple servers allow the same key to be updated, its important to detect when the values are concurrently updated across a set of replicas.

Solution

Each key value is associated with a version vector [bib-version-vector] that maintains a number for each cluster node.

In essence, a version vector is a set of counters, one for each node. A version vector for three nodes (blue, green, black) would look something like [blue: 43, green: 54, black: 12]. Each time a node has an internal update, it updates its own counter, so an update in the green node would change the vector to [blue: 43, green: 55, black: 12]. Whenever two nodes communicate, they synchronize their vector stamps, allowing them to detect any simultaneous updates.

The difference with Vector Clock

[vector-clock] [bib-vector-clock] implementation is similar. But vector clocks are used to track every event occurring on the server. In contrast, version vectors are used to detect concurrent updates to same key across a set of replicas. So an instance of a version vector is stored per key and not per server. Databases like [riak] [bib-riak] use the term version vector instead of vector clock for their implementation [bib-

> riak-vector-clock]. Refer to [version-vectors-are-not-vector-clocks] [bib-version-vectors-are-not-vector-clocks] for more details.

A typical version vector implementation is as follows:

class VersionVector…

```
private final TreeMap<String, Long> versions;

public VersionVector() {
  this(new TreeMap<>());
}
public VersionVector(TreeMap<String, Long> versions) {
  this.versions = versions;
}

public VersionVector increment(String nodeId) {
  TreeMap<String, Long> versions = new TreeMap<>();
  versions.putAll(this.versions);
  Long version = versions.get(nodeId);
  if(version == null) {
    version = 1L;
  } else {
    version = version + 1L;
  }
  versions.put(nodeId, version);
  return new VersionVector(versions);
}
```

Each value stored on the server is associated with a version vector

class VersionedValue…

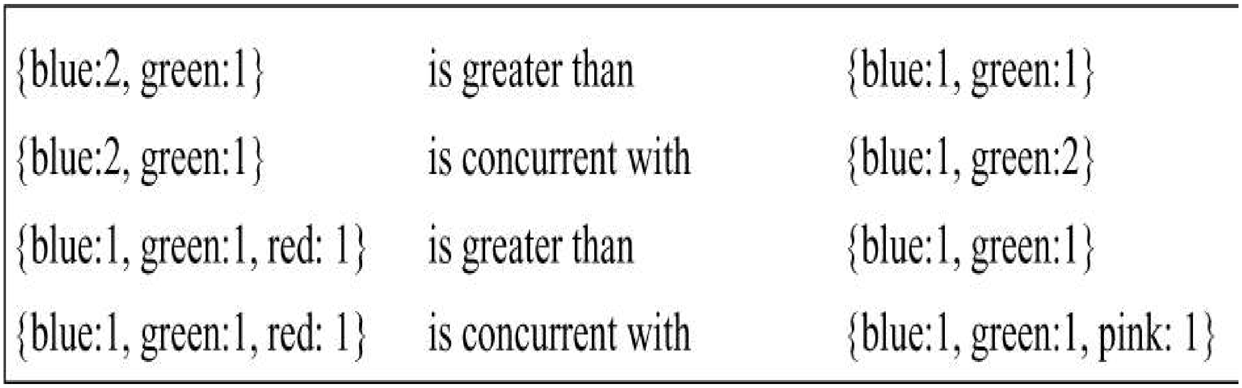
```
public class VersionedValue {
  String value;
  VersionVector versionVector;

  public VersionedValue(String value, VersionVector versionVector)
```

```
    this.value = value;
    this.versionVector = versionVector;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    VersionedValue that = (VersionedValue) o;
    return Objects.equal(value, that.value) && Objects.equal(versio
  }

  @Override
  public int hashCode() {
    return Objects.hashCode(value, versionVector);
  }
```

Comparing version vectors

Version vectors are compared by comparing version number for each node. A version vector is considered higher than the other if both of the version vectors have version number for the same cluster nodes and each version number is higher than the one in the other vector and vice versa. If the neither vector has all of the version numbers higher or if they have version numbers for different cluster nodes, they are considered concurrent.

Here are some example comparisons



{blue:2, green:1}	is greater than	{blue:1, green:1}
{blue:2, green:1}	is concurrent with	{blue:1, green:2}
{blue:1, green:1, red: 1}	is greater than	{blue:1, green:1}
{blue:1, green:1, red: 1}	is concurrent with	{blue:1, green:1, pink: 1}

The comparison is implemented as follows:

```
public enum Ordering {
  Before,
  After,
  Concurrent
}
```

class VersionVector…

```
//This is exact code for Voldermort implementation of VectorClock c
//https://github.com/voldemort/voldemort/blob/master/src/java/volde
public static Ordering compare(VersionVector v1, VersionVector v2)
  if(v1 == null || v2 == null)
     throw new IllegalArgumentException("Can’t compare null vector
  // We do two checks: v1 <= v2 and v2 <= v1 if both are true then
  boolean v1Bigger = false;
  boolean v2Bigger = false;

  SortedSet<String> v1Nodes = v1.getVersions().navigableKeySet();
  SortedSet<String> v2Nodes = v2.getVersions().navigableKeySet();
  SortedSet<String> commonNodes = getCommonNodes(v1Nodes, v2Nodes);
  // if v1 has more nodes than common nodes
  // v1 has clocks that v2 does not
  if(v1Nodes.size() > commonNodes.size()) {
     v1Bigger = true;
  }
  // if v2 has more nodes than common nodes
  // v2 has clocks that v1 does not
  if(v2Nodes.size() > commonNodes.size()) {
     v2Bigger = true;
  }
  // compare the common parts
  for(String nodeId: commonNodes) {
     // no need to compare more
     if(v1Bigger && v2Bigger) {
        break;
     }
     long v1Version = v1.getVersions().get(nodeId);
     long v2Version = v2.getVersions().get(nodeId);
     if(v1Version > v2Version) {
        v1Bigger = true;
     } else if(v1Version < v2Version) {
       v2Bigger = true;
     }
  }

  /*
  * This is the case where they are equal. Consciously return BEFOR
  * that the we would throw back an ObsoleteVersionException for on
  * writes with the same clock.
  */
  if(!v1Bigger && !v2Bigger)
     return Ordering.Before;
     /* This is the case where v1 is a successor clock to v2 */
  else if(v1Bigger && !v2Bigger)
     return Ordering.After;
    /* This is the case where v2 is a successor clock to v1 */
  else if(!v1Bigger && v2Bigger)
     return Ordering.Before;
    /* This is the case where both clocks are parallel to one anoth
  else
    return Ordering.Concurrent;
}

private static SortedSet<String> getCommonNodes(SortedSet<String> v
  // get clocks(nodeIds) that both v1 and v2 has
  SortedSet<String> commonNodes = Sets.newTreeSet(v1Nodes);
  commonNodes.retainAll(v2Nodes);
  return commonNodes;
}


public boolean descents(VersionVector other) {
  return other.compareTo(this) == Ordering.Before;
}
```

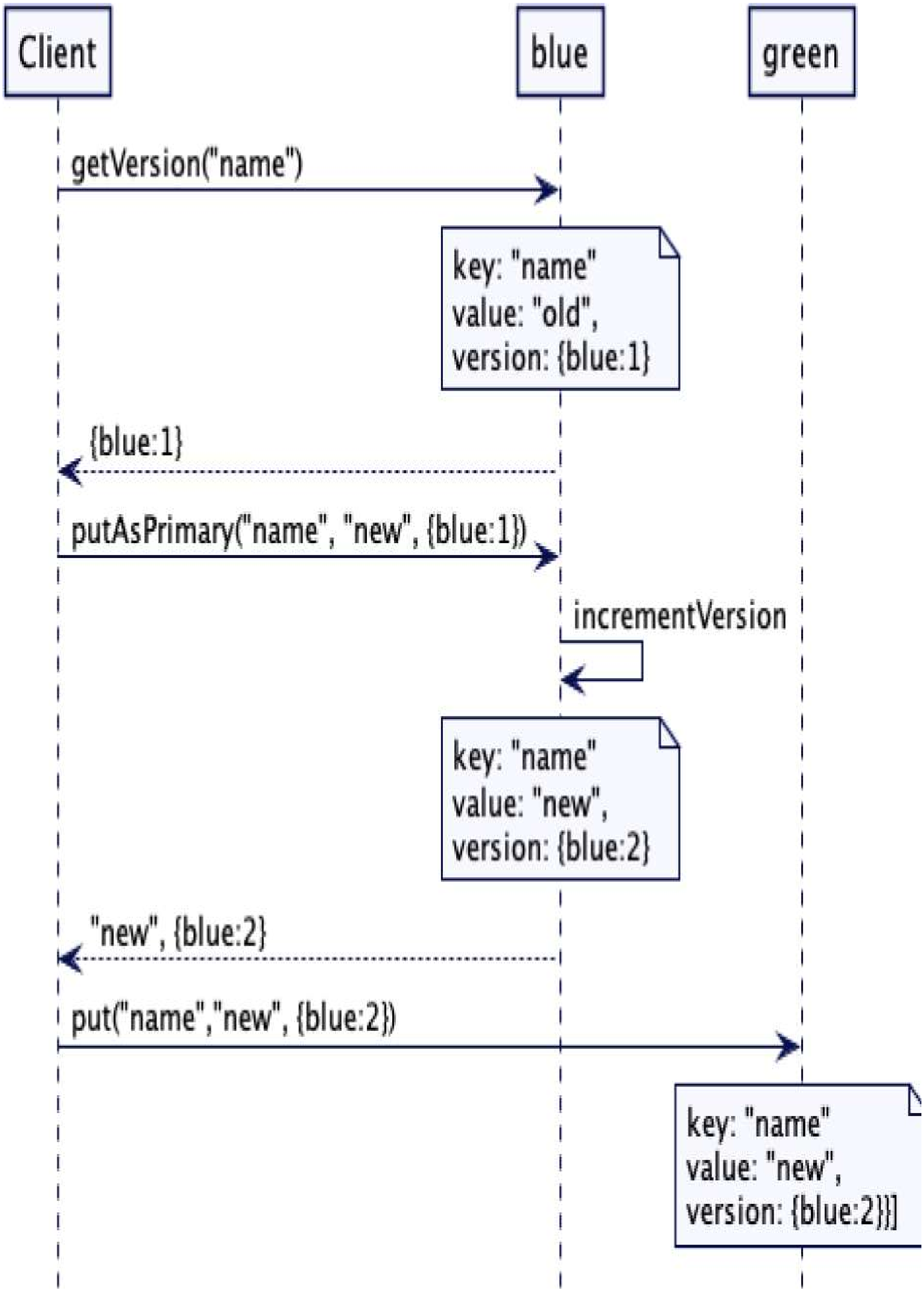
Using version vector in a key value store

The version vector can be used in a key value storage as follows. A list of versioned values is needed, as there can be multiple values which are concurrent.

class VersionVectorKVStore…

```
public class VersionVectorKVStore {
  Map<String, List<VersionedValue>> kv = new HashMap<>();
```

When a client wants to store a value, it first reads the latest known version for the given key. It then picks up the cluster node to store the value, based on the key. While storing the value, the client passes back the known version. The request flow is shown in the following diagram. There are two servers named blue and green. For the key "name", blue is the primary server.


Client
blue
green
getVersion("name")
key: "name"
value: "old",
version: {blue:1}
{blue:1}
putAsPrimary("name", "new", {blue:1})
incrementVersion
key: "name"
value: "new",
version: {blue:2}
"new", {blue:2}
put("name","new", {blue:2})
key: "name"
value: "new",
version: {blue:2}}]

In the leader-less replication scheme, the client or a coordinator node picks up the node to write data based on the key. The version vector is updated based on the primary cluster node that the key maps to. A value with the same version vector is copied on the other cluster nodes for replication. If the cluster node mapping to the key is not available, the next node is chosen. The version vector is only incremented for the first cluster node the value is saved to. All the other nodes save the copy of the data. The code for incrementing version vector in databases like [voldemort] [bib-voldemort] looks like this:

class ClusterClient…

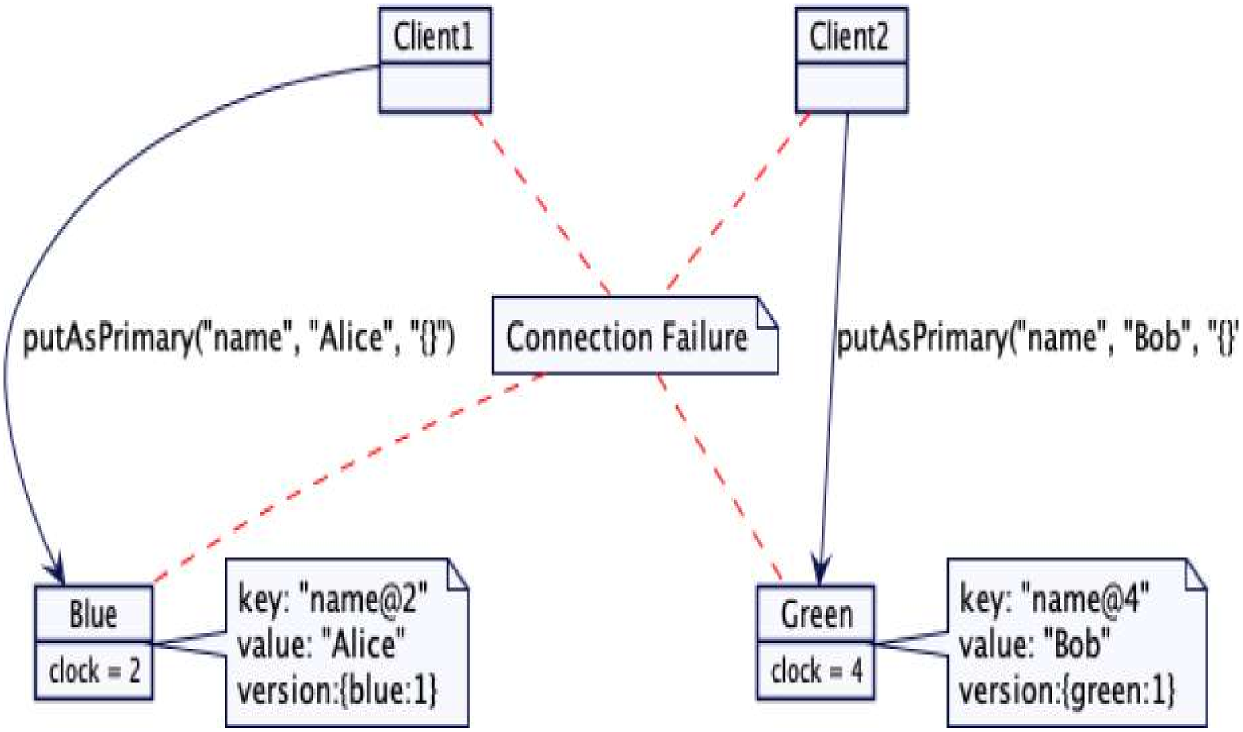
```
public void put(String key, String value, VersionVector existingVer
  List<Integer> allReplicas = findReplicas(key);
  int nodeIndex = 0;
  List<Exception> failures = new ArrayList<>();
  VersionedValue valueWrittenToPrimary = null;
  for (; nodeIndex < allReplicas.size(); nodeIndex++) {
    try {
      ClusterNode node = clusterNodes.get(nodeIndex);
      //the node which is the primary holder of the key value is re
      valueWrittenToPrimary = node.putAsPrimary(key, value, existin
      break;
   } catch (Exception e) {
     //if there is exception writing the value to the node, try oth
     failures.add(e);
   }
 }

 if (valueWrittenToPrimary == null) {
    throw new NotEnoughNodesAvailable("No node succeeded in writing
 }

 //Succeded in writing the first node, copy the same to other nodes
 nodeIndex++;
 for (; nodeIndex < allReplicas.size(); nodeIndex++) {
   ClusterNode node = clusterNodes.get(nodeIndex);
   node.put(key, valueWrittenToPrimary);
  }
}
```

The node acting as a primary is the one which increments the version number.

```
public VersionedValue putAsPrimary(String key, String value, Versio
  VersionVector newVersion = existingVersion.increment(nodeId);
  VersionedValue versionedValue = new VersionedValue(value, newVers
  put(key, versionedValue);
  return versionedValue;
}

public void put(String key, VersionedValue value) {
  versionVectorKvStore.put(key, value);
}
```

As can be seen in the above code, it is possible for different clients to update the same key on different nodes for instance when a client cannot reach a specific node. This creates a situation where different nodes have different values which are considered ‘concurrent’ according to their version vector.

As shown in the following diagram, both client1 and client2 are trying to write to the key, "name". If client1 cannot write to server green, the green server will be missing the value written by client1. When client2 tries to write, but fails to connect to server blue, it will write on server green. The version vector for the key "name", will reflect that the servers, blue and green, have concurrent writes.



Therefore the version vector based storage keeps multiple versions for any key, when the versions are considered concurrent.

class VersionVectorKVStore…

```
public void put(String key, VersionedValue newValue) {
  List<VersionedValue> existingValues = kv.get(key);
  if (existingValues == null) {
     existingValues = new ArrayList<>();
  }

  rejectIfOldWrite(key, newValue, existingValues);
  List<VersionedValue> newValues = merge(newValue, existingValues);
  kv.put(key, newValues);
}

//If the newValue is older than existing one reject it.
private void rejectIfOldWrite(String key, VersionedValue newValue,
  for (VersionedValue existingValue : existingValues) {
     if (existingValue.descendsVersion(newValue)) {
        throw new ObsoleteVersionException("Obsolete version for ke
             + "’: " + newValue.versionVector);
     }
  }
}
//Merge new value with existing values. Remove values with lower ve
//If the old value is neither before or after (concurrent) with the
private List<VersionedValue> merge(VersionedValue newValue, List<Ve
  List<VersionedValue> retainedValues = removeOlderVersions(newValu
  retainedValues.add(newValue);
  return retainedValues;
}

private List<VersionedValue> removeOlderVersions(VersionedValue new
  List<VersionedValue> retainedValues = existingValues
      .stream()
      .filter(v -> !newValue.descendsVersion(v)) //keep versions wh
      .collect(Collectors.toList());
  return retainedValues;
}
```

If concurrent values are detected while reading from multiple nodes, an error is thrown, allowing the client to do possible conflict resolution.

Resolving conflicts

If multiple versions are returned from different replicas, vector clock comparison can allow the latest value to be detected.

class ClusterClient…

```
public List<VersionedValue> get(String key) {
  List<Integer> allReplicas = findReplicas(key);

  List<VersionedValue> allValues = new ArrayList<>();
  for (Integer index : allReplicas) {
     ClusterNode clusterNode = clusterNodes.get(index);
     List<VersionedValue> nodeVersions = clusterNode.get(key);
     allValues.addAll(nodeVersions);
  }

  return latestValuesAcrossReplicas(allValues);
}
private List<VersionedValue> latestValuesAcrossReplicas(List<Versio
  List<VersionedValue> uniqueValues = removeDuplicates(allValues);
  return retainOnlyLatestValues(uniqueValues);
}

private List<VersionedValue> retainOnlyLatestValues(List<VersionedV
  for (int i = 0; i < versionedValues.size(); i++) {
    VersionedValue v1 = versionedValues.get(i);
    versionedValues.removeAll(getPredecessors(v1, versionedValues))
  }
  return versionedValues;
}
private List<VersionedValue> getPredecessors(VersionedValue v1, Lis
  List<VersionedValue> predecessors = new ArrayList<>();
  for (VersionedValue v2 : versionedValues) {
    if (!v1.sameVersion(v2) && v1.descendsVersion(v2)) {
       predecessors.add(v2);
    }
  }
  return predecessors;
}

private List<VersionedValue> removeDuplicates(List<VersionedValue>
  return allValues.stream().distinct().collect(Collectors.toList())
}
```

Just doing conflict resolution based on version vectors is not enough when there are concurrent updates. So it’s important to allow clients to provide application-specific conflict resolvers. A conflict resolver can be provided by the client while reading a value.

```
public interface ConflictResolver {
  VersionedValue resolve(List<VersionedValue> values);
}
```

class ClusterClient…

```
public VersionedValue getResolvedValue(String key, ConflictResolver
  List<VersionedValue> versionedValues = get(key);
  return resolver.resolve(versionedValues);
}
```

For example, [riak] [bib-riak] allows applications to provide conflict resolvers as explained here [bib-riak-conflict-resolver].

Last Write Wins (LWW) Conflict Resolution

Cassandra and LWW

Cassandra [bib-cassandra], while architecturally same as [riak] [bib-riak] or [voldemort] [bib-voldemort], does not use version vectors at all, and supports only last write wins conlict resolution strategy. Casasndra being a column family database, rather than a simple key value store, it stores timestamp with each column, as opposed to a value as a whole. While this takes the burden of doing conflict resolution away from the users, users need to make sure that the [ntp] [bib-ntp] service is configured and working correctly across cassandra nodes. In the worst case scenario, some latest values can get overwritten by the older values because of clock drifts.

While the version vector allows detection of concurrent writes across a different set of servers, they do not by themselves provide any help to clients in figuring out which value to choose in case of conflicts. The burden is on the client to do the resolution. Sometimes clients prefer for the key value store to do conflict resolution based on the timestamp. While there are known issues with timestamps across servers, the simplicity of this approach makes it a preferred choice for clients, even with the risk of losing some updates because of issues with timestamps across servers. They rely fully on the services like NTP to be well configured and working across the cluster. Databases like [riak] [bib-riak] and [voldemort] [bib-voldemort] allow users to select the 'last write wins' conflict resolution strategy.

To support LWW conflict resolution, a timestamp is stored with each value while its written.

class TimestampedVersionedValue…

```
class TimestampedVersionedValue {
  String value;
  VersionVector versionVector;
  long timestamp;

  public TimestampedVersionedValue(String value, VersionVector vers
    this.value = value;
    this.versionVector = versionVector;
    this.timestamp = timestamp;
  }
```

While reading the value, the client can use the timestamp to pick up the latest value. The version vector is completely ignored in this case.

class ClusterClient…

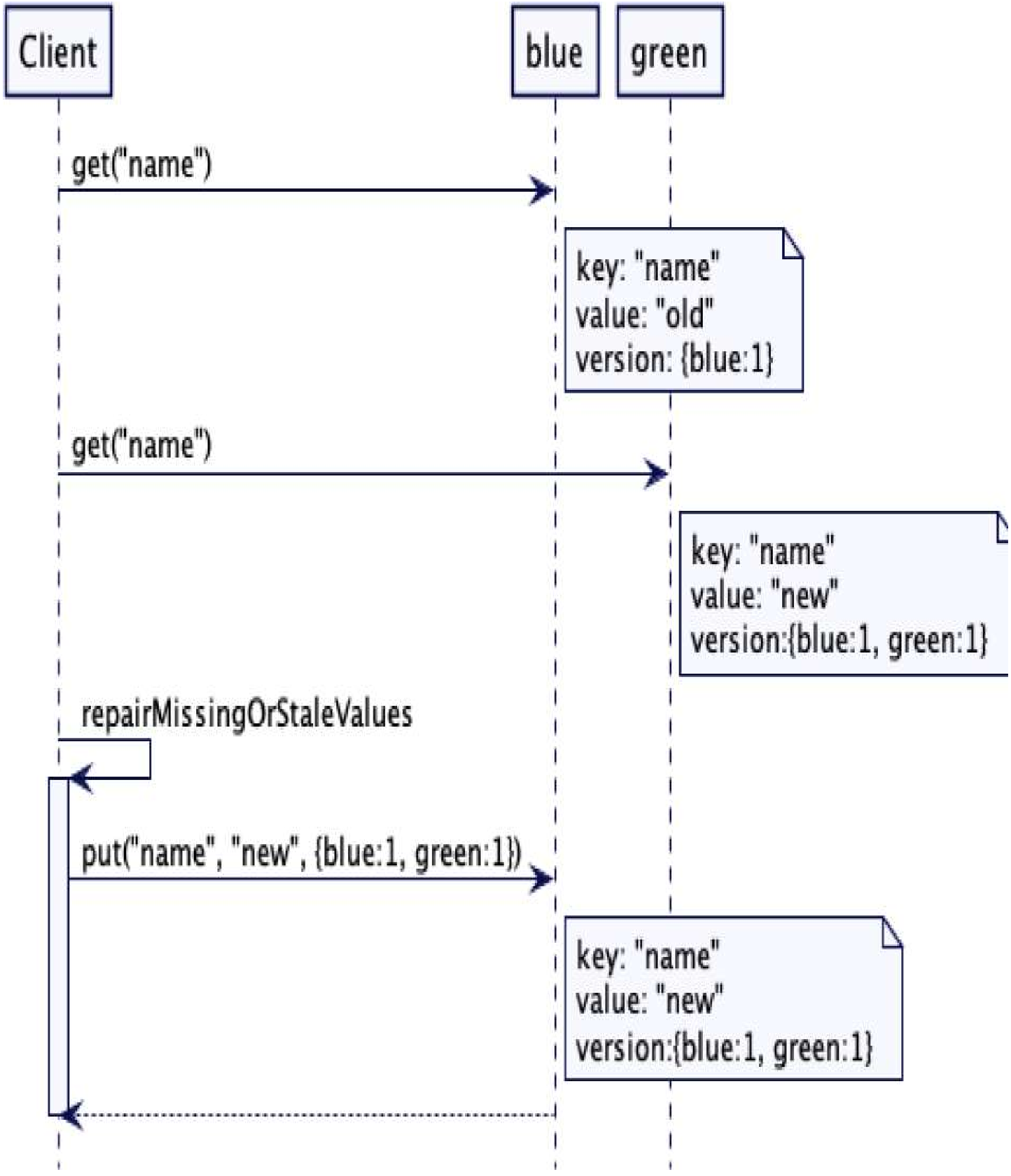
```
public Optional<TimestampedVersionedValue> getWithLWWW(List<Timesta
  return values.stream().max(Comparator.comparingLong(v -> v.timest
}
```

Read repair

While allowing any cluster node to accept write requests improves availability, it's important that eventually all of the replicas have the same data. One of the common methods to repair replicas happens when the client reads the data.

When the conflicts are resolved, it’s also possible to detect which nodes have older versions. The nodes with older versions can be sent the latest versions as part of the read request handling from the client. This is called as read repair.

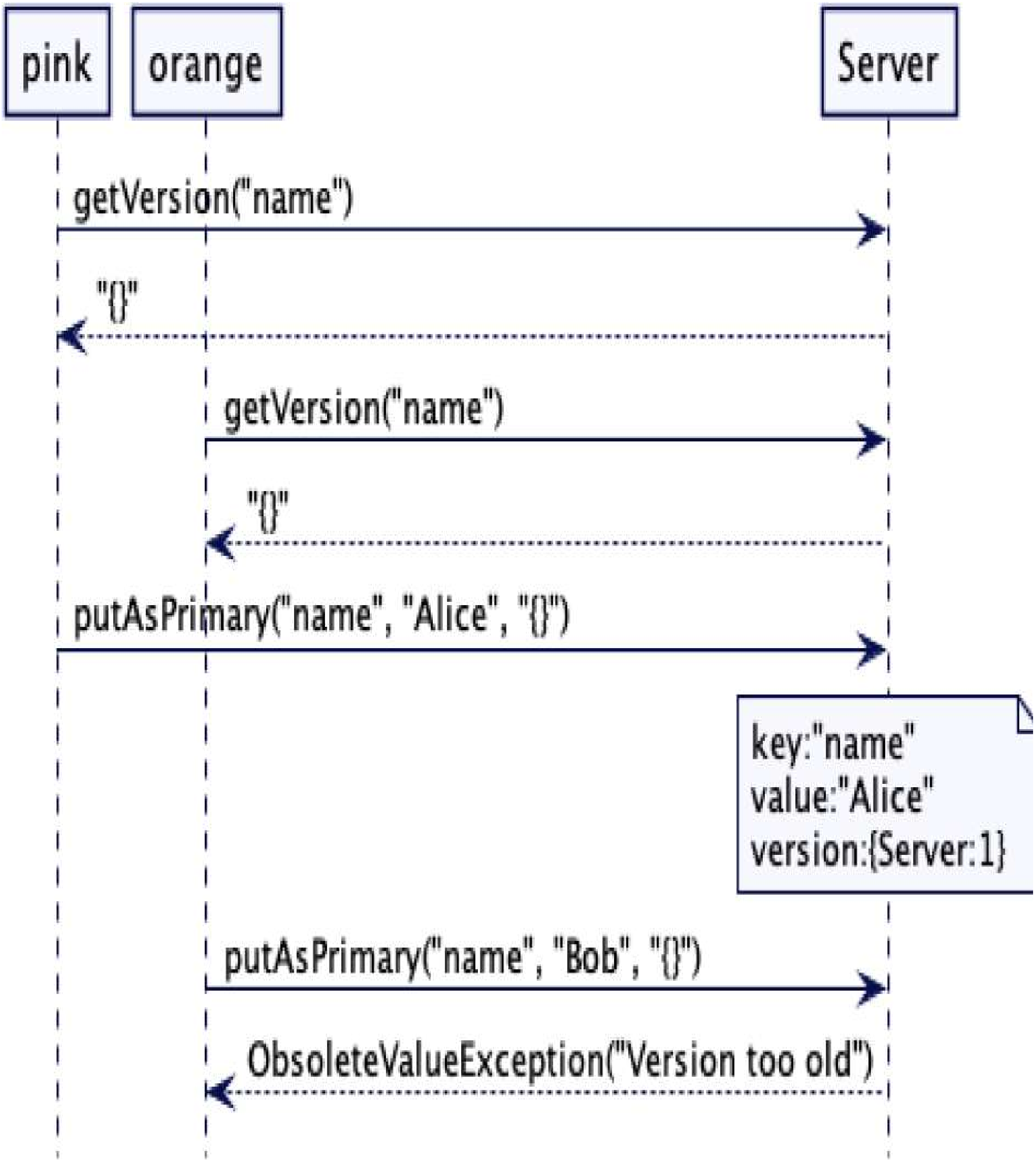
Consider a scenario shown in the following diagram. Two nodes, blue and green, have values for a key "name". The green node has the latest version with version vector [blue: 1, green:1]. When the values are read from both the replicas, blue and green, they are compared to find out which node is missing the latest version, and a put request with the latest version is sent to the cluster node.


Client
blue
green
get("name")
key: "name"
value: "old"
version: {blue:1}
get("name")
key: "name"
value: "new"
version:{blue:1, green:1}
repairMissingOrStaleValues
put("name", "new", {blue:1, green:1})
key: "name"
value: "new"
version:{blue:1, green:1}

Allowing concurrent updates on the same cluster node

There is a possibility of two clients writing concurrently to the same node. In the default implementation shown above, the second write will be rejected. The basic implementation with the version number per cluster node is not enough in this case.

Consider the following scenario. With two clients trying to update the same key, the second client will get an exception, as the version it passes in its put request is stale.


pink
orange
Server
getVersion("name")
"{}"
getVersion("name")
"{}"
putAsPrimary("name", "Alice", "{}")
key:"name"
value:"Alice"
version:{Server:1}
putAsPrimary("name", "Bob", "{}")
ObsoleteValueException("Version too old")

A database like [riak] [bib-riak] gives flexibility to clients to allow these kind of concurrent writes and prefer not getting error responses.

Using Client IDs instead of Server IDs

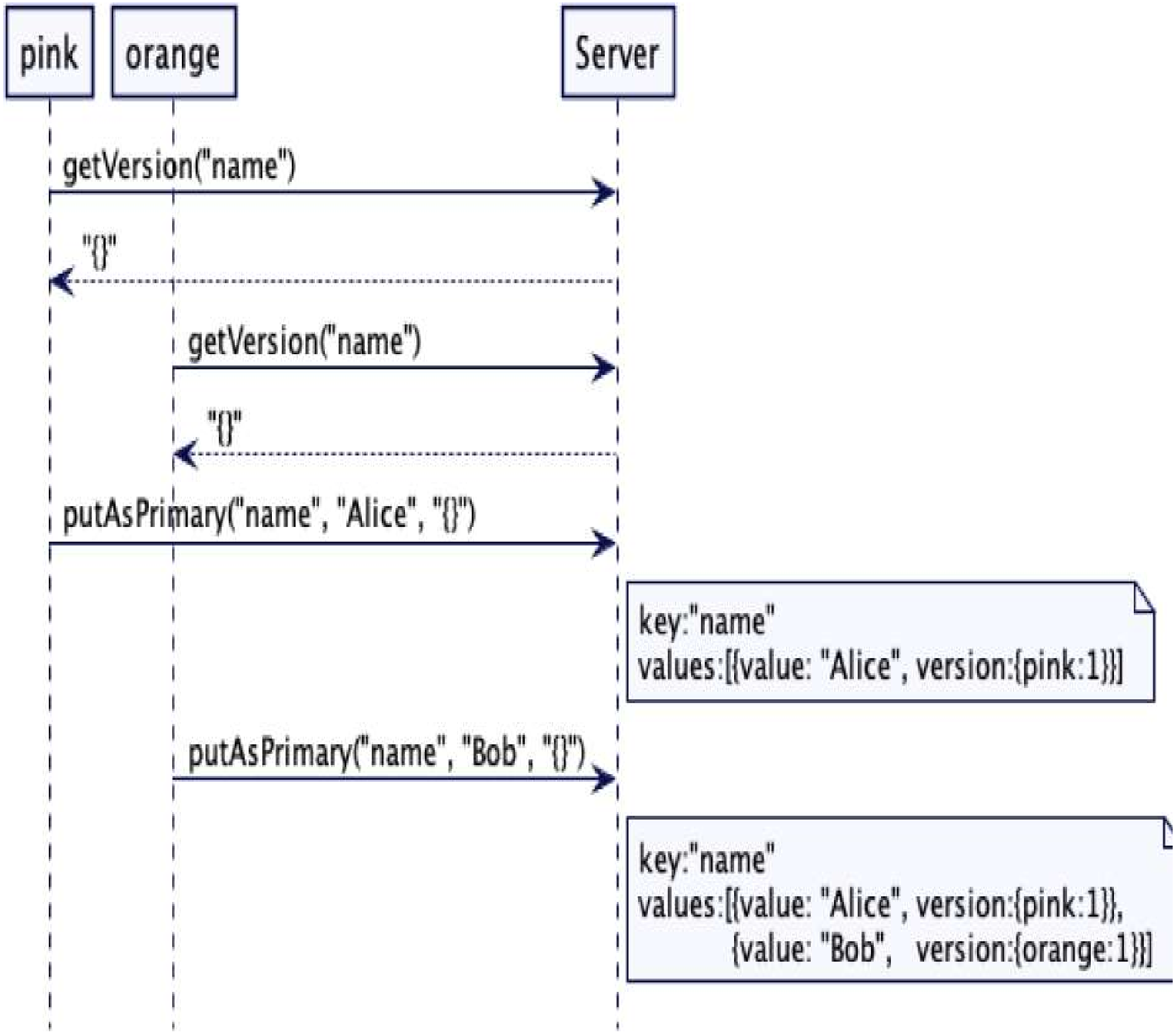
If each cluster client can have a unique ID, client ID can be used. A version number is stored per client ID. Every time a client writes a value, it first reads the existing version, increments the number associated with the client ID and writes it to the server.

class ClusterClient…

```
private VersionedValue putWithClientId(String clientId, int nodeInd
  ClusterNode node = clusterNodes.get(nodeIndex);
  VersionVector newVersion = version.increment(clientId);
  VersionedValue versionedValue = new VersionedValue(value, newVers
  node.put(key, versionedValue);
  return versionedValue;
}
```

Because each client increments its own counter, concurrent writes create sibling values on the servers, but concurrent writes never fail.

The above mentioned scenario, which gives error to second client, works as following:


pink
orange
Server
getVersion("name")
"{}"
getVersion("name")
"{}"
putAsPrimary("name", "Alice", "{}")
key:"name"
values:[{value: "Alice", version:{pink:1}}]
putAsPrimary("name", "Bob", "{}")
key:"name"
values:[{value: "Alice", version:{pink:1}},
{value: "Bob", version:{orange:1}}]

Dotted version vectors

One of the major problems with client ID based version vectors is that the size of the version vector is directly dependent on the number of clients. This causes cluster nodes to accumulate too many concurrent values for a given key over time. The problem is called as sibling explosion [bib-sibling-explosion]. To solve this issue and still allow cluster node based version vectors, [riak] [bib-riak] uses a variant of version vector called dotted version vector [bib-dvv].

Examples

[voldemort] [bib-voldemort] uses version vector in the way described here. It allows timestamp based last write wins conflict resolution.

[riak] [bib-riak] started by using client ID based version vectors, but moved to cluster node based version vectors and eventually to dotted version vectors. Riak also supports last write wins conflict resolution based on the system timestamp.

Cassandra [bib-cassandra] does not use version vectors, It supports only last write wins conflict resolution based on system timestamp.

Part III: Patterns of Data Partitioning

Chapter 19. Fixed Partitions

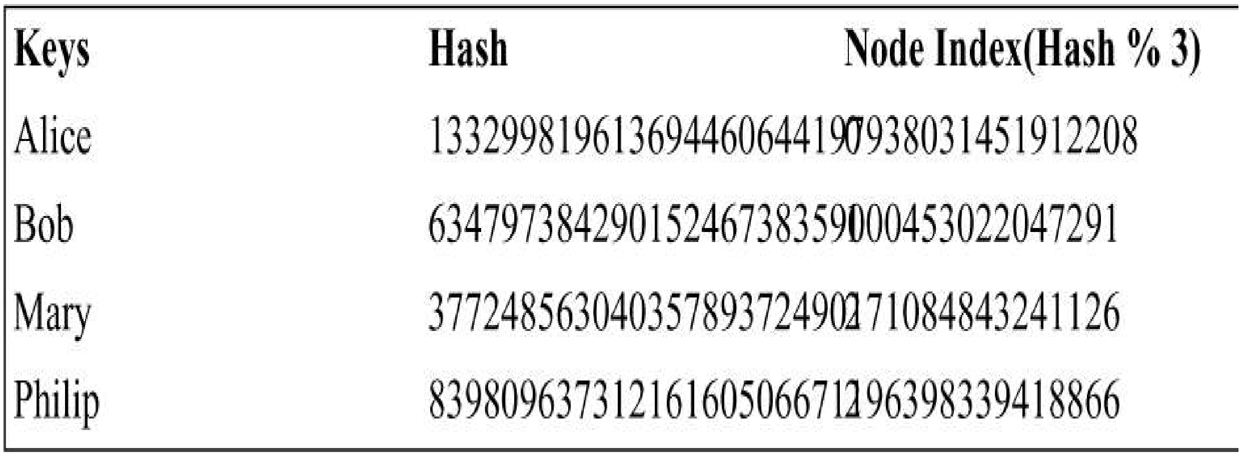
Keep the number of partitions fixed to keep the mapping of data to the partition unchanged when size of a cluster changes.

Problem

To split data across a set of cluster nodes, each data item needs to be mapped to them. There are two requirements for mapping the data to the cluster nodes.

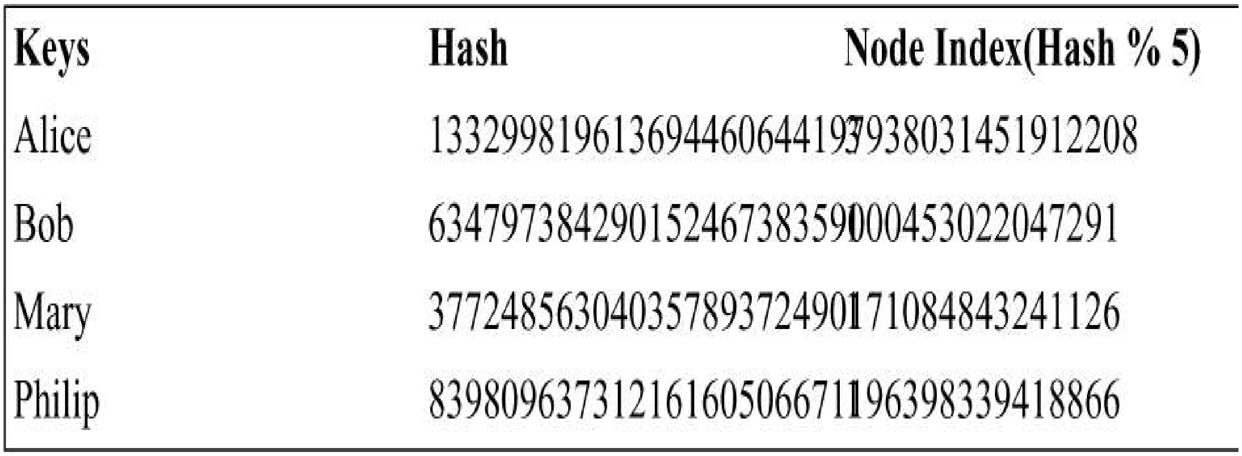
- The distribution should be uniform
- It should possible to know which cluster node stores a particular data item, without making a request to all the nodes

Considering a key value store, which is a good proxy for many storage systems, both requirements can be fulfilled by using the hash of the key and using what’s called the modulo operation to map it to a cluster node. So if we consider a three node cluster,we can map keys Alice, Bob, Mary and Philip like this:



Keys	Hash	Node Index(Hash % 3)
Alice	1332998196136944606441907938031451912208	
Bob	63479738429015246738359000453022047291	
Mary	377248563040357893724902171084843241126	
Philip	839809637312161605066712963983394 18866	

However, this method creates a problem when the cluster size changes. If two more nodes are added to the cluster, we will have five nodes. The mapping will then look like this:



Keys	Hash	Node Index(Hash % 5)
Alice	1332998196136944606441937938031451912208	
Bob	63479738429015246738359000453022047291	
Mary	37724856304035789372490171084843241126	
Philip	83980963731216160506671196398339418866	

The way almost all the keys are mapped changes. Even by adding only a few new cluster nodes, all the data needs to be moved. When the data size is large, this is undesirable.

Solution

> A message broker like Kafka [bib-kafka] needs an ordering guarantee for the data per partition. With fixed partitions, the data per partition doesn’t change even when partitions are moved around the cluster nodes when new nodes are added.This maintains the ordering of data per partition.

One of the most commonly used solution is to map data to logical partitions. Logical partitions are mapped to the cluster nodes. Even if cluster nodes are added or removed, the mapping of data to partitions doesn’t change. The cluster is launched with a preconfigured number of partitions say, for the sake of this example, 1024. This number does not change when new nodes are added to the cluster. So the way data is mapped to partitions using the hash of the key remains the same.

It’s important that partitions are evenly distributed across cluster nodes. When partitions are moved to new nodes, it should be relatively quick with only a smaller portion of the data movement. Once configured, the partition number won’t change; this mean it should have enough room for future growth of data volumes.

So the number of partitions selected should be significantly higher than the number of cluster nodes. For example, Akka [bib-akka] suggests you should have number of shards ten times the number of nodes. partitioning in Apache Ignite [bib-ignite-partitioning] has it’s default value as 1024. Hazelcast [bib-hazelcast] has a default value of 271 for cluster size smaller than 100.

Data storage or retrieval is then a two step process.

- First, you find the partition for the given data item
- Then you find the cluster node where the partition is stored

To balance data across the cluster nodes when new ones are added, some of the partitions can be moved to the new nodes.

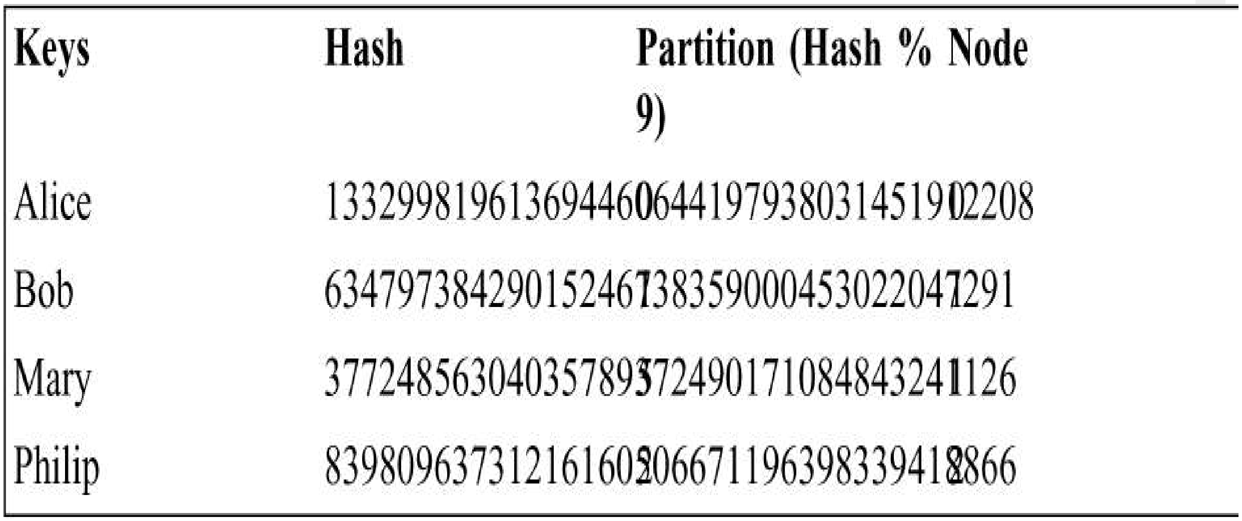
Choosing the hash function

It’s critical to choose the hashing method which gives the same hash values independent of the platform and runtime. For example, programming languages like Java provide a hash for every object. However, it’s important to note that hash value is dependent on the JVM runtime. So two different JVMs could give a different hash for the same key. To tackle this, hashing algorithms like MD5 hash or Murmur hash are used.

class HashingUtil…

```
public static BigInteger hash(String key)
{
  try
  {
    MessageDigest messageDigest = MessageDigest.getInstance("MD5");
    return new BigInteger(messageDigest.digest(key.getBytes()));
  }
  catch (Exception e)
  {
    throw new RuntimeException(e);
  }
}
```

The keys are not mapped to nodes, but to partitions. Considering there are 9 partitions, the table looks like following. With the addition of new nodes to the cluster, the mapping of a key to partition does not change.



Keys	Hash	Partition (Hash % Node 9)
Alice	133299819613694460644197938031451912208	0
Bob	63479738429015246738359000453022047291	1
Mary	37724856304035789372490171084843241126	5
Philip	83980963731216160506671196398339418866	2

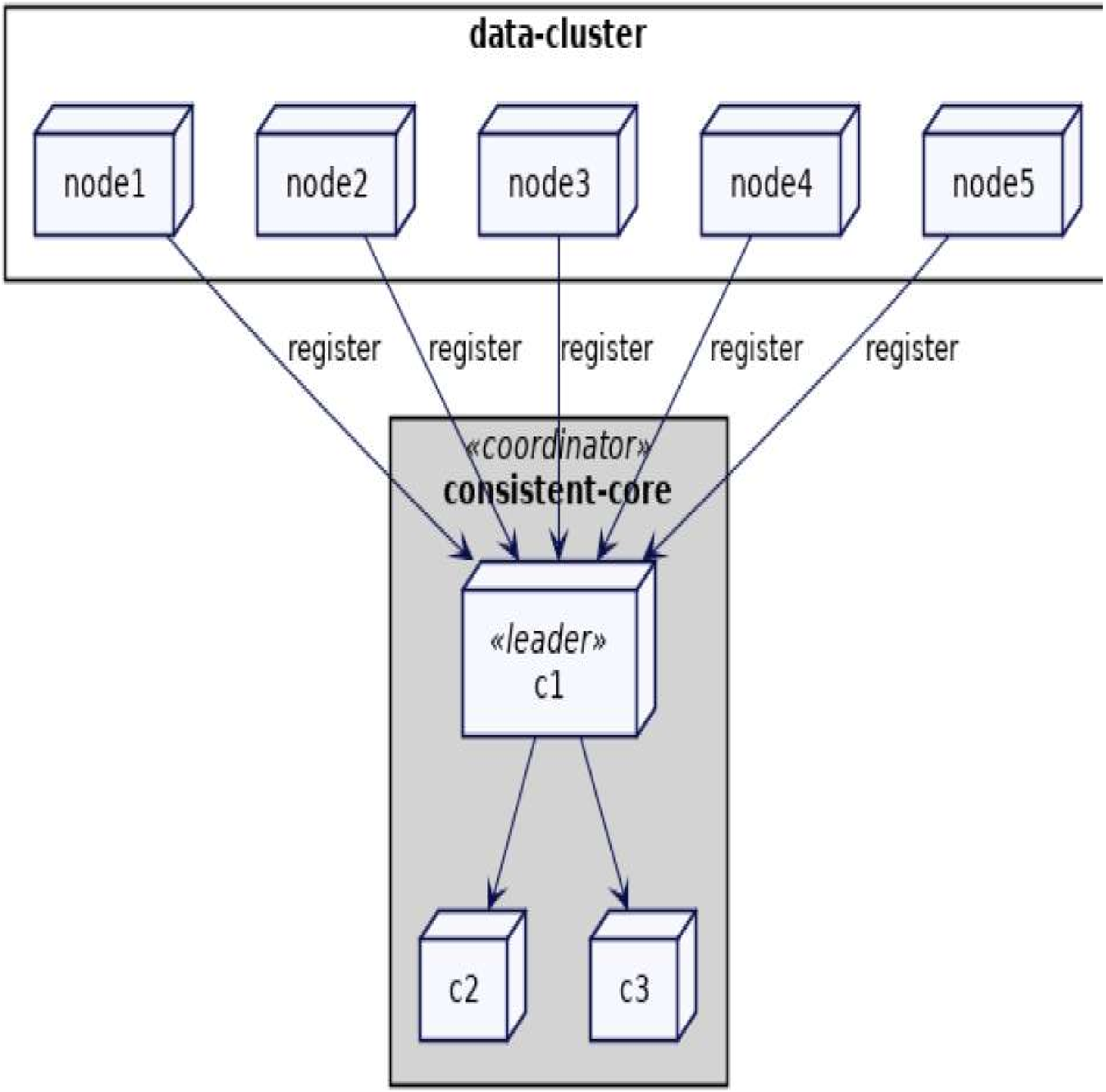
Mapping partitions to cluster nodes

Partitions need to be mapped to cluster nodes. The mapping also needs to be stored and made accessible to the clients. It's common to use a dedicated *Consistent Core*; this handles both. The dedicated Consistent Core acts as a coordinator which keeps track of all nodes in the cluster and maps partitions to nodes. It also stores the mapping in a fault tolerant way by using a *Replicated Log*. The master cluster in YugabyteDB [bib-yugabyte] or controller implementation [bib-kip-631] in Kafka are both good examples of this.

Peer-to-peer systems like Akka [bib-akka] or Hazelcast [bib-hazelcast] also need a particular cluster node to act as an coordinator. They use *Emergent Leader* as the coordinator.

Systems like Kubernetes [bib-kubernetes] use a generic Consistent Core like etcd [bib-etcd]. They need to elect one of the cluster nodes to play the role of coordinator as discussed here. [leaderfollower.xhtml#LeaderElectionUsingExternallinearizableStore]

Tracking Cluster Membership



Each cluster node will register itself with the consistent-core. It also periodically sends a *HeartBeat* to allow the Consistent Core detect node failures.

class KVStore…

```
public void start() {
  socketListener.start();
  requestHandler.start();
  network.sendAndReceive(coordLeader, new RegisterClusterNodeReques
  scheduler.scheduleAtFixedRate(()->{
    network.send(coordLeader, new HeartbeatMessage(generateMessageI
  }, 200, 200, TimeUnit.MILLISECONDS);
}
```

The coordinator handles the registration and then stores member information.

class ClusterCoordinator…

```
ReplicatedLog replicatedLog;
Membership membership = new Membership();
TimeoutBasedFailureDetector failureDetector = new TimeoutBasedFailu

private void handleRegisterClusterNodeRequest(Message message) {
  logger.info("Registering node " + message.from);
  CompletableFuture completableFuture = registerClusterNode(message
  completableFuture.whenComplete((response, error) -> {
    logger.info("Sending register response to node " + message.from
    network.send(message.from, new RegisterClusterNodeResponse(mess
  });
}

public CompletableFuture registerClusterNode(InetAddressAndPort add
  return replicatedLog.propose(new RegisterClusterNodeCommand(addre
}
```

When a registration is committed in the *Replicated Log*, the membership will be updated.

class ClusterCoordinator…

```
private void applyRegisterClusterNodeEntry(RegisterClusterNodeComma
  updateMembership(command.memberAddress);
}
```

class ClusterCoordinator…

```
private void updateMembership(InetAddressAndPort address) {
  membership = membership.addNewMember(address);
  failureDetector.heartBeatReceived(address);
}
```

The coordinator maintains a list of all nodes that are part of the cluster:

class Membership…

```
public class Membership {
  List<Member> liveMembers = new ArrayList<>();
  List<Member> failedMembers = new ArrayList<>();

  public boolean isFailed(InetAddressAndPort address) {
    return failedMembers.stream().anyMatch(m -> m.address.equals(ad
  }
```

class Member…

```
public class Member implements Comparable<Member> {
  InetAddressAndPort address;
  MemberStatus status;
```

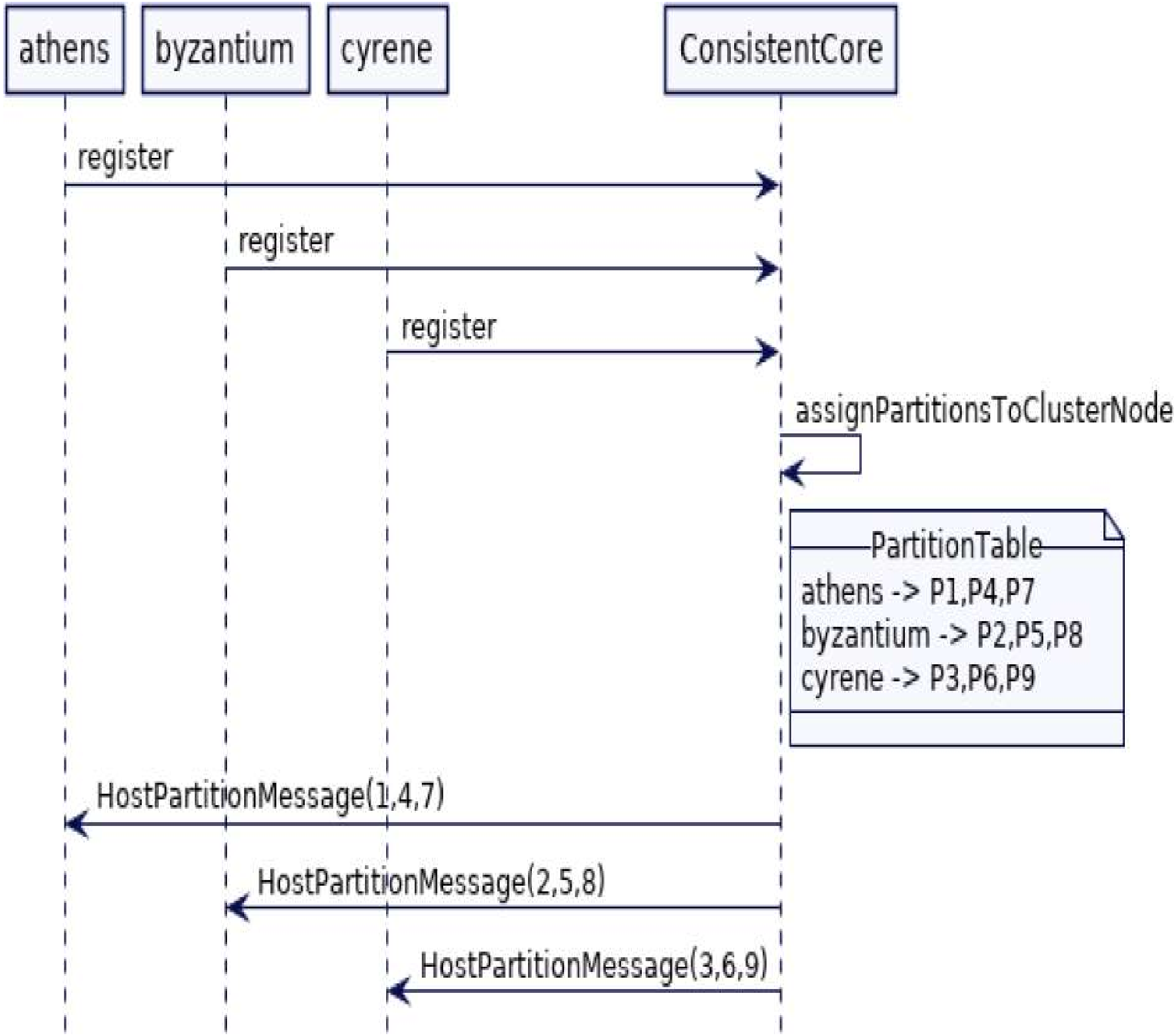
The coordinator will detect cluster node failures using a mechanism similar to *Lease*. If a cluster node stops sending the heartbeat, the node will be marked as failed.

class ClusterCoordinator…

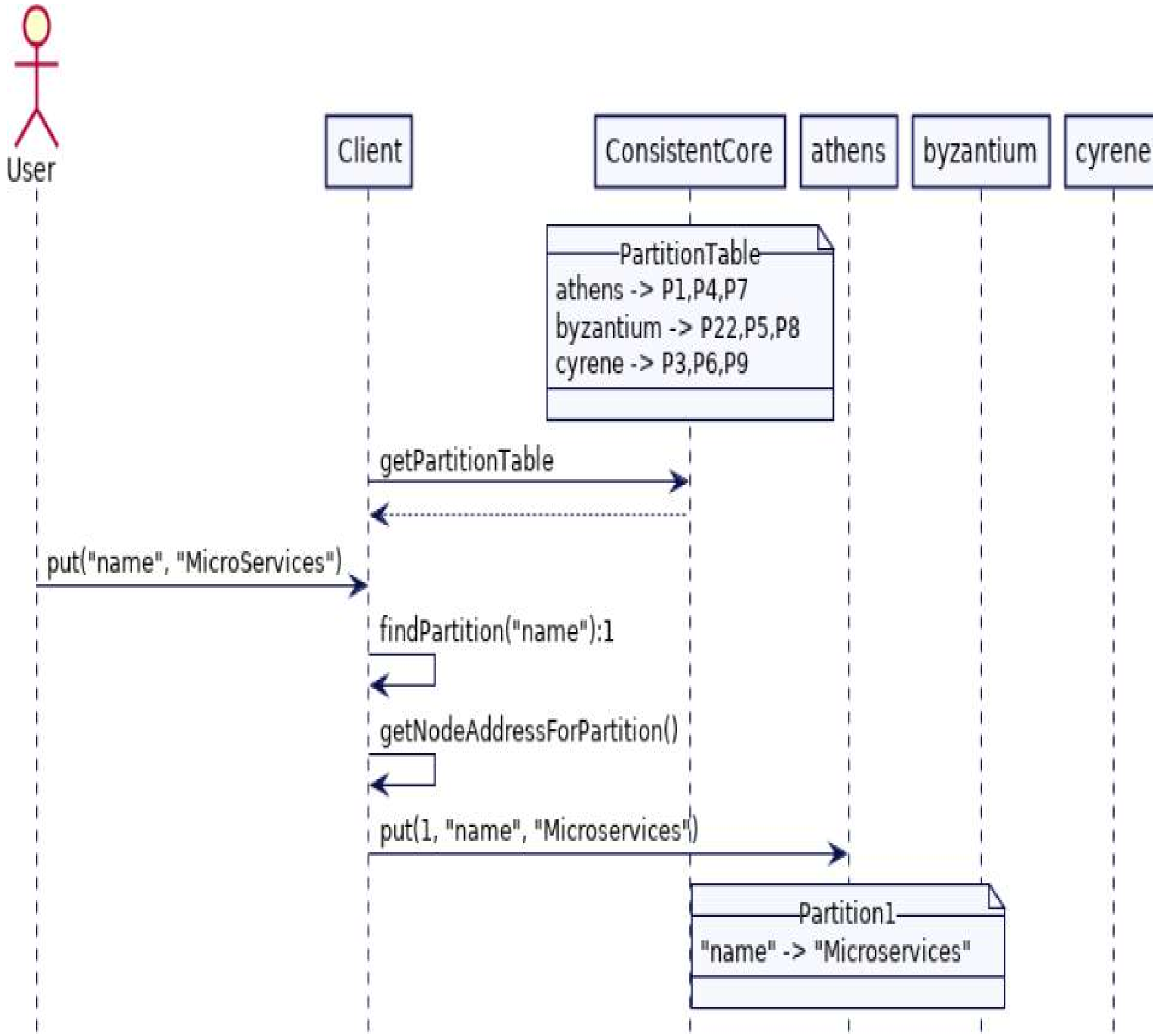
```
@Override
public void onBecomingLeader() {
  scheduledTask = executor.scheduleWithFixedDelay(this::checkMember
      1000,
      1000,
      TimeUnit.MILLISECONDS);
  failureDetector.start();
}

private void checkMembership() {
  List<Member> failedMembers = getFailedMembers();
  if (!failedMembers.isEmpty()) {
     replicatedLog.propose(new MemberFailedCommand(failedMembers));
  }
}

private List<Member> getFailedMembers() {
  List<Member> liveMembers = membership.getLiveMembers();
  return liveMembers.stream()
       .filter(m -> failureDetector.isMonitoring(m.getAddress()) &&
       .collect(Collectors.toList());
}
```

An example scenario

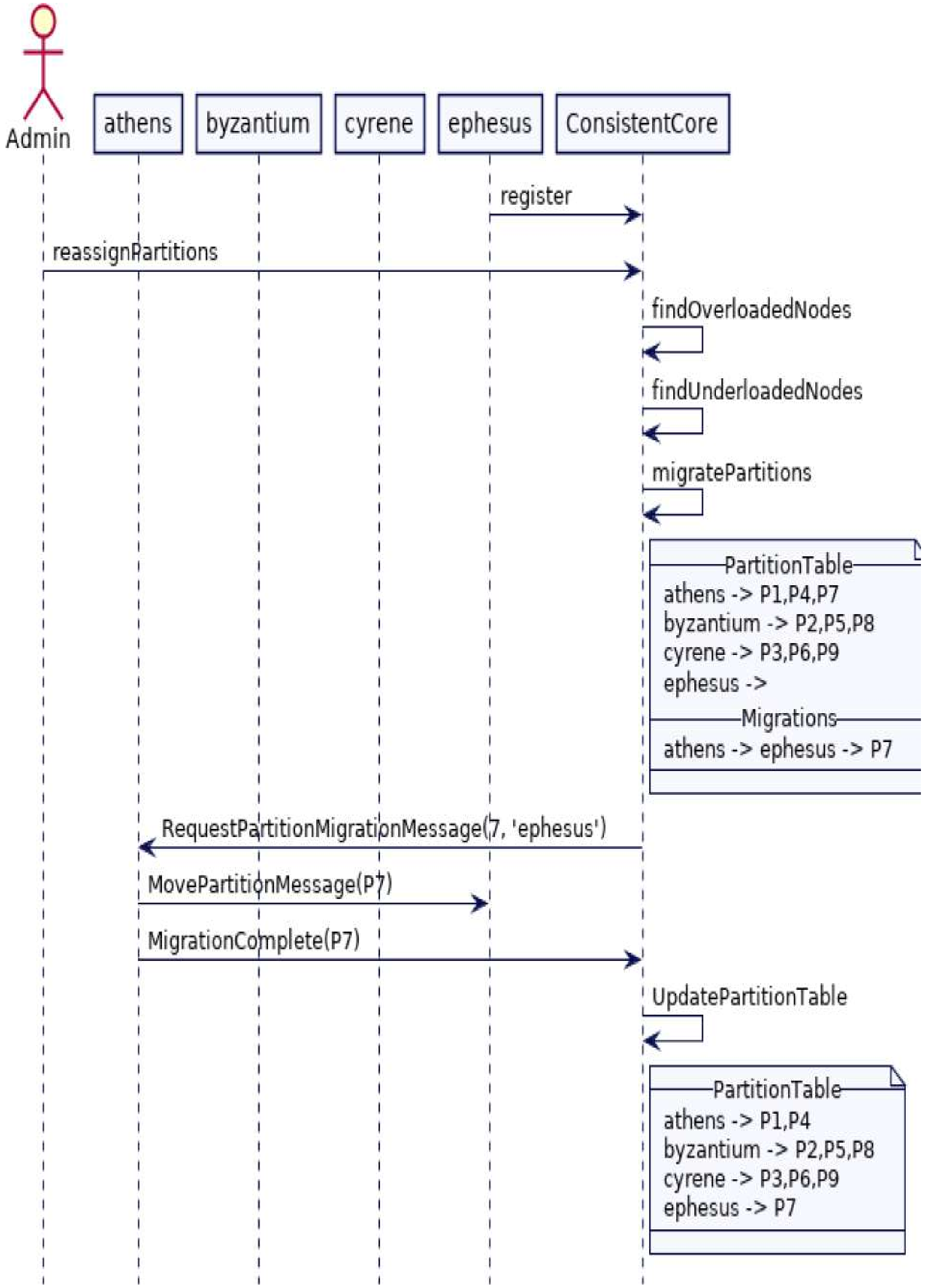
Consider that there are three data servers athens, byzantium and cyrene. Considering there are 9 partitions, the flow looks like following.



The client can then use the partition table to map a given key to a particular cluster node.



Now a new cluster node - ‘ephesus’ - is added to the cluster. The admin triggers a reassignment and the coordinator checks which nodes are underloaded by checking the partition table. It figures out that ephesus is the node which is underloaded, and decides to allocate partition 7 to it, moving it from athens. The coordinator stores the migrations and then sends the request to athens to move partition 7 to ephesus. Once the migration is complete, athens lets the coordinator know. The coordinator then updates the partition table.


Admin
athens
byzantium
cyrene
ephesus
ConsistentCore
register
reassignPartitions
findOverloadedNodes
findUnderloadedNodes
migratePartitions
PartitionTable
athens -> P1,P4,P7
byzantium -> P2,P5,P8
cyrene -> P3,P6,P9
ephesus ->
Migrations
athens -> ephesus -> P7
RequestPartitionMigrationMessage(7, 'ephesus')
MovePartitionMessage(P7)
MigrationComplete(P7)
UpdatePartitionTable
PartitionTable
athens -> P1,P4
byzantium -> P2,P5,P8
cyrene -> P3,P6,P9
ephesus -> P7

Assigning Partitions To Cluster Nodes

> For data stores like Kafka [bib-kafka] or Hazelcast [bib-hazelcast] which have logical storage structures like topics, caches or tables, the partitions are created at the same time as the tables, topics or caches. The expectation is that, the storage structures will be created after all nodes in the cluster are launched and have registered with the Consistent Core.

The coordinator assigns partitions to cluster nodes which are known at that point in time. If it's triggered every time a new cluster node is added, it might map partitions too early until the cluster reaches a stable state. This is why the coordinator should be configured to wait until the cluster reaches a minimum size.

The first time the partition assignment is done, it can simply be done in a round robin fashion. Ignite [bib-ignite] uses a more sophisticated mapping using [rendezvous_hashing] [bib-rendezvous_hashing]

class ClusterCoordinator…

```
CompletableFuture assignPartitionsToClusterNodes() {
  if (!minimumClusterSizeReached()) {
     return CompletableFuture.failedFuture(new NotEnoughClusterNode
  }
  return initializePartitionAssignment();
}

private boolean minimumClusterSizeReached() {
  return membership.getLiveMembers().size() >= MINIMUM_CLUSTER_SIZE
}

private CompletableFuture initializePartitionAssignment() {
  partitionAssignmentStatus = PartitionAssignmentStatus.IN_PROGRESS
  PartitionTable partitionTable = arrangePartitions();
  return replicatedLog.propose(new PartitiontableCommand(partitionT
}

public PartitionTable arrangePartitions() {
  PartitionTable partitionTable = new PartitionTable();
  List<Member> liveMembers = membership.getLiveMembers();
  for (int partitionId = 1; partitionId <= noOfPartitions; partitio
    int index = partitionId % liveMembers.size();
    Member member = liveMembers.get(index);
    partitionTable.addPartition(partitionId, new PartitionInfo(part
  }
  return partitionTable;
}
```

The replication log makes the partition table persistent.

class ClusterCoordinator…

```
PartitionTable partitionTable;
PartitionAssignmentStatus partitionAssignmentStatus = PartitionAssi

private void applyPartitionTableCommand(PartitiontableCommand comma
  this.partitionTable = command.partitionTable;
  partitionAssignmentStatus = PartitionAssignmentStatus.ASSIGNED;
  if (isLeader()) {
     sendMessagesToMembers(partitionTable);
  }
}
```

Once the partition assignment is persisted, the coordinator sends messages to all cluster nodes to tell each node which partitions it now owns.

class ClusterCoordinator…

```
List<Integer> pendingPartitionAssignments = new ArrayList<>();


private void sendMessagesToMembers(PartitionTable partitionTable) {
  Map<Integer, PartitionInfo> partitionsTobeHosted = partitionTable
  partitionsTobeHosted.forEach((partitionId, partitionInfo) -> {
    pendingPartitionAssignments.add(partitionId);
    HostPartitionMessage message = new HostPartitionMessage(request
    logger.info("Sending host partition message to " + partitionInf
    scheduler.execute(new RetryableTask(partitionInfo.hostedOn, net
  });
}
```

The controller will keep trying to reach nodes continuously until its message is successful.

class RetryableTask…

```
static class RetryableTask implements Runnable {
  Logger logger = LogManager.getLogger(RetryableTask.class);
  InetAddressAndPort address;
  Network network;
  ClusterCoordinator coordinator;
  Integer partitionId;
  int attempt;
  private Message message;

  public RetryableTask(InetAddressAndPort address, Network network,
    this.address = address;
    this.network = network;
    this.coordinator = coordinator;
    this.partitionId = partitionId;
    this.message = message;
  }

  @Override
  public void run() {
    attempt++;
    try {
       //stop trying if the node is failed.
       if (coordinator.isSuspected(address)) {
          return;
```

```
   }
   logger.info("Sending " + message + " to=" + address);
   network.send(address, message);
  } catch (Exception e) {
   logger.error("Error trying to send ");
   scheduleWithBackOff();
  }
 }

 private void scheduleWithBackOff() {
  scheduler.schedule(this, getBackOffDelay(attempt), TimeUnit.MIL
 }


 private long getBackOffDelay(int attempt) {
  long baseDelay = (long) Math.pow(2, attempt);
  long jitter = randomJitter();
  return baseDelay + jitter;
 }

 private long randomJitter() {
  int i = new Random(1).nextInt();
  i = i < 0 ? i * -1 : i;
  long jitter = i % 50;
  return jitter;
 }
}
```

When cluster node receives the request to create the partition, it creates one with the given partition id. If we imagine this happening within a simple key-value store, its implementation will look something like this:

class KVStore…

```
Map<Integer, Partition> allPartitions = new ConcurrentHashMap<>();
private void handleHostPartitionMessage(Message message) {
 Integer partitionId = ((HostPartitionMessage) message).getPartiti
 addPartitions(partitionId);
```

```
  logger.info("Adding partition " + partitionId + " to " + listenAd
  network.send(message.from, new HostPartitionAcks(message.messageI
}

public void addPartitions(Integer partitionId) {
  allPartitions.put(partitionId, new Partition(partitionId));

}
```

class Partition…

```
SortedMap<String, String> kv = new TreeMap<>();
private Integer partitionId;
```

Once the coordinator receives the message that the partition has been successfully created, it persists it in the replicated log and updates the partition status to be online.

class ClusterCoordinator…

```
private void handleHostPartitionAck(Message message) {
  int partitionId = ((HostPartitionAcks) message).getPartitionId();
  pendingPartitionAssignments.remove(Integer.valueOf(partitionId));
  logger.info("Received host partition ack from " + message.from +
  CompletableFuture future = replicatedLog.propose(new UpdatePartit
  future.join();
}
```

Once the *High-Water Mark* is reached, and the record is applied, the partition's status will be updated.

class ClusterCoordinator…

```
private void updateParitionStatus(UpdatePartitionStatusCommand comm
  removePendingRequest(command.partitionId);
  logger.info("Changing status for " + command.partitionId + " to "
  logger.info(partitionTable.toString());
  partitionTable.updateStatus(command.partitionId, command.status);
}
```

Client Interface

If we again consider the example of a simple key and value store, if a client needs to store or get a value for a particular key, it can do so by following these steps:

- The client applies the hash function to the key and finds the relevant partition based on the total number of partitions.
- The client gets the partition table from the coordinator and finds the cluster node that is hosting the partition. The client also periodically refreshes the partition table.

> Kafka [bib-kafka] faced an issue [bib-kafka-metadata-issue] in which all the producer/consumers were fetching partition metadata from Zookeeper [bib-zookeeper] and decided to make metadata available on all the brokers.
>
> A similar issue [bib-yb-metadata-issue] was also observed in YugabyteDB [bib-yugabyte]

Clients fetching a partition table from the coordinator can quickly lead to bottlenecks, especially if all requests are being served by a single coordinator leader. That is why it is common practice to keep metadata available on all cluster nodes. The coordinator can either push metadata to cluster nodes, or cluster nodes can pull it from the coordinator. Clients can then connect with any cluster node to refresh the metadata.

This is generally implemented inside the client library provided by the key value store, or by client request handling (which happens on the cluster nodes.)

class Client…

```
public void put(String key, String value) throws IOException {
  Integer partitionId = findPartition(key, noOfPartitions);
  InetAddressAndPort nodeAddress = getNodeAddressFor(partitionId);
  sendPutMessage(partitionId, nodeAddress, key, value);
}

private InetAddressAndPort getNodeAddressFor(Integer partitionId) {
  PartitionInfo partitionInfo = partitionTable.getPartition(partiti
  InetAddressAndPort nodeAddress = partitionInfo.getAddress();
  return nodeAddress;
}

private void sendPutMessage(Integer partitionId, InetAddressAndPort
  PartitionPutMessage partitionPutMessage = new PartitionPutMessage
  SocketClient socketClient = new SocketClient(address);
  socketClient.blockingSend(new RequestOrResponse(RequestId.Partiti
                            JsonSerDes.serialize(partitionPutMessage
}

public String get(String key) throws IOException {
  Integer partitionId = findPartition(key, noOfPartitions);
  InetAddressAndPort nodeAddress = getNodeAddressFor(partitionId);
  return sendGetMessage(partitionId, key, nodeAddress);
}

private String sendGetMessage(Integer partitionId, String key, Inet
  PartitionGetMessage partitionGetMessage = new PartitionGetMessage
  SocketClient socketClient = new SocketClient(address);
  RequestOrResponse response = socketClient.blockingSend(new Reques
  PartitionGetResponseMessage partitionGetResponseMessage = JsonSer
  return partitionGetResponseMessage.getValue();
}
```

Moving partitions to newly added members

When new nodes are added to a cluster, some partitions can be moved to other nodes. This can be done automatically once a new cluster node is added. But it can involve a lot of data being moved across the cluster node, which is why an administrator will typically trigger the repartitioning. One simple method to do this is to calculate the average number of partitions each node should host and then move the additional partitions to the new node. For example, if the number of partitions is 30 and there are three existing nodes in the cluster, each node should host 10 partitions. If a new node is added, the average per node is about 7. The coordinator will therefore try to move three partitions from each cluster node to the new one.

class ClusterCoordinator…

```
List<Migration> pendingMigrations = new ArrayList<>();

boolean reassignPartitions() {
  if (partitionAssignmentInProgress()) {
     logger.info("Partition assignment in progress");
     return false;
  }
  List<Migration> migrations = repartition(this.partitionTable);
  CompletableFuture proposalFuture = replicatedLog.propose(new Migr
  proposalFuture.join();
  return true;
}

public List<Migration> repartition(PartitionTable partitionTable) {
  int averagePartitionsPerNode = getAveragePartitionsPerNode();
  List<Member> liveMembers = membership.getLiveMembers();
  var overloadedNodes = partitionTable.getOverloadedNodes(averagePa
  var underloadedNodes = partitionTable.getUnderloadedNodes(average

  var migrations = tryMovingPartitionsToUnderLoadedMembers(averageP
  return migrations;
}
private List<Migration> tryMovingPartitionsToUnderLoadedMembers(int
                                            Map<InetAddressAndPort,
                                            Map<InetAddressAndPort,
  List<Migration> migrations = new ArrayList<>();
  for (InetAddressAndPort member : overloadedNodes.keySet()) {
    var partitions = overloadedNodes.get(member);
    var toMove = partitions.subList(averagePartitionsPerNode, parti
    overloadedNodes.put(member, partitions.subList(0, averagePartit
    ArrayDeque<Integer> moveQ = new ArrayDeque<Integer>(toMove.part
    while (!moveQ.isEmpty() && nodeWithLeastPartitions(underloadedN
      assignToNodesWithLeastPartitions(migrations, member, moveQ, u
   }
   if (!moveQ.isEmpty()) {
      overloadedNodes.get(member).addAll(moveQ);
   }
  }
  return migrations;
}

int getAveragePartitionsPerNode() {
  return noOfPartitions / membership.getLiveMembers().size();
}
```

The coordinator will persist the computed migrations in the replicated log and then send requests to move partitions across the cluster nodes.

```
private void applyMigratePartitionCommand(MigratePartitionsCommand
  logger.info("Handling partition migrations " + command.migrations
  for (Migration migration : command.migrations) {
    RequestPartitionMigrationMessage message = new RequestPartition
    pendingMigrations.add(migration);
    if (isLeader()) {
       scheduler.execute(new RetryableTask(migration.fromMember, ne
    }
  }
}
```

When a cluster node receives a request to migrate, it will mark the partition as migrating. This stops any further modifications to the partition. It will then send the entire partition data to the target node.

class KVStore…

```
private void handleRequestPartitionMigrationMessage(RequestPartitio
  Migration migration = message.getMigration();
  Integer partitionId = migration.getPartitionId();
  InetAddressAndPort toServer = migration.getToMember();
  if (!allPartitions.containsKey(partitionId)) {
     return;// The partition is not available with this node.
  }
  Partition partition = allPartitions.get(partitionId);
  partition.setMigrating();
  network.send(toServer, new MovePartitionMessage(requestNumber++,
}
```

The cluster node that receives the request will add the new partition to itself and return an acknowledgement.

class KVStore…

```
private void handleMovePartition(Message message) {
  MovePartitionMessage movePartitionMessage = (MovePartitionMessage
  Partition partition = movePartitionMessage.getPartition();
  allPartitions.put(partition.getId(), partition);
  network.send(message.from, new PartitionMovementComplete(message.
      new Migration(movePartitionMessage.getMigrateFrom(), movePart
}
```

The cluster node previously owned the partition will then send the migration complete message to the cluster coordinator.

class KVStore…

```
private void handlePartitionMovementCompleteMessage(PartitionMoveme
  allPartitions.remove(message.getMigration().getPartitionId());
  network.send(coordLeader, new MigrationCompleteMessage(requestNum
      message.getMigration()));
}
```

The cluster coordinator will then mark the migration as complete. The change will be stored in the replicated log.

class ClusterCoordinator…

```
private void handleMigrationCompleteMessage(MigrationCompleteMessag
  MigrationCompleteMessage migrationCompleteMessage = message;
  CompletableFuture propose = replicatedLog.propose(new MigrationCo
  propose.join();
}
```

class ClusterCoordinator…

```
private void applyMigrationCompleted(MigrationCompletedCommand comm
  pendingMigrations.remove(command.getMigration());
  logger.info("Completed migration " + command.getMigration());
  logger.info("pendingMigrations = " + pendingMigrations);
  partitionTable.migrationCompleted(command.getMigration());
}
```

class PartitionTable…

```
public void migrationCompleted(Migration migration) {
  this.addPartition(migration.partitionId, new PartitionInfo(migrat
}
```

Alternative Solution-Partitions proportional to number of nodes

There is an alternative to fixed partitions, as popularized by Cassandra [bib-cassandra], is to have the number of partitions proportional to the number of nodes in the cluster.

The number of partitions increases when new nodes are added to the cluster. This technique is also sometimes called as [consistent-hashing] [bib-consistent-hashing] It requires storing a randomly generated hash per partition, and needs to search through the sorted list of hashes, taking more time compared to O(1) computation of no. of partitions % hash. This technique is also shown to create some imbalance in data assigned to partitions, so most data systems use fixed-partitions technique.

The basic mechanism works as following. Each node is assigned a random integer token. This value is typically generated as hash of a random GUID. For example, Cassandra [bib-cassandra] generates it as following:

```
static final Random random = new Random();
public static ByteBuffer guidAsBytes()
{
  String s_id = getLocalHost();
  StringBuilder sbValueBeforeMD5 = new StringBuilder();
  long rand = random.nextLong();
  sbValueBeforeMD5.append(s_id)
      .append(":")
      .append(Long.toString(System.currentTimeMillis()))
      .append(":")
      .append(Long.toString(rand));

  String valueBeforeMD5 = sbValueBeforeMD5.toString();
  return ByteBuffer.wrap(hash(valueBeforeMD5.getBytes()));
}

private static String getLocalHost() {
  String s_id = null;
  try {
     s_id = InetAddress.getLocalHost().toString();
  } catch (UnknownHostException e) {
     throw new RuntimeException(e);
  }
  return s_id;
}

public static byte[] hash(byte[] data)
{
  byte[] result = null;
  try
  {
    MessageDigest messageDigest = MessageDigest.getInstance("MD5");
    result = messageDigest.digest(data);

  } catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(e);
  }
  return result;
}
```

The client maps the key to the cluster nodes as following.

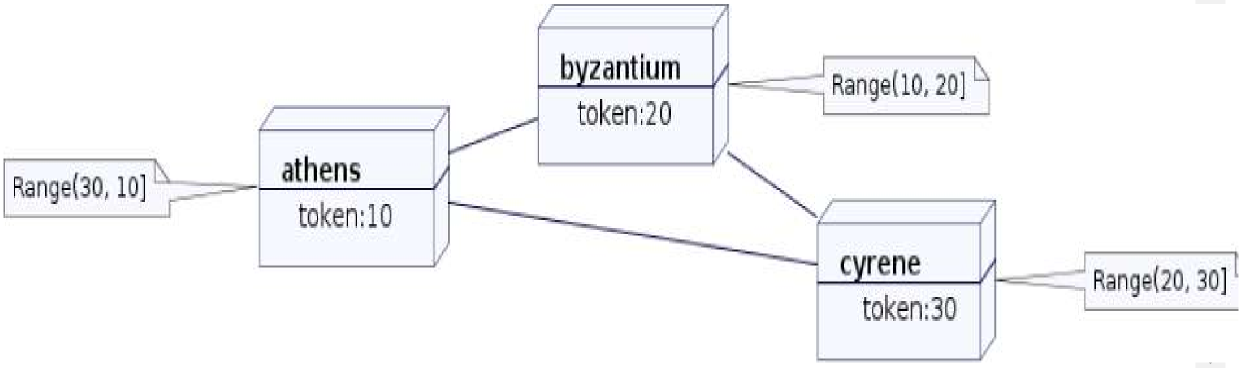
- It computes the hash of the key
- It then gets the sorted list of all the available tokens. It then searches the lowest token value, that is higher than the hash of the key. The cluster node owning that token is the node storing the given key.
- Because the list is considered as circular, any hash value of the key, greater than the last token in the list, maps to the first token.

The code for this looks as following:

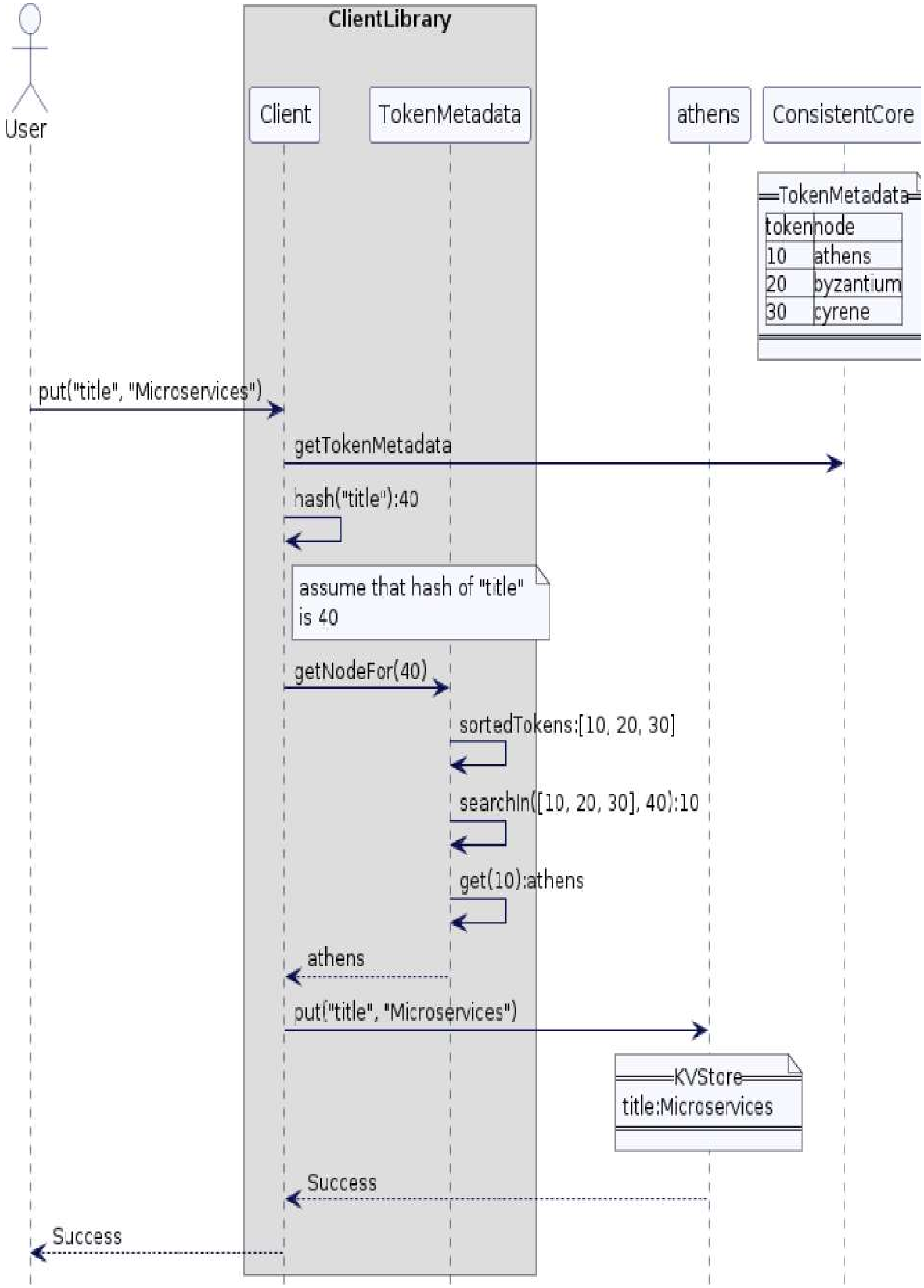
class TokenMetadata…

```
public Node getNodeFor(BigInteger keyHash) {
  List<BigInteger> tokens = sortedTokens();
  BigInteger token = searchToken(tokens, keyHash);
  return tokenToNodeMap.get(token);
}

private static BigInteger searchToken(List<BigInteger> tokens, BigI
  int index = Collections.binarySearch(tokens, keyHash);
  if (index < 0) {
     index = (index + 1) * (-1);
     if (index >= tokens.size())
        index = 0;
  }
  BigInteger token = tokens.get(index);
  return token;
}

List<BigInteger> sortedTokens() {
  List<BigInteger> tokens = new ArrayList<>(tokenToNodeMap.keySet()
  Collections.sort(tokens);
  return tokens;
}
```

To see how this works we take some example values for tokens. Consider a three node cluster, with athens, byzantium and cyrene each having token values as 10, 20 and 30 respectively.



For this example, imagine this metadata stored with the *Consistent Core*. The client library gets the token metadata, and uses it to map given key to the cluster node.

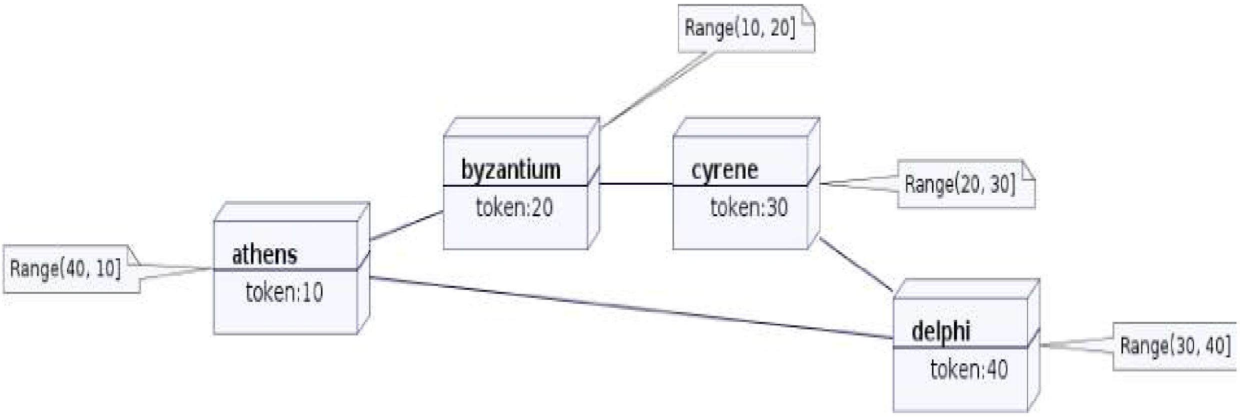


TokenMetadata

token	node
10	athens
20	byzantium
30	cyrene

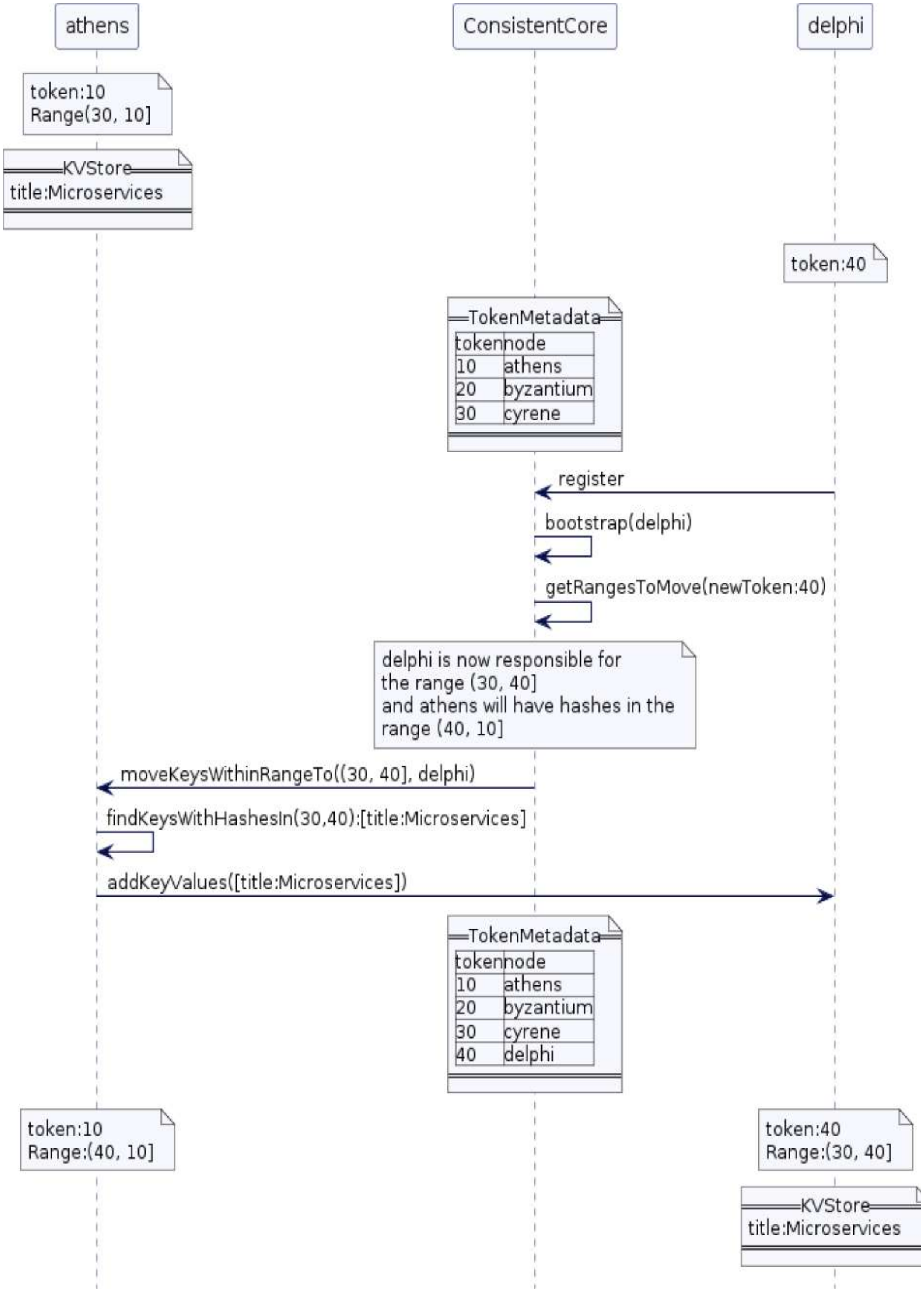
Adding new node to the cluster

The main advantage of this scheme is that we have more partitions when new nodes are added to the cluster.



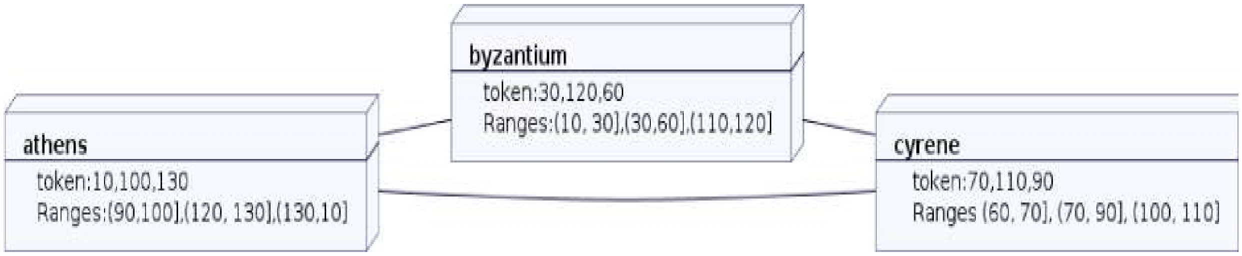
Consider a new node, delphi added to the cluster, with a random token assigned to it as 40. We can see that athens, which was hosting all the keys with hashes above 30, now needs to move the keys with hashes between 30 and 40 to the new node. So all the keys do not need to be moved, but only a small portion needs to be moved to the new node.

As before, lets consider a *Consistent Core* is tracking the cluster membership and mapping partitions to the cluster nodes. When delphi registers with the Consistent Core, it first figures out which existing nodes are affected because of this new addition. In our example, athens needs to move part of the data to the new node. The Consistent Core tells athens to move all the keys with hashes between 30 and 40 to delphi. After the movement is complete, delphi’s token is added to token metadata.

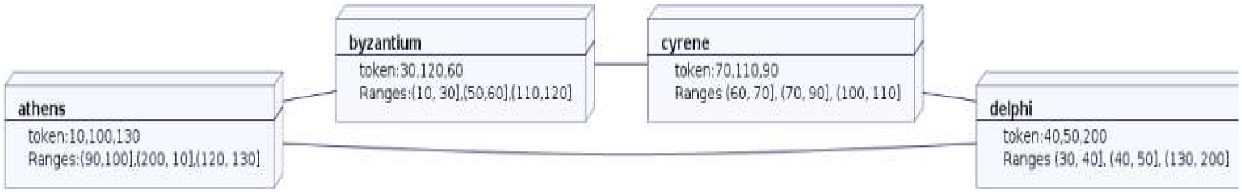

athens
ConsistentCore
delphi
token:10
Range(30, 10]
KVStore
title:Microservices
token:40
TokenMetadata
token node
10 athens
20 byzantium
30 cyrene
register
bootstrap(delphi)
getRangesToMove(newToken:40)
delphi is now responsible for
the range (30, 40]
and athens will have hashes in the
range (40, 10]
moveKeysWithinRangeTo((30, 40], delphi)
findKeysWithHashesIn(30,40):[title:Microservices]
addKeyValues([title:Microservices])
TokenMetadata
token node
10 athens
20 byzantium
30 cyrene
40 delphi
token:10
Range:(40, 10]
token:40
Range:(30, 40]
KVStore
title:Microservices

This basic technique of assigning a single token to each node has shown to create data imbalance. When a new node is added, it also puts the burden of moving data on one of the existing nodes. For this reason Cassandra [bib-cassandra] changed its design to have multiple random tokens [bib-cassandra-vnode] assigned to nodes. This allows more even distribution of data. When a new node is added to the cluster, a small amount of data is moved from multiple existing nodes, avoiding a load on single node.

Considering the same example as above, instead of single token, each of athens, byzantium and cyrene can have three tokens each. Three number of tokens is taken to simplify the example. The default value for Cassandra [bib-cassandra] was 256. The tokens are randomly allocated to nodes. It is important to note that tokens assigned to nodes are randomly generated GUID hashes, so they are not contiguous. If contiguous numbers like 10,20,30 are assigned to each node, it will have similar problem as single token per node, when a new node is added.


byzantium
token:30,120,60
Ranges:(10, 30],(30,60],(110,120]
athens
token:10,100,130
Ranges:(90,100],(120, 130],(130,10]
cyrene
token:70,110,90
Ranges (60, 70], (70, 90], (100, 110]

When a new node delphi is added, with tokens say, 40, 50 and 200, The key ranges athens and byzantium are responsible for changes. Range (130,10] on athens is split with delphi owning keys with hashes between (130,200]. Range (30,60] on byzantium is split with delphi owning keys with hashes between (40,50] Keys in the range (130,200] from athens and (40,50] from byzantium are moved to delphi.


byzantium
token:30,120,60
Ranges:(10, 30],(50,60],(110,120]
cyrene
token:70,110,90
Ranges (60, 70], (70, 90], (100, 110]
athens
token:10,100,130
Ranges:(90,100],(200, 10],(120, 130]
delphi
token:40,50,200
Ranges (30, 40], (40, 50], (130, 200]

Examples

In Kafka [bib-kafka] each topic is created with a fixed number of partitions

Shard allocation in Akka [bib-akka-shard-allocation] has a fixed number of shards configured. The guideline is to have the number of shards to be 10 times the size of the cluster

In-memory data grid products like partitioning in Apache Ignite [bib-ignite-partitioning] and partitioning in Hazelcast [bib-hazelcast-partitioning] have a fixed number of partitions that are configured for their caches.

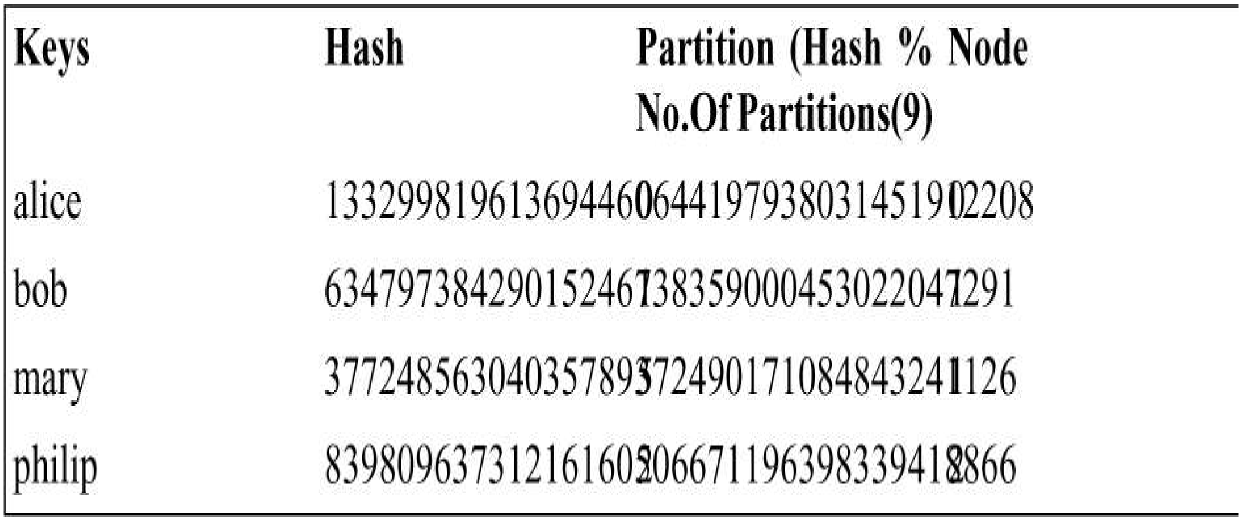
Chapter 20. Key-Range Partitions

Partition data in sorted key ranges to efficiently handle range queries.

Problem

To split data across a set of cluster nodes, each data item needs to be mapped to one. If users want to query a range of keys, specifying only the start and end key, all partitions will need to be queried in order for the values to be acquired. Querying every partition for a single request is far from optimal.

If we take an key value store example, we can store the author names using hash based mapping. (as used in *Fixed Partitions*).



Keys	Hash	Partition (Hash % Node No.Of Partitions(9)
alice	133299819613694460644197938031451902208	
bob	63479738429015246738359000453022047291	
mary	37724856304035789372490171084843241126	
philip	83980963731216160506671196398339418866	

If a user wants to get values for a range of names, - beginning with, say, letter ‘a’ to ‘f’ - there’s no way to know which partitions we should fetch data from if the hash of the key is being used to map keys to partitions. All partitions need to be queried to get the values required.

Solution

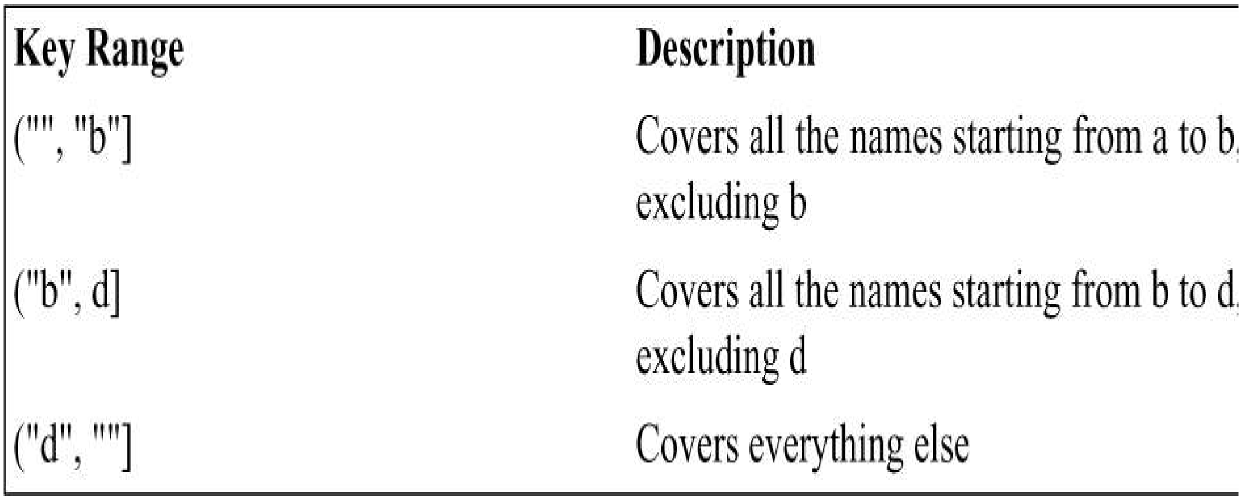
Create logical partitions for keys ranges in a sorted order. The partitions can then be mapped to cluster nodes. To query a range of data, the client can get all partitions that contain keys from a given range and query only those specific partitions to get the values required.

Predefining key ranges

If we already know the whole key space and distribution of keys, the ranges for partitions can be specified upfront.

Let’s return to our simple key value store with string keys and values. In this example we are storing author names and their books. If we know the author name distribution upfront, we can then define partition splits at specific letters - let’s say, in this instance, ‘b’ and ‘d’.

The start and end of the entire key range needs to be specifically marked. We can use an empty string to mark the lowest and the highest key. The ranges will be created like this:



Key Range	Description
("", "b"]	Covers all the names starting from a to b, excluding b
("b", d]	Covers all the names starting from b to d, excluding d
("d", ""]	Covers everything else

The range will be represented by a start and an end key

class Range…

```
private String startKey;
private String endKey;
```

The cluster coordinator creates ranges from the specified split points. The partitions will then be assigned to cluster nodes.

class ClusterCoordinator…

```
PartitionTable createPartitionTableFor(List<String> splits) {
  List<Range> ranges = createRangesFromSplitPoints(splits);
  return arrangePartitions(ranges, membership.getLiveMembers());
}

List<Range> createRangesFromSplitPoints(List<String> splits) {
  List<Range> ranges = new ArrayList<>();
  String startKey = Range.MIN_KEY;
  for (String split : splits) {
    String endKey = split;
    ranges.add(new Range(startKey, endKey));
    startKey = split;
  }
  ranges.add(new Range(startKey, Range.MAX_KEY));
  return ranges;
}
PartitionTable arrangePartitions(List<Range> ranges, List<Member> l
  PartitionTable partitionTable = new PartitionTable();
  for (int i = 0; i < ranges.size(); i++) {
    //simple round-robin assignment.
    Member member = liveMembers.get(i % liveMembers.size());
    int partitionId = newPartitionId();
    Range range = ranges.get(i);
    PartitionInfo partitionInfo = new PartitionInfo(partitionId, me
    partitionTable.addPartition(partitionId, partitionInfo);
  }
  return partitionTable;
}
```

The consistent core, acting as a cluster coordinator, stores the mapping in a fault tolerant way by using a *Replicated Log*. The implementation is similar to the one explained in the pattern fixed partitions. [fixedpartitions.xhtml#MappingThePartitionsToClusterNodes]

Client Interface

If a client needs to store or get a value for a particular key in a key-value store, it needs to follow these steps

class Client…

```
public List<String> getValuesInRange(Range range) throws IOExceptio
  PartitionTable partitionTable = getPartitionTable();
  List<PartitionInfo> partitionsInRange = partitionTable.getPartiti
  List<String> values = new ArrayList<>();
  for (PartitionInfo partitionInfo : partitionsInRange) {
    List<String> partitionValues = sendGetRangeMessage(partitionInf
    values.addAll(partitionValues);
  }
  return values;
}

private PartitionTable getPartitionTable() throws IOException {
  GetPartitionTableResponse response = sendGetPartitionTableRequest
  return response.getPartitionTable();
}

private List<String> sendGetRangeMessage(int partitionId, Range ran
  GetAllInRangeRequest partitionGetMessage = new GetAllInRangeReque
  GetAllInRangeResponse response = sendGetRangeRequest(address, par
  return response.getValues();
}
```

class PartitionTable…

```
public List<PartitionInfo> getPartitionsInRange(Range range) {
  List<PartitionInfo> allPartitions = getAllPartitions();
  List<PartitionInfo> partitionsInRange = allPartitions.stream().fi
  return partitionsInRange;
}
```

class Range…

```
public boolean isOverlapping(Range range) {
  return this.contains(range.startKey) || range.contains(this.start
}

public boolean contains(String key) {
  return key.compareTo(startKey) >= 0 &&
       (endKey.equals(Range.MAX_KEY) || endKey.compareTo(key) > 0);
}
```

class Partition…

```
public List<String> getAllInRange(Range range) {
  return kv.subMap(range.getStartKey(), range.getEndKey()).values()
}
```

Storing a value

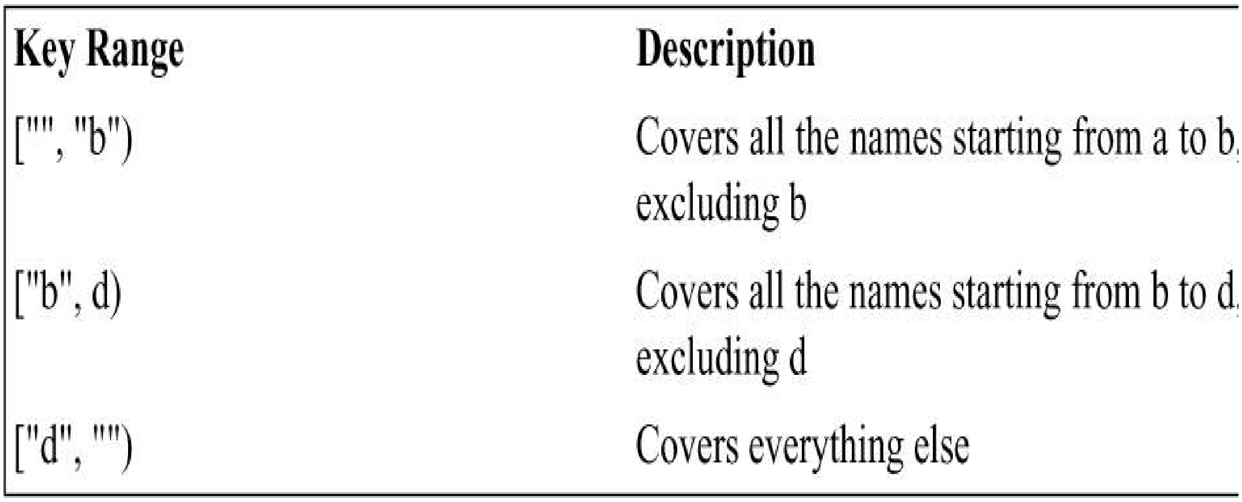
To store a key value, the client needs to find the right partition for a given key. Once a partition is found, the request is sent to the cluster node that is hosting that partition.

class Client…

```
public void put(String key, String value) throws IOException {
  PartitionInfo partition = findPartition(key);
  sendPutMessage(partition.getPartitionId(), partition.getAddress()
}

private PartitionInfo findPartition(String key) {
  return partitionTable.getPartitionFor(key);
}
```

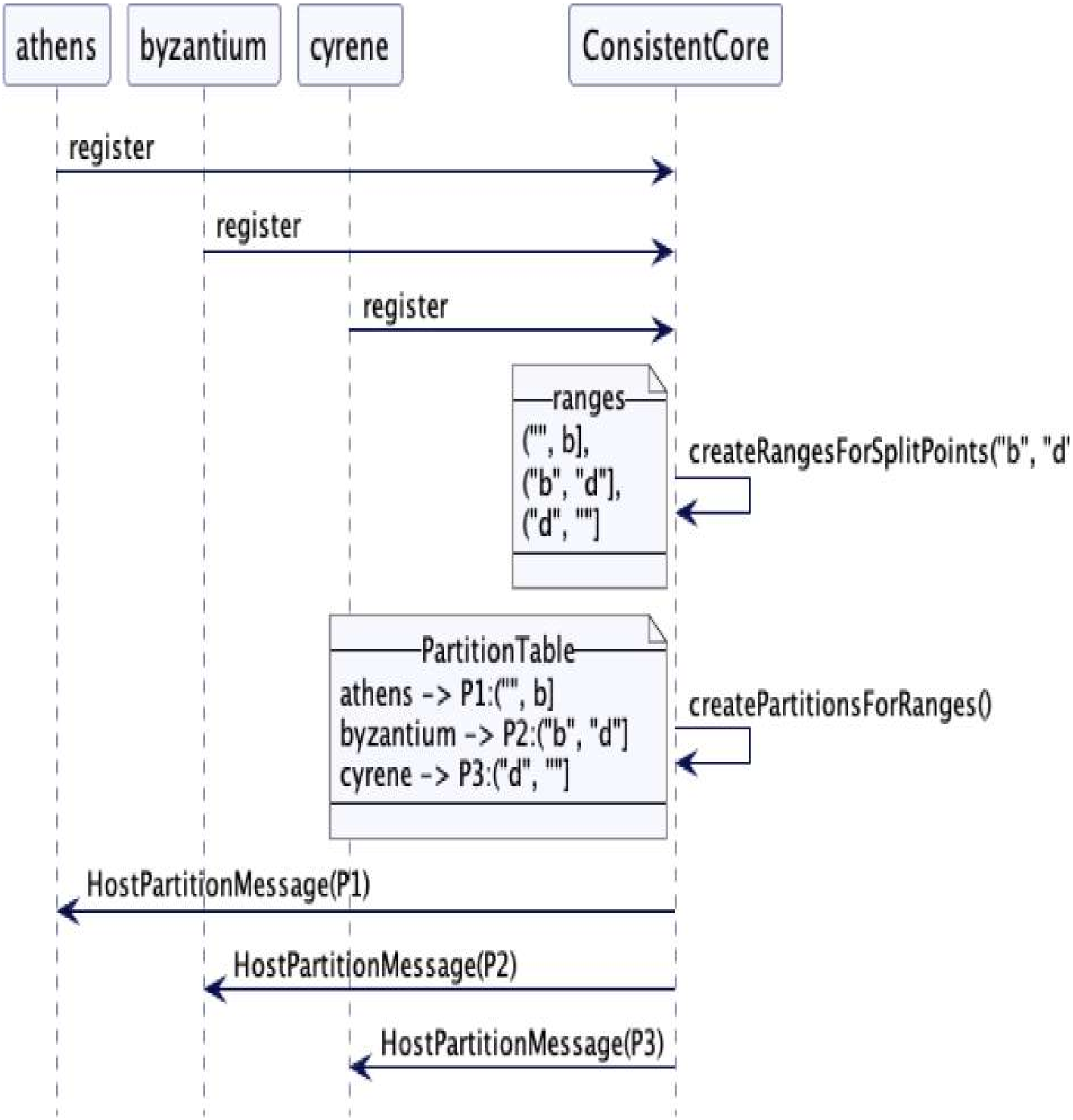
An example scenario

Let’s explore this with another example.Consider three data servers: athens, byzantium and cyrene. The partitions splits are defined as "b" and "d". The three ranges will be created like this:

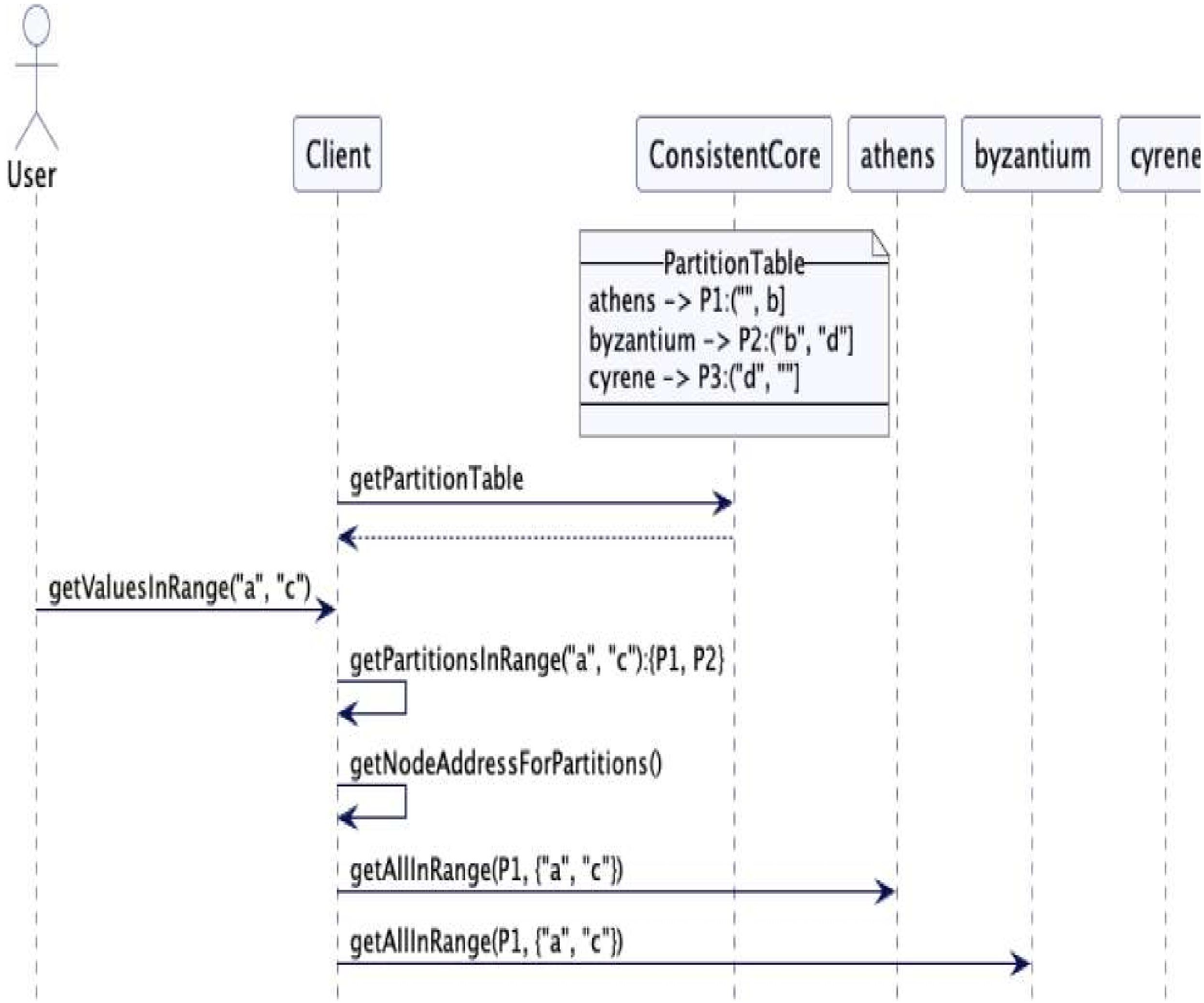


Key Range	Description
["", "b")	Covers all the names starting from a to b, excluding b
["b", d)	Covers all the names starting from b to d, excluding d
["d", "")	Covers everything else

The coordinator then creates three partitions for these ranges and maps them to the cluster nodes.



If a client now wants to get all the values for names starting with "a" and "c", it gets all the partitions which have key ranges containing keys starting with "a" and "c". It then sends requests to only those partitions to get the values.


User
Client
ConsistentCore
athens
byzantium
cyrene
PartitionTable
athens -> P1:("", b]
byzantium -> P2:("b", "d"]
cyrene -> P3:("d", ""]
getPartitionTable
getValuesInRange("a", "c")
getPartitionsInRange("a", "c"):{P1, P2}
getNodeAddressForPartitions()
getAllInRange(P1, {"a", "c"})
getAllInRange(P1, {"a", "c"})

Auto splitting ranges

Often it can be difficult to know what the suitable split points are upfront.In these instances, we can implement auto-splitting.

Here, the coordinator will create only one partition with a key range which includes all the key space.

class ClusterCoordinator…

```
private CompletableFuture initializeRangePartitionAssignment(List<S
  partitionAssignmentStatus = PartitionAssignmentStatus.IN_PROGRESS
  PartitionTable partitionTable = splits.isEmpty() ?
     createPartitionTableWithOneRange():createPartitionTableFor(spl
  return replicatedLog.propose(new PartitiontableCommand(partitionT
}

public PartitionTable createPartitionTableWithOneRange() {
  PartitionTable partitionTable = new PartitionTable();
  List<Member> liveMembers = membership.getLiveMembers();
  Member member = liveMembers.get(0);
  Range firstRange = new Range(Range.MIN_KEY, Range.MAX_KEY);
  int partitionId = newPartitionId();
  partitionTable.addPartition(partitionId, new PartitionInfo(partit
  return partitionTable;
}
```

Each partition can be configured with a fixed maximum size. A background task then runs on each cluster node to track the size of the partitions. When a partition reaches its maximum size, it's split into two partitions, each one being approximately half the size of the original.

class KVStore…

```
public void scheduleSplitCheck() {
  scheduler.scheduleAtFixedRate(()->{
    splitCheck();
  }, 1000, 1000, TimeUnit.MILLISECONDS);
}

public void splitCheck() {
  for (Integer partitionId : allPartitions.keySet()) {
    splitCheck(allPartitions.get(partitionId));
  }
}

int MAX_PARTITION_SIZE = 1000;
public void splitCheck(Partition partition) {
  String middleKey = partition.getMiddleKeyIfSizeCrossed(MAX_PARTIT
  if (!middleKey.isEmpty()) {
     logger.info("Partition " + partition.getId() + " reached size
     network.send(coordLeader, new SplitTriggerMessage(partition.ge
  }
}
```

Calculating partition size and Finding the middle key

Scanning the complete partition to find the split key is resource intensive. This is why databases like TiKV [bib-tikv] store the size of the partition and corresponding key in the data store. The middle key can then be found without scanning the full partition.

Databases like YugabyteDB [bib-yb] or [hbase] [bib-hbase] which use a store per partition find an approximate mid key by scanning through the metadata of the store files.

Getting the size of the partition and finding the middle key is dependent on what storage engines are being used. A simple way of dong this can be to just scan through the entire partition to calculate its size. TiKV [bib-tikv] initially used this approach. To be able to split the tablet, the key which is situated at the mid point needs to be found as well. To avoid scanning through the partition twice, a simple implementation can get the middle key if the size is more than the configured maximum.

class Partition…

```
public String getMiddleKeyIfSizeCrossed(int partitionMaxSize) {
  int kvSize = 0;
  for (String key : kv.keySet()) {
    kvSize += key.length() + kv.get(key).length();
    if (kvSize >= partitionMaxSize / 2) {
       return key;
    }
  }
  return "";
}
```

The coordinator, handling the split trigger message update the key range metadata for the original partition, and creates a new partition metadata for the split range.

class ClusterCoordinator…

```
private void handleSplitTriggerMessage(SplitTriggerMessage message)
  logger.info("Handling SplitTriggerMessage " + message.getPartitio
  splitPartition(message.getPartitionId(), message.getSplitKey());
}
public CompletableFuture splitPartition(int partitionId, String spl
  logger.info("Splitting partition " + partitionId + " at key " + s
  PartitionInfo parentPartition = partitionTable.getPartition(parti
  Range originalRange = parentPartition.getRange();
  List<Range> splits = originalRange.split(splitKey);
  Range shrunkOriginalRange = splits.get(0);
  Range newRange = splits.get(1);
  return replicatedLog.propose(new SplitPartitionCommand(partitionI
}
```

After the partitions metadata is stored successfully, it sends a message to the cluster node that is hosting the parent partition to split the parent partition's data.

class ClusterCoordinator…

```
private void applySplitPartitionCommand(SplitPartitionCommand comma
  PartitionInfo originalPartition = partitionTable.getPartition(com
  Range originalRange = originalPartition.getRange();
  if (!originalRange.coveredBy(command.getUpdatedRange().getStartKe
     logger.error("The original range start and end keys "+ origina
     return;
  }

  originalPartition.setRange(command.getUpdatedRange());
  PartitionInfo newPartitionInfo = new PartitionInfo(newPartitionId
  partitionTable.addPartition(newPartitionInfo.getPartitionId(), ne

  //send requests to cluster nodes if this is the leader node.
  if (isLeader()) {
     var message = new SplitPartitionMessage(command.getOriginalPar
     scheduler.execute(new RetryableTask(originalPartition.getAddre
  }
}
```

class Range…

```
public boolean coveredBy(String startKey, String endKey) {
  return getStartKey().equals(startKey)
      && getEndKey().equals(endKey);
}
```

The cluster node splits the original partition and creates a new partition. The data from the original partition is then copied to the new partition. It then responds to the coordinator telling that the split is complete.

class KVStore…

```
private void handleSplitPartitionMessage(SplitPartitionMessage spli
  splitPartition(splitPartitionMessage.getPartitionId(),
                    splitPartitionMessage.getSplitKey(),
                    splitPartitionMessage.getSplitPartitionId());
  network.send(coordLeader,
      new SplitPartitionResponseMessage(splitPartitionMessage.getPa
          splitPartitionMessage.getPartitionId(),
          splitPartitionMessage.getSplitPartitionId(),
          splitPartitionMessage.messageId, listenAddress));
}

private void splitPartition(int parentPartitionId, String splitKey,
  Partition partition = allPartitions.get(parentPartitionId);
  Partition splitPartition = partition.splitAt(splitKey, newPartiti
  logger.info("Adding new partition " + splitPartition.getId() + "
  allPartitions.put(splitPartition.getId(), splitPartition);
}
```

class Partition…

```
public Partition splitAt(String splitKey, int newPartitionId) {
  List<Range> splits = this.range.split(splitKey);
  Range shrunkOriginalRange = splits.get(0);
  Range splitRange = splits.get(1);

  SortedMap<String, String> partition1Kv =
      (range.getStartKey().equals(Range.MIN_KEY))
           ? kv.headMap(splitKey)
           : kv.subMap(range.getStartKey(), splitKey);

  SortedMap<String, String> partition2Kv =
       (range.getEndKey().equals(Range.MAX_KEY))
            ? kv.tailMap(splitKey)
            : kv.subMap(splitKey, range.getEndKey());

  this.kv = partition1Kv;
  this.range = shrunkOriginalRange;

  return new Partition(newPartitionId, partition2Kv, splitRange);
}
```

class Range…

```
public List<Range> split(String splitKey) {
  return Arrays.asList(new Range(startKey, splitKey), new Range(spl
}
```

Once the coordinator receives the message, it marks the partitions as online

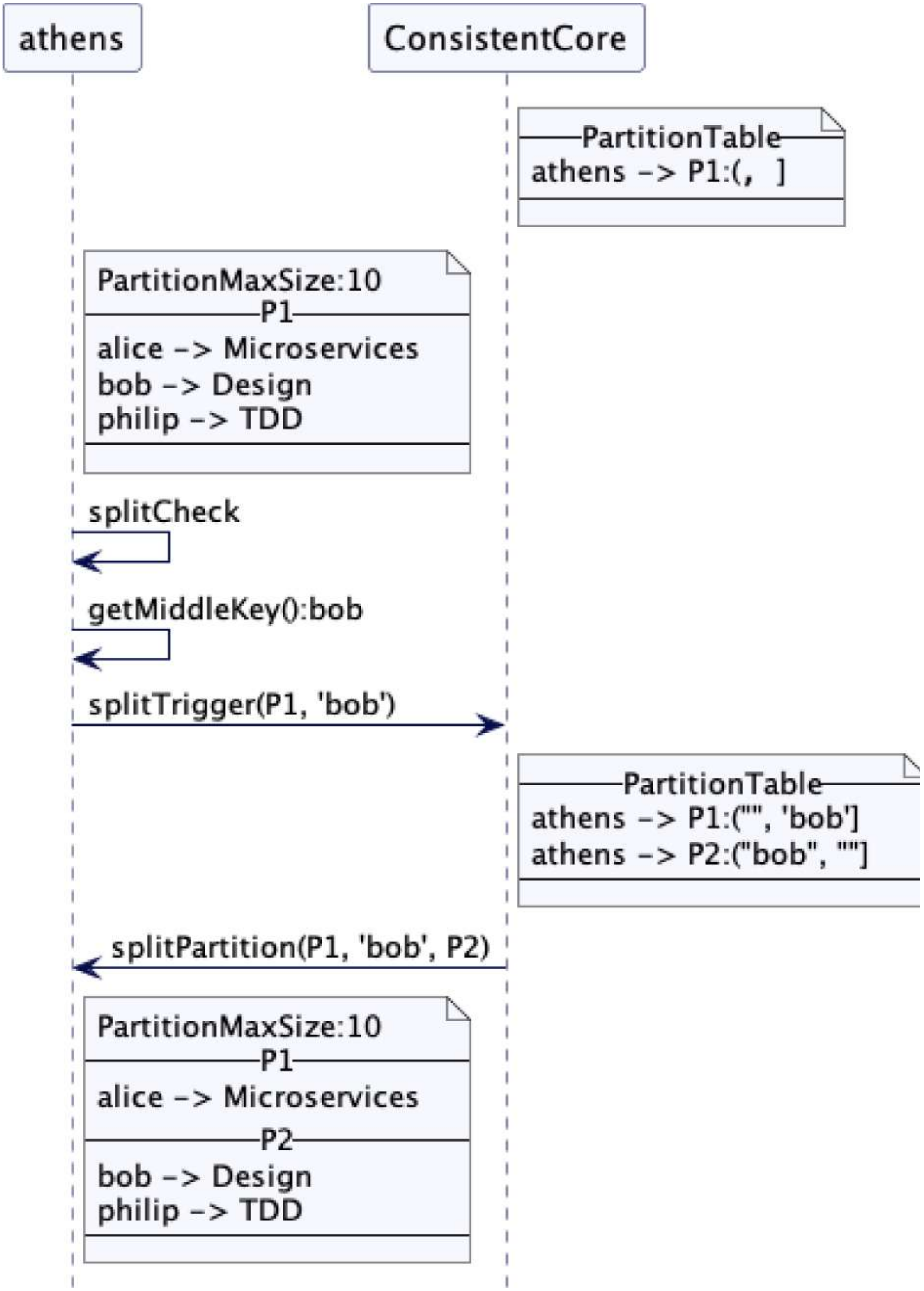
class ClusterCoordinator…

```
private void handleSplitPartitionResponse(SplitPartitionResponseMes
  replicatedLog.propose(new UpdatePartitionStatusCommand(message.ge
}
```

One of the possible issues that can arise when trying to modify the existing partition is that the client cannot cache and always needs to get the latest partition metadata before it can send any requests to the cluster node. Data stores use *Generation Clock* for partitions; this is updated every single time a partition is split. Any client requests with an older generation number will be rejected. Clients can then reload the partition table from the coordinator and retry the request. This ensures that clients that possess older metadata don't get the wrong results. YugabyteDB [bib-yb] chooses to create two separate new partitions and marks the original as explained in their Automatic table splitting design. [bib-yb-automatic-table-splitting].

Example Scenario

Consider an example where the cluster node athens holds partition P1 covering the entire key range. The maximum partition size is configured to be 10 bytes. The SplitCheck detects the size has grown beyond 10, and finds the approximate middle key to be bob. It then sends a message to the cluster coordinator, asking it to create metadata for the split partition. Once this metadata has been successfully created by the coordinator, the coordinator then asks athens to split partition P1 and passes it the partitionId from the metadata. Athens can then shrink P1 and create a new partition, copying the data from P1 to the new partition. After the partition has been successfully created it sends confirmation to the coordinator. The coordinator then marks the new partition as online.


athens
ConsistentCore
PartitionTable
athens -> P1:(,]
PartitionMaxSize:10
P1
alice -> Microservices
bob -> Design
philip -> TDD
splitCheck
getMiddleKey():bob
splitTrigger(P1, 'bob')
PartitionTable
athens -> P1:("", 'bob']
athens -> P2:("bob", ""]
splitPartition(P1, 'bob', P2)
PartitionMaxSize:10
P1
alice -> Microservices
P2
bob -> Design
philip -> TDD

Load based splitting

With auto-splitting, we only ever begin with one range. This means all client requests go to a single server even if there are other nodes in the cluster. All requests will continue to go to the single server that is hosting the single range until the range is split and moved to other servers. This is why sometimes splitting on parameters such as total nunmber of requests, or CPU, and memory usage are also used to trigger a partition split. Modern databases like CockroachDB [bib-cockroachdb] and YugabyteDB [bib-yb] support load based plitting. More details can be found in their documentation at [cockroach-load-splitting] [bib-cockroach-load-splitting] and [yb-load-splitting] [bib-yb-load-splitting]

Examples

Databases like [hbase] [bib-hbase], CockroachDB [bib-cockroachdb], YugabyteDB [bib-yb] and TiKV [bib-tikv] support range partitioning.

Chapter 21. Two Phase Commit

Update resources on multiple nodes in one atomic operation.

Problem

When data needs to be atomically stored on multiple cluster nodes, Cluster nodes cannot make the data accessible to clients before the decision of other cluster nodes is known. Each node needs to know if other nodes successfully stored the data or they failed.

Solution

Comparison with *Paxos* and *Replicated Log*

Paxos and Replicated log implementations also have two phases of execution. But the key difference is that these consensus algorithms are used when all the cluster nodes involved store the same values.

Two phase commit works across cluster nodes storing different values. For example, across different partitions of a database. Each partition can be using Replicated Log to replicate the state involved in two phase commit.

The essence of two phase commit, unsurprisingly, is that it carries out an update in two phases:

- the first, prepare, asks each node if it's able to promise to carry out the update

- the second, commit, actually carries it out.

As part of the prepare phase, each node participating in the transaction acquires whatever it needs to assure that it will be able to do the commit in the second phase, for instance any locks that are required. Once each node is able to ensure it can commit in the second phase, it lets the coordinator know, effectively promising the coordinator that it can and will commit in the second phase. If any node is unable to make that promise, then the coordinator tells all nodes to rollback, releasing any locks they have, and the transaction is aborted. Only if all the participants agree to go ahead does the second phase commence, at which point it's expected they will all successfully update.

Considering a simple distributed key value store implementation, the two phase commit protocol works as follows.

The transactional client creates a unique identifier called a transaction identifier. The client also keeps track of other details like the transaction start time. This is used, as described later by the locking mechanism, to prevent deadlocks. The unique id, along with the additional details like the start timestamp, that the client tracks is used to refer the transaction across the cluster nodes. The client maintains a transaction reference as follows, which is passed along with every request from the client to other cluster nodes.

class TransactionRef...

```
private UUID txnId;
private long startTimestamp;

  public TransactionRef(long startTimestamp) {
    this.txnId = UUID.randomUUID();
    this.startTimestamp = startTimestamp;
}
```

class TransactionClient...

```
TransactionRef transactionRef;

public TransactionClient(ReplicaMapper replicaMapper, SystemClock s
  this.clock = systemClock;
  this.transactionRef = new TransactionRef(clock.now());
  this.replicaMapper = replicaMapper;
}
```

One of the cluster nodes acts as a coordinator which tracks the status of the transaction on behalf of the client. In a key-value store, it is generally the cluster node holding data for one of the keys. It is generally picked up as the cluster node storing data for the first key used by the client.

Before storing any value, the client communicates with the coordinator to notify it about the start of the transaction. Because the coordinator is one of the cluster nodes storing values, it is picked up dynamically when the client initiates a get or put operation with a specific key.

class TransactionClient…

```
private TransactionalKVStore coordinator;
private void maybeBeginTransaction(String key) {
  if (coordinator == null) {
     coordinator = replicaMapper.serverFor(key);
     coordinator.begin(transactionRef);
  }
}
```

The transaction coordinator keeps track of the status of the transaction. It records every change in a *Write-Ahead Log* to make sure that the details are available in case of a crash.

class TransactionCoordinator…

```
Map<TransactionRef, TransactionMetadata> transactions = new Concurr
WriteAheadLog transactionLog;

public void begin(TransactionRef transactionRef) {
  TransactionMetadata txnMetadata = new TransactionMetadata(transac
  transactionLog.writeEntry(txnMetadata.serialize());
  transactions.put(transactionRef, txnMetadata);
}
```

class TransactionMetadata…

```
private TransactionRef txn;
private List<String> participatingKeys = new ArrayList<>();
private TransactionStatus transactionStatus;
```

> The example code shows that every put request goes to the respective servers. But because the values are not made visible until the transaction commits, they can very well be buffered on the client side until the client decides to commit, to optimize on the network round trips.

The client sends each key which is part of the transaction to the coordinator. This way the coordinator tracks all the keys which are part of the transaction. The coordinator records the keys which are part of the transaction in the transaction metadata.The keys then can be used to know about all of the cluster nodes which are part of the transaction. Because each key-value is generally replicated with the Replicated Log, the leader server handling the requests for a particular key might change over the lifetime of the transaction, so the keys are tracked instead of the actual server addresses. The client then sends the put or get requests to the server holding the data for the key. The server is picked based on the partitioning strategy. The thing to note is that the client directly communicates with the server and not through the coordinator. This avoids sending data twice over the network, from client to coordinator, and then from coordinator to the respective server.

The keys then can be used to know about all the cluster nodes which are part of the transaction. Because each key-value is generally replicated with Replicated Log, the leader server handling the requests for a particular key might change over the life time of the transaction, so keys are tracked, rather than the actual server addresses.

class TransactionClient…

```
public CompletableFuture<String> get(String key) {
  maybeBeginTransaction(key);
  coordinator.addKeyToTransaction(transactionRef, key);
  TransactionalKVStore kvStore = replicaMapper.serverFor(key);
  return kvStore.get(transactionRef, key);
}

public void put(String key, String value) {
  maybeBeginTransaction(key);
  coordinator.addKeyToTransaction(transactionRef, key);
  replicaMapper.serverFor(key).put(transactionRef, key, value);
}
```

class TransactionCoordinator…

```
public synchronized void addKeyToTransaction(TransactionRef transac
  TransactionMetadata metadata = transactions.get(transactionRef);
  if (!metadata.getParticipatingKeys().contains(key)) {
     metadata.addKey(key);
     transactionLog.writeEntry(metadata.serialize());
  }
}
```

The cluster node handling the request detects that the request is part of a transaction with the transaction ID. It manages the state of the transaction, where it stores the key and the value in the request. The key values are not directly made available to the key value store, but stored separately.

class TransactionalKVStore…

```
public void put(TransactionRef transactionRef, String key, String v
  TransactionState state = getOrCreateTransactionState(transactionR
  state.addPendingUpdates(key, value);
}
```

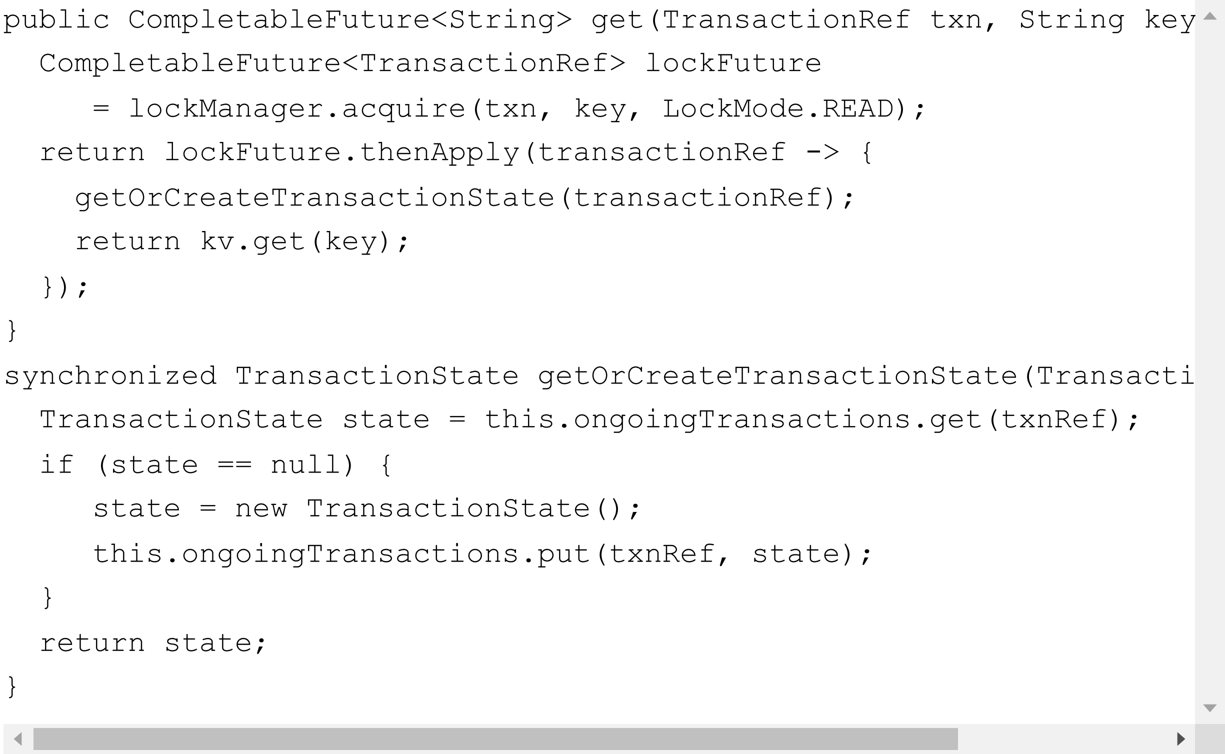
Locks and Transaction Isolation

Problems with non-serializable isolation

Because the serializable isolation levels has impact on overall performance, mostly because of the locks held for the duration of the transaction, most data stores provide relaxed isolation levels, where the locks are released earlier. This is a problem particularly when clients need to do read-modify-write operations. The operations can potentially overwrite the values from the previous transactions. So modern datastores like Spanner [bib-spanner] or CockroachDB [bib-cockroachdb] provide serializable isolation.

The requests also take a lock on the keys. Particularly, the get requests take a read lock and the put requests take a write lock. The read locks are taken as the values are read.

class TransactionalKVStore…



```
public CompletableFuture<String> get(TransactionRef txn, String key
  CompletableFuture<TransactionRef> lockFuture
     = lockManager.acquire(txn, key, LockMode.READ);
  return lockFuture.thenApply(transactionRef -> {
    getOrCreateTransactionState(transactionRef);
    return kv.get(key);
  });
}
synchronized TransactionState getOrCreateTransactionState(Transacti
  TransactionState state = this.ongoingTransactions.get(txnRef);
  if (state == null) {
     state = new TransactionState();
     this.ongoingTransactions.put(txnRef, state);
  }
  return state;
}
```

The write locks can be taken only when the transaction is about to commit and the values are to be made visible in the key value store. Until then, the cluster node can just track the modified values as pending operations.

Delaying locking decreases the chances of conflicting transactions.

class TransactionalKVStore…

```
public void put(TransactionRef transactionRef, String key, String v
  TransactionState state = getOrCreateTransactionState(transactionR
  state.addPendingUpdates(key, value);
}
```

> The key design decision is about which values are made visible to the concurrent transactions. Different transaction isolation levels give different levels of visibility. For example in strictly serial transactions, read requests are blocked till the transaction doing the write completes. To improve performance, data stores can get around two-phase-locking, and release locks earlier. But then consistency of the data is compromised. There are a a lot of different choices [bib-database-consistency] defined by different isolation levels that data stores provide.

It is important to note that the locks are long lived and not released when the request completes. They are released only when the transaction commits. This technique of holding locks for the duration of the transaction and releasing them only when the transaction commits or rolls back is called two-phase-locking [bib-two-phase-locking]. Two-phase locking is critical in providing the serializable isolation level. Serializable meaning that the effects of the transactions are visible as if they are executed one at a time.

Deadlock Prevention

Usage of locks can cause deadlocks where two transactions wait for each other to release the locks. Deadlocks can be avoided if transactions are not allowed to wait and aborted when the conflicts are detected. There are different strategies used to decide which transactions are aborted and which are allowed to continue.

The lock manager implements these wait policies as follows:

class LockManager…

```
WaitPolicy waitPolicy;
```

The WaitPolicy decides what to do when there are conflicting requests.

```
public enum WaitPolicy {
  WoundWait,
  WaitDie,
  Error
}
```

The lock is an object which tracks the transactions which currently own the lock and the ones which are waiting for the lock.

class Lock…

```
Queue<LockRequest> waitQueue = new LinkedList<>();
List<TransactionRef> owners = new ArrayList<>();
LockMode lockMode;
```

When a transaction requests to acquire a lock, the lock manager grants the lock immediately if there are no conflicting transactions already owning the lock.

class LockManager…

```
public synchronized CompletableFuture<TransactionRef> acquire(Trans
  return acquire(txn, key, lockMode, new CompletableFuture<>());
}

CompletableFuture<TransactionRef> acquire(TransactionRef txnRef,
                        String key,
                        LockMode askedLockMode,
                        CompletableFuture<TransactionRef> lockFutur
  Lock lock = getOrCreateLock(key);
logger.debug("acquiring lock for = " + txnRef + " on key = " + key
if (lock.isCompatible(txnRef, askedLockMode)) {
    lock.addOwner(txnRef, askedLockMode);
    lockFuture.complete(txnRef);
    logger.debug("acquired lock for = " + txnRef);
    return lockFuture;
}
if (lock.isLockedBy(txnRef) && lock.lockMode == askedLockMode) {
   lockFuture.complete(txnRef);
   logger.debug("Lock already acquired lock for = " + txnRef);
   return lockFuture;
}
```

class Lock…

```
public boolean isCompatible(TransactionRef txnRef, LockMode lockMod
  if(hasOwner()) {
    return (inReadMode() && lockMode == LockMode.READ)
         || isOnlyOwner(txnRef);
  }
  return true;
}
```

If there are conflicts, the lock manager acts depending on the wait policy.

Error On Conflict

If the wait policy is to error out, it will throw an error and the calling transaction will rollback and retry after a random timeout.

class LockManager…

```
private CompletableFuture<TransactionRef> handleConflict(Lock lock,
                                TransactionRef txnRef,
                                String key,
                                LockMode askedLockMode,
                                CompletableFuture<TransactionRef> l
  switch (waitPolicy) {
    case Error: {
      lockFuture.completeExceptionally(new WriteConflictException(t
      return lockFuture;
    }
    case WoundWait: {
      return lock.woundWait(txnRef, key, askedLockMode, lockFuture,
    }
    case WaitDie: {
      return lock.waitDie(txnRef, key, askedLockMode, lockFuture, t
    }
  }
  throw new IllegalArgumentException("Unknown waitPolicy " + waitPo
}
```

In case of contention when there are a lot of user transactions trying to acquire locks, if all of them need to restart, it severely limits the systems throughput. Data stores try to make sure that there are minimal transaction restarts.

A common technique is to assign a unique ID to transactions and order them. For example, Spanner [bib-spanner] assigns unique IDs [bib-spanner-concurrency] to transactions in such a way that they can be ordered. The technique is very similar to the one discussed in *Paxos* to order requests across cluster nodes. Once the transactions can be ordered, there are two techniques used to avoid deadlock, but still allow transactions to continue without restarting

The transaction reference is created in such a way that it can be compared and ordered with other transaction references. The easiest method is to assign a timestamp to each transaction and compare based on the timestamp.

class TransactionRef...

```
boolean after(TransactionRef otherTransactionRef) {
  return this.startTimestamp > otherTransactionRef.startTimestamp;
}
```

But in distributed systems, wall clocks are not monotonic [time-bound-lease.xhtml#wall-clock-not-monotonic], so a different method like assigning unique IDs to transactions in such a way that they can be ordered is used. Along with ordered IDs, the age of each is tracked to be able to order the transactions. Spanner [bib-spanner] orders transactions by tracking the age of each transaction in the system.

To be able to order all the transactions, each cluster node is assigned a unique ID. The client picks up the coordinator at the start of the transaction and gets the transaction ID from the coordinator The cluster node acting as a coordinator generates transaction IDs as follows.

class TransactionCoordinator…

```
private int requestId;
public MonotonicId begin() {
  return new MonotonicId(requestId++, config.getServerId());
}
```

class MonotonicId…

```
public class MonotonicId implements Comparable<MonotonicId> {
  public int requestId;
  int serverId;

  public MonotonicId(int requestId, int serverId) {
    this.serverId = serverId;
    this.requestId = requestId;
  }

  public static MonotonicId empty() {
    return new MonotonicId(-1, -1);
  }
  public boolean isAfter(MonotonicId other) {
    if (this.requestId == other.requestId) {
       return this.serverId > other.serverId;
    }
    return this.requestId > other.requestId;
}
```

class TransactionClient…

```
private void beginTransaction(String key) {
  if (coordinator == null) {
     coordinator = replicaMapper.serverFor(key);
     MonotonicId transactionId = coordinator.begin();
     transactionRef = new TransactionRef(transactionId, clock.nanoT
  }
}
```

The client tracks the age of the transaction by recording the elapsed time since the beginning of the transaction.

class TransactionRef…

```
public void incrementAge(SystemClock clock) {
  age = clock.nanoTime() - startTimestamp;
}
```

The client increments the age, every time a get or a put request is sent to the servers. The transactions are then ordered as per their age. The transaction id is used to break the ties when there are same age transactions.

class TransactionRef…

```
public boolean isAfter(TransactionRef other) {
  return age == other.age?
         this.id.isAfter(other.id)
         :this.age > other.age;
}
```

Wound-Wait

In the wound-wait [bib-wound-wait] method, if there is a conflict, the transaction reference asking for the lock is compared to all the transactions currently owning the lock. If the lock owners are all younger than the transaction asking for the lock, all of those transactions are aborted. But if the transaction asking the lock is younger than the ones owning the transaction, it waits for the lock

class Lock…

```
public CompletableFuture<TransactionRef> woundWait(TransactionRef t
                                    String key,
                                    LockMode askedLockMode,
                                    CompletableFuture<TransactionRef> lockF
                                    LockManager lockManager) {

  if (allOwningTransactionsStartedAfter(txnRef) && !anyOwnerIsPrepa
      abortAllOwners(lockManager, key, txnRef);
      return lockManager.acquire(txnRef, key, askedLockMode, lockFut
  }

  LockRequest lockRequest = new LockRequest(txnRef, key, askedLockM
  lockManager.logger.debug("Adding to wait queue = " + lockRequest)
  addToWaitQueue(lockRequest);
  return lockFuture;
}
```

class Lock…

```
private boolean allOwningTransactionsStartedAfter(TransactionRef tx
  return owners.stream().filter(o -> !o.equals(txn)).allMatch(owner
}
```

One of the key things to notice is that if the transaction owning the lock is already in the prepared state of two-phase-commit, it is not aborted.

Wait-Die

The wait-die [bib-wait-die] method works in the opposite way to wound-wait [bib-wound-wait]. If the lock owners are all younger than the transaction asking for the lock, then the transaction waits for the lock. But if the transaction asking for the lock is younger than the ones owning the transaction, the transaction is aborted.

class Lock…

```
public CompletableFuture<TransactionRef> waitDie(TransactionRef txn
                              String key,
                              LockMode askedLockMode,
                              CompletableFuture<TransactionRef> lockF
                              LockManager lockManager) {
  if (allOwningTransactionsStartedAfter(txnRef)) {
     addToWaitQueue(new LockRequest(txnRef, key, askedLockMode, loc
     return lockFuture;
  }

  lockManager.abort(txnRef, key);
  lockFuture.completeExceptionally(new WriteConflictException(txnRe
  return lockFuture;
}
```

Wound-wait mechanism generally has fewer restarts [bib-comparing-wait-die-and-wound-wait] compared to the wait-die method. So data stores like Spanner [bib-spanner] use the wound-wait [bib-wound-wait] method.

When the owner of the transaction releases a lock, the waiting transactions are granted the lock.

class LockManager…

```
private void release(TransactionRef txn, String key) {
  Optional<Lock> lock = getLock(key);
  lock.ifPresent(l -> {
    l.release(txn, this);
  });
}
```

class Lock…

```
public void release(TransactionRef txn, LockManager lockManager) {
  removeOwner(txn);
  if (hasWaiters()) {
     LockRequest lockRequest = getFirst(lockManager.waitPolicy);
     lockManager.acquire(lockRequest.txn, lockRequest.key, lockRequ
  }
}
```

Commit and Rollback

Once the client successfully reads without facing any conflicts and writes all the key values, it initiates the commit request by sending a commit request to the coordinator.

class TransactionClient…

```
public CompletableFuture<Boolean> commit() {
  return coordinator.commit(transactionRef);
}
```

The transaction coordinator records the state of the transaction as preparing to commit. The coordinator implements the commit handling in two phases.

- It first sends the prepare request to each of the participants.
- Once it receives a successful response from all the participants, the coordinator marks the transaction as prepared to complete. Then it sends the commit request to all the participants.

class TransactionCoordinator…

```
public CompletableFuture<Boolean> commit(TransactionRef transaction
  TransactionMetadata metadata = transactions.get(transactionRef);
  metadata.markPreparingToCommit(transactionLog);
  List<CompletableFuture<Boolean>> allPrepared = sendPrepareRequest
  CompletableFuture<List<Boolean>> futureList = sequence(allPrepare
  return futureList.thenApply(result -> {
    if (!result.stream().allMatch(r -> r)) {
       logger.info("Rolling back = " + transactionRef);
       rollback(transactionRef);
       return false;
    }
    metadata.markPrepared(transactionLog);
    sendCommitMessageToParticipants(transactionRef);
    metadata.markCommitComplete(transactionLog);
    return true;
  });
 }

 public List<CompletableFuture<Boolean>> sendPrepareRequestToPartic
   TransactionMetadata transactionMetadata = transactions.get(trans
   var transactionParticipants = getParticipants(transactionMetadat
   return transactionParticipants.keySet()
        .stream()
        .map(server -> server.handlePrepare(transactionRef))
        collect(Collectors.toList());
}

private void sendCommitMessageToParticipants(TransactionRef transac
  TransactionMetadata transactionMetadata = transactions.get(transa
  var participantsForKeys = getParticipants(transactionMetadata.get
  participantsForKeys.keySet().stream()
        .forEach(kvStore -> {
           List<String> keys = participantsForKeys.get(kvStore);
           kvStore.handleCommit(transactionRef, keys);
        });
}

private Map<TransactionalKVStore, List<String>> getParticipants(Lis
  return participatingKeys.stream()
       .map(k -> Pair.of(serverFor(k), k))
       .collect(Collectors.groupingBy(Pair::getKey, Collectors.mapp
}
```

The cluster node receiving the prepare requests do two things:

- It tries to grab the write locks for all of the keys.
- Once successful, it writes all of the changes to the write-ahead log.

If it can successfully do these, it can guarantee that there are no conflicting transactions, and even in the case of a crash the cluster node can recover all the required state to complete the transaction.

class TransactionalKVStore…

```
public synchronized CompletableFuture<Boolean> handlePrepare(Transa
  try {
     TransactionState state = getTransactionState(txn);
     if (state.isPrepared()) {
        return CompletableFuture.completedFuture(true); //already p
     }

     if (state.isAborted()) {
        return CompletableFuture.completedFuture(false); //aborted
     }

     Optional<Map<String, String>> pendingUpdates = state.getPendin
     CompletableFuture<Boolean> prepareFuture = prepareUpdates(txn,
     return prepareFuture.thenApply(ignored -> {
       Map<String, Lock> locksHeldByTxn = lockManager.getAllLocksFo
       state.markPrepared();
       writeToWAL(new TransactionMarker(txn, locksHeldByTxn, Transa
       return true;
     });

   } catch (TransactionException| WriteConflictException e) {
     logger.error(e);
   }
   return CompletableFuture.completedFuture(false);
```

```
}

private CompletableFuture<Boolean> prepareUpdates(TransactionRef tx
  if (pendingUpdates.isPresent()) {
     Map<String, String> pendingKVs = pendingUpdates.get();
     CompletableFuture<List<TransactionRef>> lockFuture = acquireLo
     return lockFuture.thenApply(ignored -> {
       writeToWAL(txn, pendingKVs);
       return true;
     });
   }
   return CompletableFuture.completedFuture(true);
}

TransactionState getTransactionState(TransactionRef txnRef) {
  return ongoingTransactions.get(txnRef);
}

private void writeToWAL(TransactionRef txn, Map<String, String> pen
  for (String key : pendingUpdates.keySet()) {
      String value = pendingUpdates.get(key);
      wal.writeEntry(new SetValueCommand(txn, key, value).serialize
  }
}

private CompletableFuture<List<TransactionRef>> acquireLocks(Transa
  List<CompletableFuture<TransactionRef>> lockFutures = new ArrayLi
  for (String key : keys) {
    CompletableFuture<TransactionRef> lockFuture = lockManager.acqu
    lockFutures.add(lockFuture);
  }
  return sequence(lockFutures);
}
```

When the cluster node receives the commit message from the coordinator, it is safe to make the key-value changes visible. The cluster node does three things while committing the changes:

- It marks the transaction as committed. Should the cluster node fail at this point, it knows the outcome of the transaction, and can repeat the following steps.
- It applies all the changes to the key-value storage
- It releases all the acquired locks.

class TransactionalKVStore…

```
public synchronized void handleCommit(TransactionRef transactionRef
  if (!ongoingTransactions.containsKey(transactionRef)) {
     return; //this is a no-op. Already committed.
  }

  if (!lockManager.hasLocksFor(transactionRef, keys)) {
     throw new IllegalStateException("Transaction " + transactionRe
  }

  writeToWAL(new TransactionMarker(transactionRef, TransactionStatu

  applyPendingUpdates(transactionRef);

  releaseLocks(transactionRef, keys);
}

private void removeTransactionState(TransactionRef txnRef) {
  ongoingTransactions.remove(txnRef);
}

private void applyPendingUpdates(TransactionRef txnRef) {
  TransactionState state = getTransactionState(txnRef);
  Optional<Map<String, String>> pendingUpdates = state.getPendingUp
  apply(txnRef, pendingUpdates);
}

private void apply(TransactionRef txnRef, Optional<Map<String, Stri
  if (pendingUpdates.isPresent()) {
     Map<String, String> pendingKv = pendingUpdates.get();
     apply(pendingKv);
  }
  removeTransactionState(txnRef);
}

private void apply(Map<String, String> pendingKv) {
  for (String key : pendingKv.keySet()) {
    String value = pendingKv.get(key);
    kv.put(key, value);
  }
}
private void releaseLocks(TransactionRef txn, List<String> keys) {
     lockManager.release(txn, keys);
}

private Long writeToWAL(TransactionMarker transactionMarker) {
  return wal.writeEntry(transactionMarker.serialize());
}
```

The rollback is implemented in a similar way. If there is any failure, the client communicates with the coordinator to rollback the transaction.

class TransactionClient…

```
public void rollback() {
  coordinator.rollback(transactionRef);
}
```

The transaction coordinator records the state of the transaction as preparing to rollback. Then it forwards the rollback request to all of the servers which stored the values for the given transaction. Once all of the requests are successful, the coordinator marks the transaction rollback as complete.In case the coordinator crashes after the transaction is marked as ‘prepared to rollback’, it can keep on sending the rollback messages to all the participating cluster nodes.

class TransactionCoordinator…

```
public void rollback(TransactionRef transactionRef) {
  transactions.get(transactionRef).markPrepareToRollback(this.trans

  sendRollbackMessageToParticipants(transactionRef);

  transactions.get(transactionRef).markRollbackComplete(this.transa
}

private void sendRollbackMessageToParticipants(TransactionRef trans
  TransactionMetadata transactionMetadata = transactions.get(transa
  var participants = getParticipants(transactionMetadata.getPartici
  for (TransactionalKVStore kvStore : participants.keySet()) {
     List<String> keys = participants.get(kvStore);
     kvStore.handleRollback(transactionMetadata.getTxn(), keys);
  }
}
```

The cluster nodes receiving the rollback request does three things:

- It records the state of the transaction as rolled back in the write-ahead log.
- It discards the transaction state.
- It releases all of the locks

class TransactionalKVStore…

```
public synchronized void handleRollback(TransactionRef transactionR
  if (!ongoingTransactions.containsKey(transactionRef)) {
     return; //no-op. Already rolled back.
  }
  writeToWAL(new TransactionMarker(transactionRef, TransactionStatu
  this.ongoingTransactions.remove(transactionRef);
  this.lockManager.release(transactionRef, keys);
}
```

Idempotent Operations

In case of network failures, the coordinator can retry calls to prepare, commit or abort. So these operations need to be idempotent [idempotentreceiver.xhtml#IdempotentAndNon-idempotentRequests].

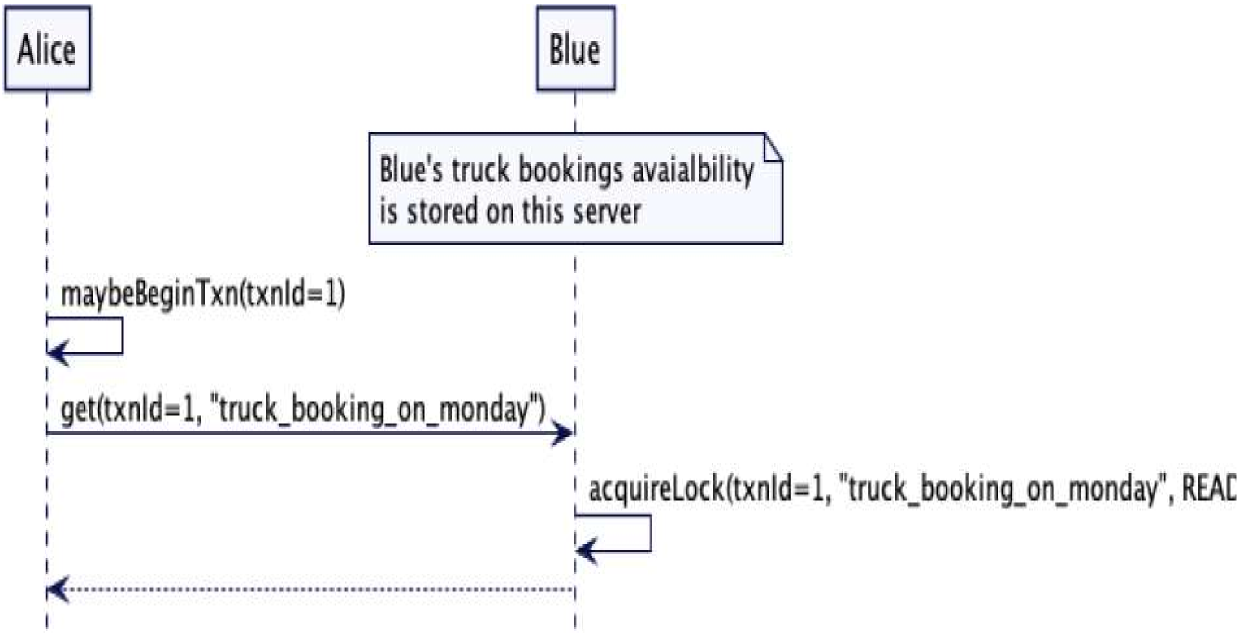
An Example Scenario

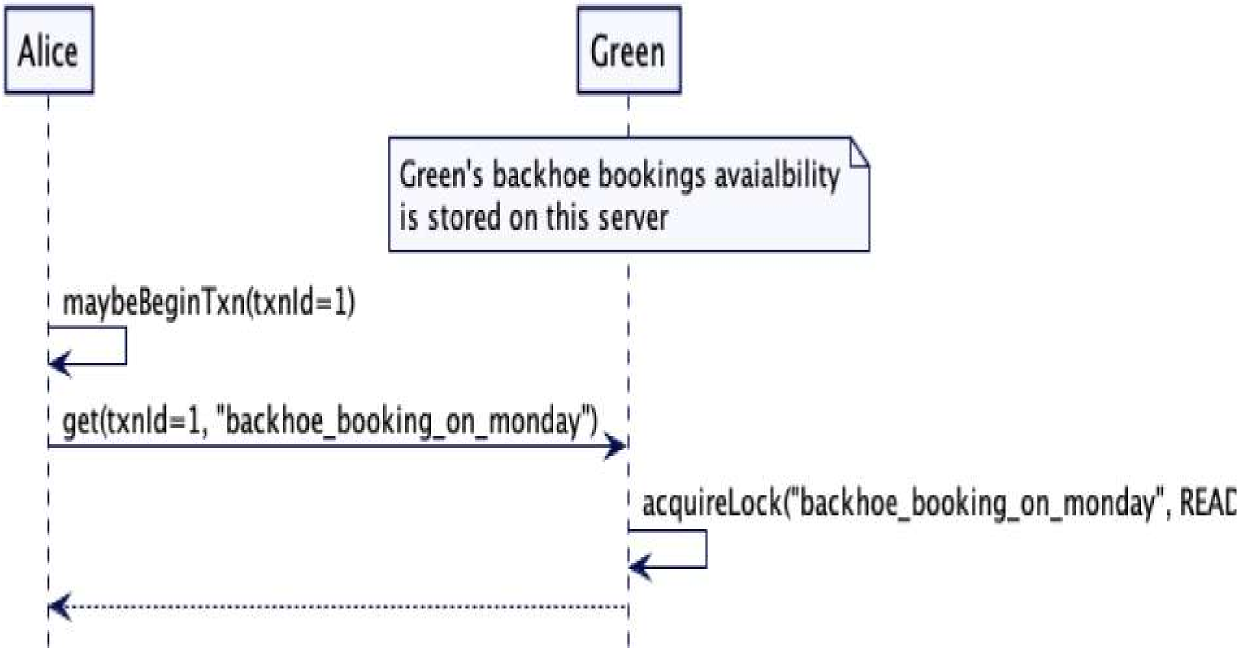
Atomic Writes

Consider the following scenario. Paula Blue has a truck and Steven Green has a backhoe. The availability and the booking status of the truck and the backhoe are stored on a distributed key-value store. Depending on how the keys are mapped to servers, Blue's truck and Green's backhoe bookings are stored on separate cluster nodes. Alice is trying to book a truck and backhoe for the construction work she is planning to start on a Monday. She needs both the truck and the backhoe to be available.

The booking scenario happens as follows.

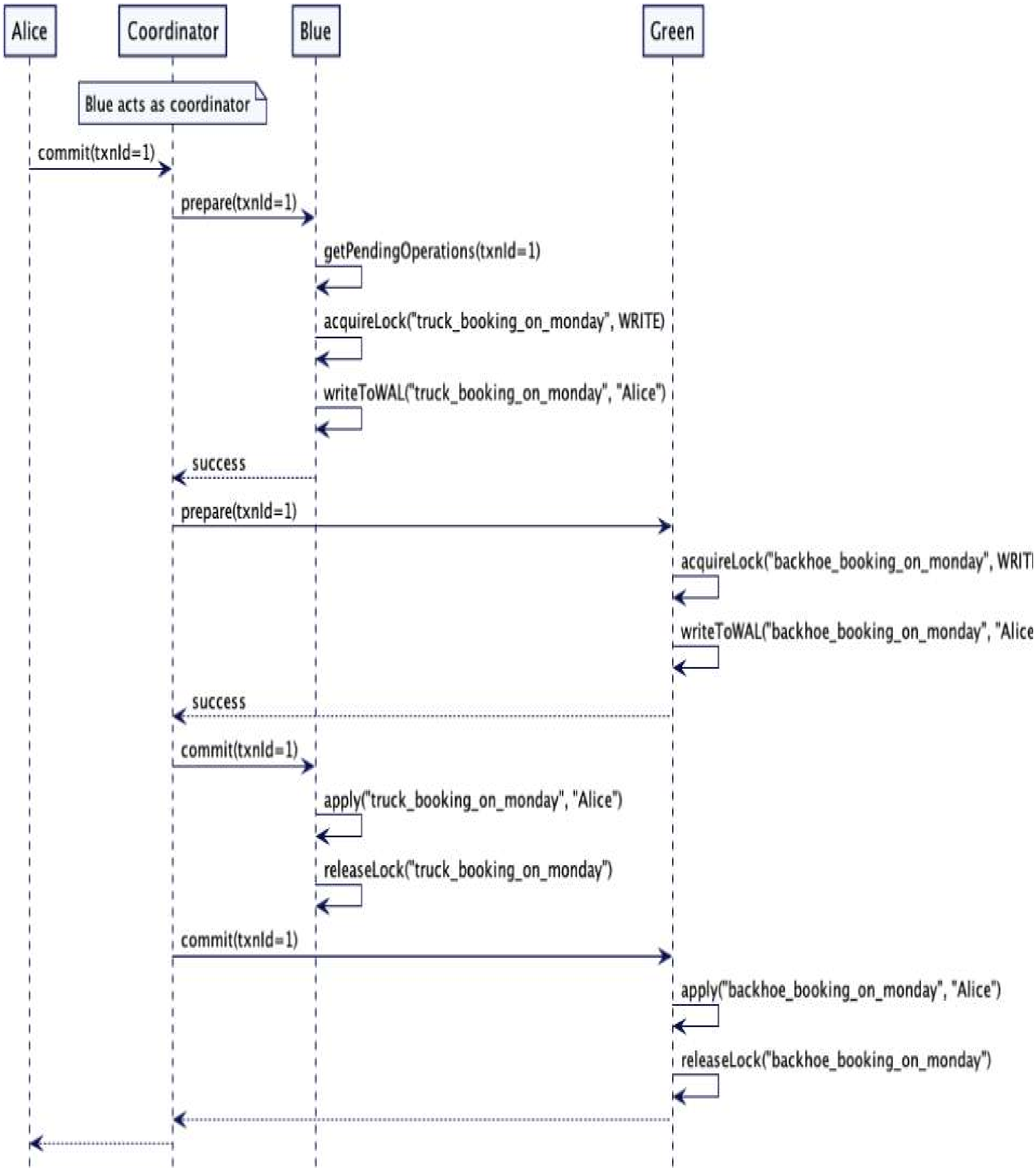
Alice checks the availability of Blue's truck and Green's backhoe. by reading the keys 'truck_booking_monday' and 'backhoe_booking_monday'


Alice
Blue
Blue's truck bookings avaialbility is stored on this server
maybeBeginTxn(txnId=1)
get(txnId=1, "truck_booking_on_monday")
acquireLock(txnId=1, "truck_booking_on_monday", REAC


Alice
Green
Green's backhoe bookings avaialbility is stored on this server
maybeBeginTxn(txnId=1)
get(txnId=1, "backhoe_booking_on_monday")
acquireLock("backhoe_booking_on_monday", REAC

If the values are empty, the booking is free. She reserves the truck and the backhoe. It is important that both the values are set atomically. If there is any failure, then none of the values is set.

The commit happens in two phases. The first server Alice contacts acts as the coordinator and executes the two phases.


Alice
Coordinator
Blue
Green
Blue acts as coordinator
commit(txnId=1)
prepare(txnId=1)
getPendingOperations(txnId=1)
acquireLock("truck_booking_on_monday", WRITE)
writeToWAL("truck_booking_on_monday", "Alice")
success
prepare(txnId=1)
acquireLock("backhoe_booking_on_monday", WRITI
writeToWAL("backhoe_booking_on_monday", "Alice
success
commit(txnId=1)
apply("truck_booking_on_monday", "Alice")
releaseLock("truck_booking_on_monday")
commit(txnId=1)
apply("backhoe_booking_on_monday", "Alice")
releaseLock("backhoe_booking_on_monday")

The coordinator is a separate participant in the protocol, and is shown that way on the sequence diagram. However usually one of the servers (Blue or Green) acts as the coordinator, thus playing two roles in the interaction.

Conflicting Transactions

Consider a scenario where another person, Bob, is also trying to book a truck and backhoe for construction work on the same Monday.

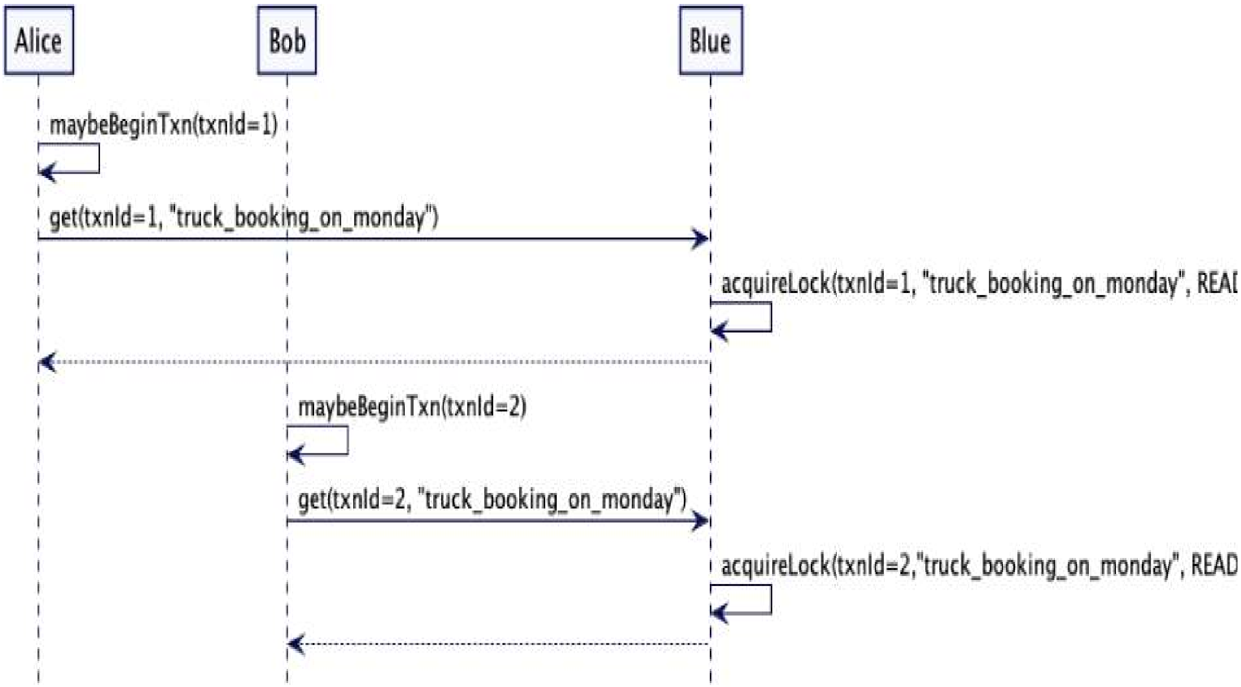
The booking scenario happens as follows:

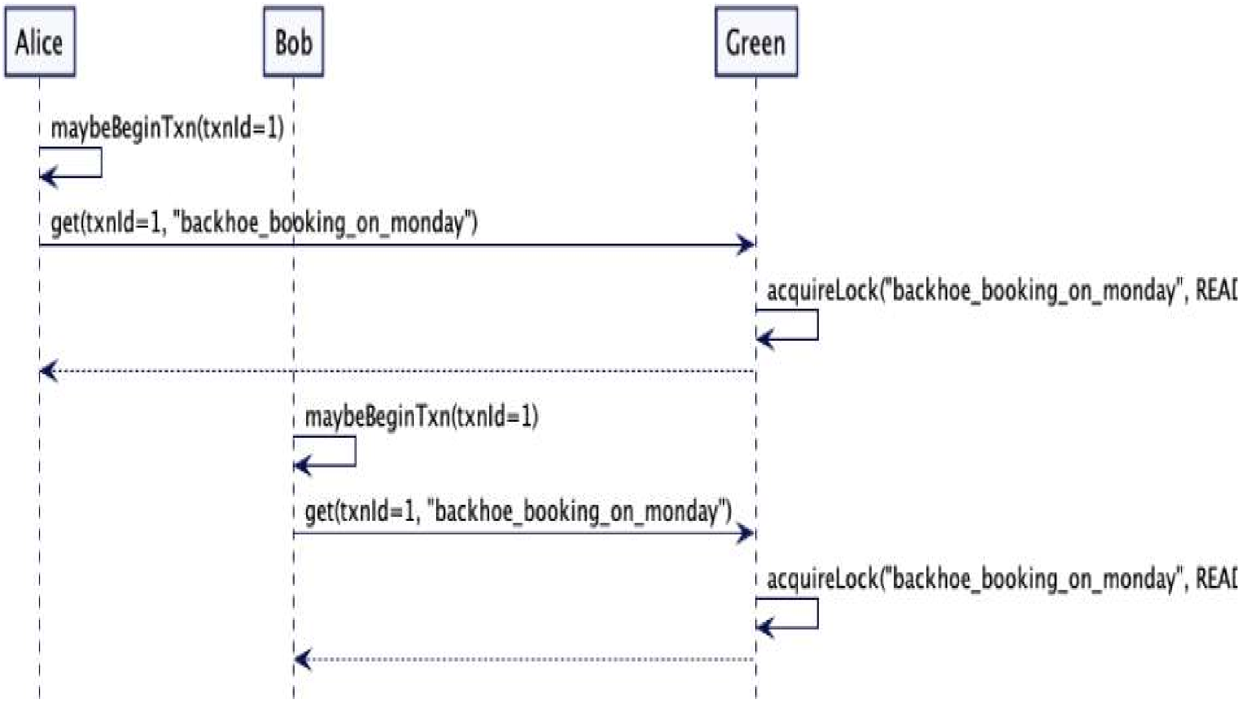
- Both Alice and Bob read the keys ‘truck_booking_monday’ and ‘backhoe_booking_monday’
- Both see that the values are empty, meaning the booking is free.
- Both try to book the truck and the backhoe.

The expectation is that, only Alice or Bob, should be able to book, because the transactions are conflicting. In case of errors, the whole flow needs to be retried and hopefully, one will go ahead with the booking. But in no situation, should booking be done partially. Either both bookings should be done or neither is done.

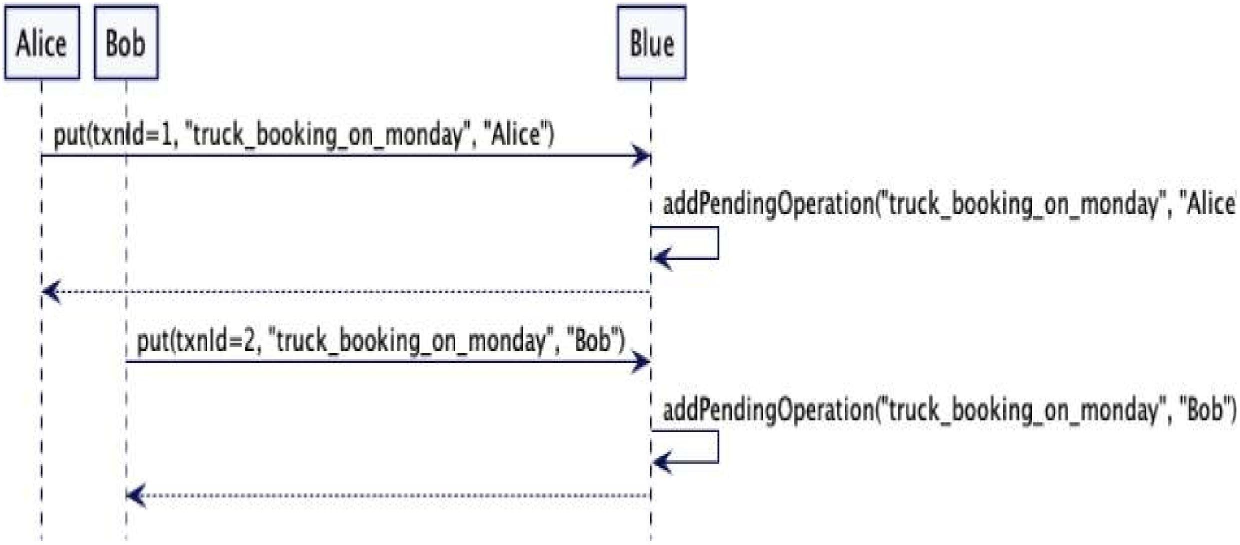
> The scenario gets into a deadlock situation because both the transactions depend on the locks held by others. The way out is for transactions to back-out and fail. The example implementation shown here will fail the transaction if it detects a conflicting transaction held holding a lock for a given key.

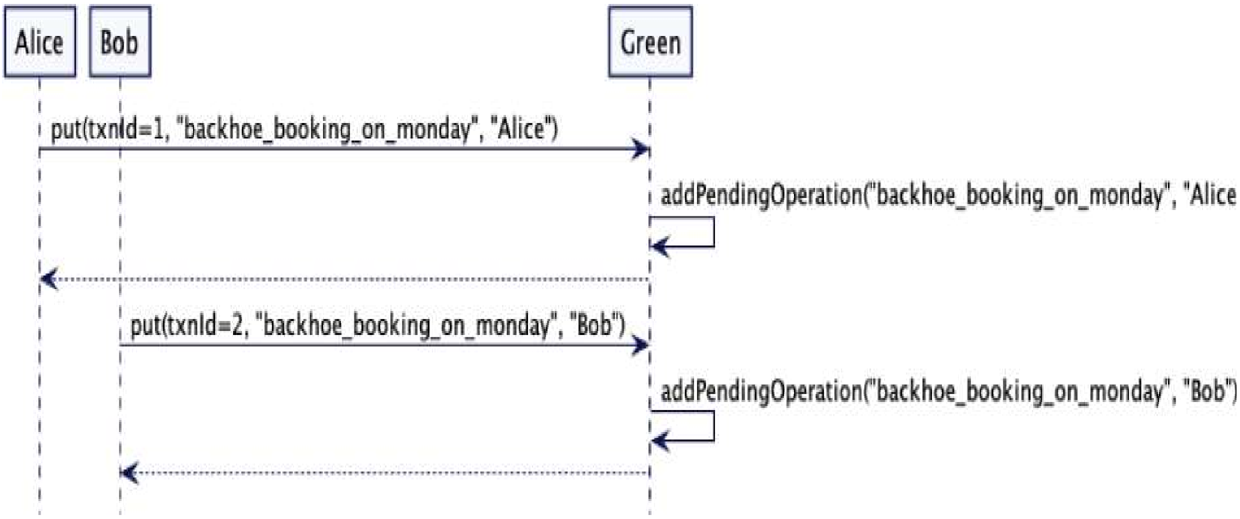
To check the availability, both Alice and Bob start a transaction and contact Blue and Green’s servers respectively to check for the availability. Blue holds a read lock for the key "truck_booking_on_monday" and Green holds a read lock for the key "backhoe_booking_on_monday". Because read locks are shared, both Alice and Bob can read the values.


Alice
Bob
Blue
maybeBeginTxn(txnId=1)
get(txnId=1, "truck_booking_on_monday")
acquireLock(txnId=1, "truck_booking_on_monday", REAI
maybeBeginTxn(txnId=2)
get(txnId=2, "truck_booking_on_monday")
acquireLock(txnId=2,"truck_booking_on_monday", READ


Alice
Bob
Green
maybeBeginTxn(txnId=1)
get(txnId=1, "backhoe_booking_on_monday")
acquireLock("backhoe_booking_on_monday", REAI
maybeBeginTxn(txnId=1)
get(txnId=1, "backhoe_booking_on_monday")
acquireLock("backhoe_booking_on_monday", REAI

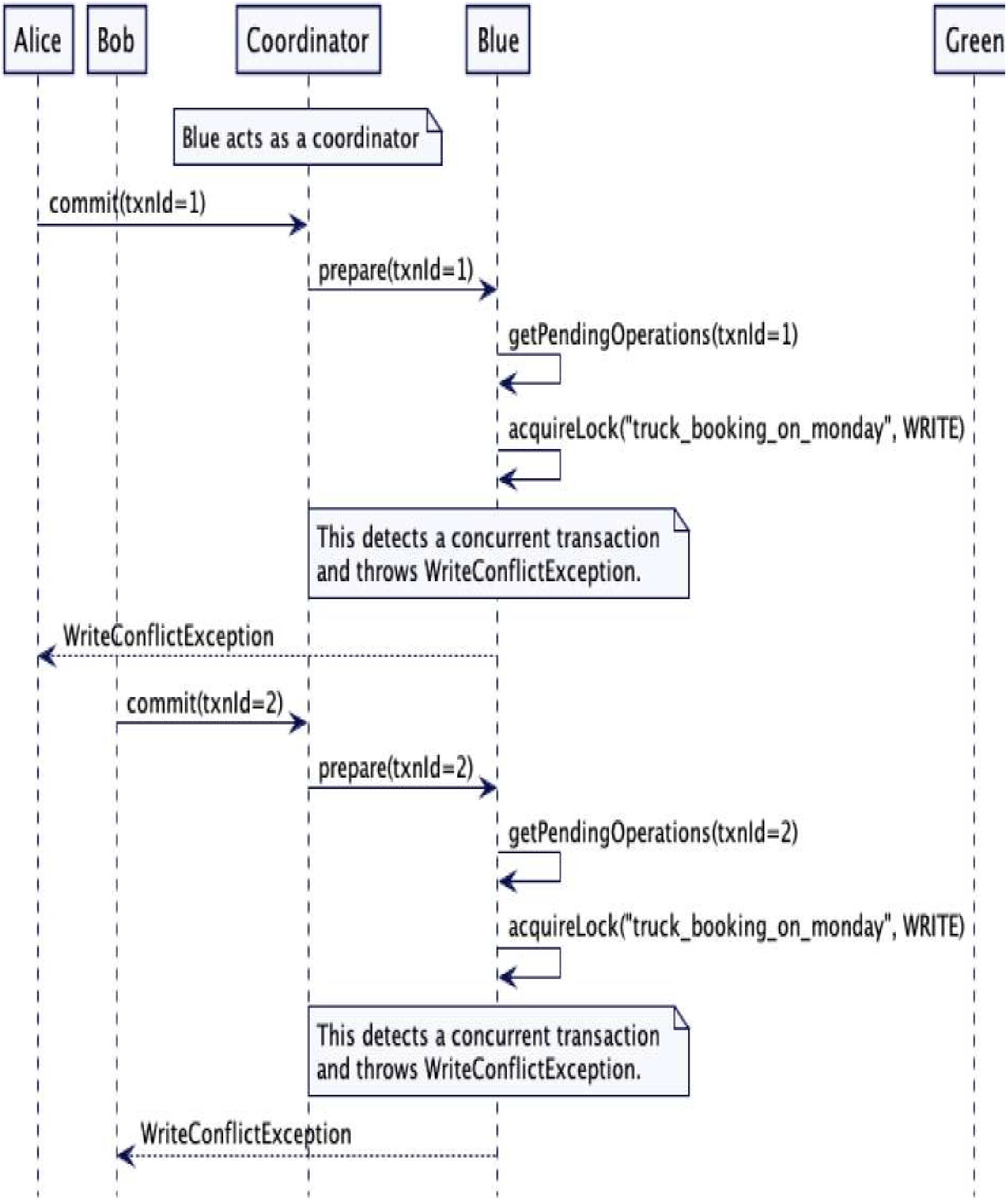
Alice and Bob see that both the bookings are available on Monday. So they reserve by sending the put requests to servers. Both the servers hold the put requests in the temporary storage.


Alice
Bob
Blue
put(txnId=1, "truck_booking_on_monday", "Alice")
addPendingOperation("truck_booking_on_monday", "Alice
put(txnId=2, "truck_booking_on_monday", "Bob")
addPendingOperation("truck_booking_on_monday", "Bob")


Alice
Bob
Green
put(txnId=1, "backhoe_booking_on_monday", "Alice")
addPendingOperation("backhoe_booking_on_monday", "Alice
put(txnId=2, "backhoe_booking_on_monday", "Bob")
addPendingOperation("backhoe_booking_on_monday", "Bob")

When Alice and Bob decide to commit the transactions- assuming that Blue acts as a coordinator- it triggers the two-phase commit protocol and sends the prepare requests to itself and Green.

For Alice’s request it tries to grab a write lock for the key ‘truck_booking_on_monday’, which it can not get, because there is a conflicting read lock grabbed by another transaction. So Alice’s transaction fails in the prepare phase. The same thing happens with Bob’s request.


Alice
Bob
Coordinator
Blue
Green
Blue acts as a coordinator
commit(txnId=1)
prepare(txnId=1)
getPendingOperations(txnId=1)
acquireLock("truck_booking_on_monday", WRITE)
This detects a concurrent transaction and throws WriteConflictException.
WriteConflictException
commit(txnId=2)
prepare(txnId=2)
getPendingOperations(txnId=2)
acquireLock("truck_booking_on_monday", WRITE)
This detects a concurrent transaction and throws WriteConflictException.
WriteConflictException

Transactions can be retried with a retry loop as follows:

class TransactionExecutor…

```
public boolean executeWithRetry(Function<TransactionClient, Boolean
  for (int attempt = 1; attempt <= maxRetries; attempt++) {
    TransactionClient client = new TransactionClient(replicaMapper,
    try {
       boolean checkPassed = txnMethod.apply(client);
       Boolean successfullyCommitted = client.commit().get();
       return checkPassed && successfullyCommitted;
    } catch (Exception e) {
      logger.error("Write conflict detected while executing." + cli
      client.rollback();
      randomWait(); //wait for random interval
    }
  }
  return false;
}
```

The example booking code for Alice and Bob will look as follows:

class TransactionalKVStoreTest…

```
@Test
public void retryWhenConflict() {
  List<TransactionalKVStore> allServers = createTestServers(WaitPol

  TransactionExecutor aliceTxn = bookTransactionally(allServers, "A
  TransactionExecutor bobTxn = bookTransactionally(allServers, "Bob

  TestUtils.waitUntilTrue(() -> (aliceTxn.isSuccess() && !bobTxn.is
}

  private TransactionExecutor bookTransactionally(List<Transactiona
    List<String> bookingKeys = Arrays.asList("truck_booking_on_mond
    TransactionExecutor t1 = new TransactionExecutor(allServers);
    t1.executeAsyncWithRetry(txnClient -> {
       if (txnClient.isAvailable(bookingKeys)) {
          txnClient.reserve(bookingKeys, user);
          return true;
       }
```

```
        return false;
    }, systemClock);
    return t1;
}
```

In this case one of the transactions will eventually succeed and the other will back out.

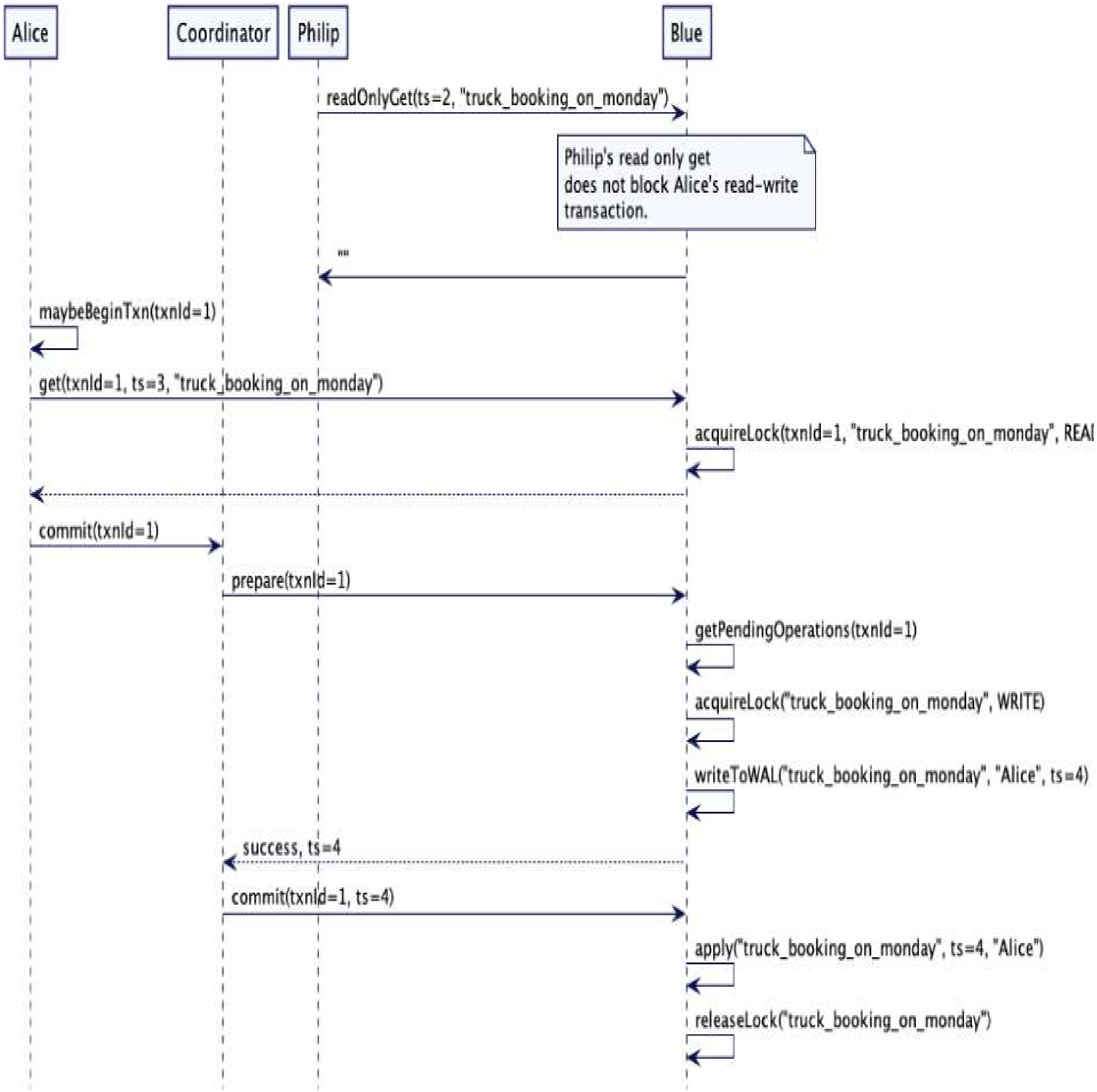
While it is very easy to implement, with Error WaitPolicy , there will be multiple transaction restarts,reducing the overall throughput. As explained in the above section, if Wound-Wait policy is used it will have fewer transaction restarts. In the above example, only one transaction will possibly restart instead of both restarting in case of conflicts.

Using *Versioned Value*

It is very constraining to have conflicts for all the read and write operations, particularly so when the transactions can be read-only. It is optimal if read-only transactions can work without holding any locks and still guarantee that the values read in a transaction do not change with a concurrent read-write transaction.

Data-stores generally store multiple versions of the values, as described in Versioned Value. The version used is the timestamp following *Lamport Clock*. Mostly a *Hybrid Clock* is used in databases like MongoDB [bib-mongodb] or CockroachDB [bib-cockroachdb]. To use it with the two-phase commit protocol, the trick is that every server participating in the transaction sends the timestamp it can write the values at, as response to the prepare request. The coordinator chooses the maximum of these timestamps as a commit timestamp and sends it along with the value. The participating servers then save the value at the commit timestamp. This allows read-only requests to be executed without holding locks, because it’s guaranteed that the value written at a particular timestamp is never going to change.

Consider a simple example as follows. Philip is running a report to read all of the bookings that happened until timestamp 2. If it is a long-running operation holding a lock, Alice, who is trying to book a truck, will be blocked until Philip’s work completes. With Versioned Value Philip’s get requests, which are part of a read-only operation, can continue at timestamp 2, while Alice's booking continues at timestamp 4.


Alice
Coordinator
Philip
Blue
readOnlyGet(ts=2, "truck_booking_on_monday")
Philip's read only get
does not block Alice's read-write
transaction.
""
maybeBeginTxn(txnId=1)
get(txnId=1, ts=3, "truck_booking_on_monday")
acquireLock(txnId=1, "truck_booking_on_monday", REAI
commit(txnId=1)
prepare(txnId=1)
getPendingOperations(txnId=1)
acquireLock("truck_booking_on_monday", WRITE)
writeToWAL("truck_booking_on_monday", "Alice", ts=4)
success, ts=4
commit(txnId=1, ts=4)
apply("truck_booking_on_monday", ts=4, "Alice")
releaseLock("truck_booking_on_monday")

Note that read requests which are part of a read-write transaction, still need to hold a lock.

The example code with Lamport Clock looks as follows:

class MvccTransactionalKVStore…

```
public String readOnlyGet(String key, long readTimestamp) {
  adjustServerTimestamp(readTimestamp);
  return kv.get(new VersionedKey(key, readTimestamp));
}

public CompletableFuture<String> get(TransactionRef txn, String key
  adjustServerTimestamp(readTimestamp);
  CompletableFuture<TransactionRef> lockFuture = lockManager.acquir
  return lockFuture.thenApply(transactionRef -> {
     getOrCreateTransactionState(transactionRef);
     return kv.get(key);
  });
}
private void adjustServerTimestamp(long readTimestamp) {
  this.timestamp = readTimestamp > this.timestamp ? readTimestamp:t
}

public void put(TransactionRef txnId, String key, String value) {
  timestamp = timestamp + 1;
  TransactionState transactionState = getOrCreateTransactionState(t
  transactionState.addPendingUpdates(key, value);
}
```

class MvccTransactionalKVStore…

```
private long prepare(TransactionRef txn, Optional<Map<String, Strin
  if (pendingUpdates.isPresent()) {
     Map<String, String> pendingKVs = pendingUpdates.get();

     acquireLocks(txn, pendingKVs);

     timestamp = timestamp + 1; //increment the timestamp for write

     writeToWAL(txn, pendingKVs, timestamp);
  }
  return timestamp;
}
```

class MvccTransactionCoordinator…

```
public long commit(TransactionRef txn) {
     long commitTimestamp = prepare(txn);

     TransactionMetadata transactionMetadata = transactions.get(txn
     transactionMetadata.markPreparedToCommit(commitTimestamp, this

     sendCommitMessageToAllTheServers(txn, commitTimestamp, transac

     transactionMetadata.markCommitComplete(transactionLog);

     return commitTimestamp;
}

public long prepare(TransactionRef txn) throws WriteConflictExcepti
  TransactionMetadata transactionMetadata = transactions.get(txn);
  Map<MvccTransactionalKVStore, List<String>> keysToServers = getPa
  List<Long> prepareTimestamps = new ArrayList<>();
  for (MvccTransactionalKVStore store : keysToServers.keySet()) {
    List<String> keys = keysToServers.get(store);
    long prepareTimestamp = store.prepare(txn, keys);
    prepareTimestamps.add(prepareTimestamp);
  }
  return prepareTimestamps.stream().max(Long::compare).orElse(txn.g
}
```

All the participating cluster nodes then store the key-values at the commit timestamp.

class MvccTransactionalKVStore…

```
public void commit(TransactionRef txn, List<String> keys, long comm
  if (!lockManager.hasLocksFor(txn, keys)) {
    throw new IllegalStateException("Transaction should hold all th
  }
  adjustServerTimestamp(commitTimestamp);
  applyPendingOperations(txn, commitTimestamp);

  lockManager.release(txn, keys);

  logTransactionMarker(new TransactionMarker(txn, TransactionStatus
}

private void applyPendingOperations(TransactionRef txnId, long comm
  Optional<TransactionState> transactionState = getTransactionState
  if (transactionState.isPresent()) {
     TransactionState t = transactionState.get();
     Optional<Map<String, String>> pendingUpdates = t.getPendingUpd
     apply(txnId, pendingUpdates, commitTimestamp);
  }
}

private void apply(TransactionRef txnId, Optional<Map<String, Strin
  if (pendingUpdates.isPresent()) {
     Map<String, String> pendingKv = pendingUpdates.get();
     apply(pendingKv, commitTimestamp);
  }
  ongoingTransactions.remove(txnId);
}


private void apply(Map<String, String> pendingKv, long commitTimest
  for (String key : pendingKv.keySet()) {
     String value = pendingKv.get(key);
     kv.put(new VersionedKey(key, commitTimestamp), value);
  }
}
```

Technical Considerations

There is another subtle issue to be tackled here. Once a particular response is returned at a given timestamp, no write should happen at a lower timestamp than the one received in the read request. This is achieved by different techniques. Google Percolator [bib-percolator] and datastores like TiKV [bib-tikv] inspired by Percolator use a separate server called Timestamp oracle which is guaranteed to give monotonic timestamps. Databases like MongoDB [bib-mongodb] or CockroachDB [bib-cockroachdb] use *Hybrid Clock* to guarantee it because every request will adjust the hybrid clock on each server to be the most up-todate. The timestamp is also advanced monotonically with every write request. Finally, the commit phase picks up the maximum timestamp across the set of participating servers, making sure that the write will always follow a previous read request.

It is important to note that, if the client is reading at a timestamp value lower than the one at which server is writing to, it is not an issue. But if the client is reading at a timestamp while the server is about to write at a particular timestamp, then it is a problem. If servers detect that a client is reading at a timestamp which the server might have an in-flight writes (the ones which are only prepared), the servers reject the write. CockroachDB [bib-cockroachdb] throws error an if a read happens at a timestamp for which there is an ongoing transaction. Spanner [bib-spanner] reads have a phase where the client gets the time of the last successful write on a particular partition. If a client reads at a higher timestamp, the read requests wait till the writes happen at that timestamp.

Using *Replicated Log*

To improve fault tolerance cluster nodes use Replicated Log. The coordinator uses Replicated Log to store the transaction log entries.

Considering the example of Alice and Bob in the above section, the Blue servers will be a group of servers, so are the Green servers. All the booking data will be replicated across a set of servers. Each request which is part of the two-phase commit goes to the leader of the server group. The replication is implemented using Replicated Log.

The client communicates with the leader of each server group. The replication is necessary only when the client decides to commit the transaction, so it happens as part of the prepare request.

The coordinator replicates every state change to replicated log as well.

> Multi-Raft is very different thing than Multi-Paxos [bib-multi-paxos]. Multi-Paxos refers to a single Replicated Log. multiple Paxos instances, with a Paxos instance per log entry. Multi-Raft refers to multiple Replicated Logs.

In a distributed datastore, each cluster node handles multiple partitions. A Replicated Log is maintained per partition. When Raft [bib-raft] is used as part of replication it's sometimes referred to as multi-raft [bib-multi-raft].

Client communicates with the leader of each partition participating in the transaction.

Failure Handling

Two-phase commit protocol heavily relies on the coordinator node to communicate the outcome of the transaction. Until the outcome of the transaction is known, the individual cluster nodes cannot allow any other transactions to write to the keys participating in the pending transaction. The cluster nodes block until the outcome of the transaction is known. This puts some critical requirements on the coordinator

The coordinator needs to remember the state of the transactions even in case of a process crash.

Coordinator uses *Write-Ahead Log* to record every update to the state of the transaction. This way, when the coordinator crashes and comes back up, it can continue to work on the transactions which are incomplete.

class TransactionCoordinator…

```
public void loadTransactionsFromWAL() throws IOException {
  List<WALEntry> walEntries = this.transactionLog.readAll();
  for (WALEntry walEntry : walEntries) {
     TransactionMetadata txnMetadata = (TransactionMetadata) Comman
     transactions.put(txnMetadata.getTxn(), txnMetadata);
  }
  startTransactionTimeoutScheduler();
  completePreparedTransactions();
```

```
}
private void completePreparedTransactions() throws IOException {
  List<Map.Entry<TransactionRef, TransactionMetadata>> preparedTran
     = transactions.entrySet().stream().filter(entry -> entry.getVa
  for (Map.Entry<TransactionRef, TransactionMetadata> preparedTrans
     TransactionMetadata txnMetadata = preparedTransaction.getValue
     sendCommitMessageToParticipants(txnMetadata.getTxn());
  }
}
```

The client can fail before sending the commit message to the coordinator.

The transaction coordinator tracks when each transaction state was updated. If no state update is received in a timeout period, which is configured, it triggers a transaction rollback.

class TransactionCoordinator…

```
private ScheduledThreadPoolExecutor scheduler = new ScheduledThread
private ScheduledFuture<?> taskFuture;
private long transactionTimeoutMs = Long.MAX_VALUE; //for now.

public void startTransactionTimeoutScheduler() {
  taskFuture = scheduler.scheduleAtFixedRate(() -> timeoutTransacti
       transactionTimeoutMs,
       transactionTimeoutMs,
       TimeUnit.MILLISECONDS);
}

private void timeoutTransactions() {
  for (TransactionRef txnRef : transactions.keySet()) {
     TransactionMetadata transactionMetadata = transactions.get(txn
     long now = systemClock.nanoTime();
     if (transactionMetadata.hasTimedOut(now)) {
     sendRollbackMessageToParticipants(transactionMetadata.getTxn()
     transactionMetadata.markRollbackComplete(transactionLog);
   }
```

```
    }
}
```

Transactions across heterogenous systems

The solution outlined here demonstrates the two-phase commit implementation in a homogenous system. Homogenous meaning all the cluster nodes are part of the same system and store same kind of data. For example a distributed data store like MongoDb or a distributed message broker like Kafka.

Historically, two-phase commit was mostly discussed in the context of heterogeneous systems. Most common usage of two-phase commit was with [XA] [bib-XA] transactions. In the J2EE servers, it is very common to use two-phase commit across a message broker and a database. The most common usage pattern is when a message needs to be produced on a message broker like ActiveMQ or JMS and a record needs to be inserted/updated in a database.

As seen in the above sections, the fault tolerance of the coordinator plays a critical role in two-phase commit implementation. In case of XA transactions the coordinator is mostly the application process making the database and message broker calls. The application in most modern scenarios is a stateless microservice which is running in a containerized environment. It is not really a suitable place to put the responsibility of the coordinator. The coordinator needs to maintain state and recover quickly from failures to commit or rollback, which is difficult to implement in this case.

This is the reason that while XA transactions seem so attractive, they often run into issues in practice [bib-activemq-slow-restart] and are avoided. In the microservices world, patterns like [transactional-outbox] [bib-transactional-outbox] are preferred over XA transactions.

On the other hand most distributed storage systems implement two-phase commit across a set of partitions, and it works well in practice.

Examples

Distributed databases like CockroachDB [bib-cockroachdb], MongoDB [bib-mongodb] etc. implement two phase commit to atomically storing values across partitions

Kafka [bib-kafka] allows producing messages across multiple partitions atomically with the implementation similar to two phase commit.

Part IV: Patterns of Distributed Time

Chapter 22. Lamport Clock

Use logical timestamps as a version for a value to allow ordering of values across servers

Problem

When values are stored across multiple servers, there needs to be a way to know which values were stored before the other. The system timestamp can not be used, because wall clocks are not monotonic [time-bound-lease.xhtml#wall-clock-not-monotonic] and clock values from two different servers should not be compared.

The system timestamp, which represents the time of the day, is measured by a clock machinery generally built with an crystal oscillator. The known problem with this mechanism is that it can drift away from the actual time of the day, based on how fast or slow the crystals oscillate. To fix this, computers typically have a service like NTP [bib-ntp] which synchronizes computer clocks with well known time sources on the internet. Because of this, two consecutive readings of the system time on a given server can have time going backwards.

As there is no upper bound on clock drift across servers, it is impossible to compare timestamps on two different servers

Solution

Lamport Clock maintains a single number to represent timestamp as following:

class LamportClock…

```
class LamportClock {
  int latestTime;

  public LamportClock(int timestamp) {
    latestTime = timestamp;
  }
```

Every cluster node maintains an instance of a Lamport Clock.

class Server…

```
MVCCStore mvccStore;
LamportClock clock;
public Server(MVCCStore mvccStore) {
  this.clock = new LamportClock(1);
  this.mvccStore = mvccStore;
}
```

Whenever a server carries out any write operation, it should advance the Lamport Clock, using the `tick()` method

class LamportClock…

```
public int tick(int requestTime) {
  latestTime = Integer.max(latestTime, requestTime);
  latestTime++;
  return latestTime;
}
```

This way, the server can be sure that the write is sequenced after the request and after any other action the server has carried out since the request was initiated by the client. The server returns the timestamp that was used for writing the value to the client. The requesting client then uses this timestamp to issue any further writes to other set of servers. This way, the causal chain of requests is maintained.

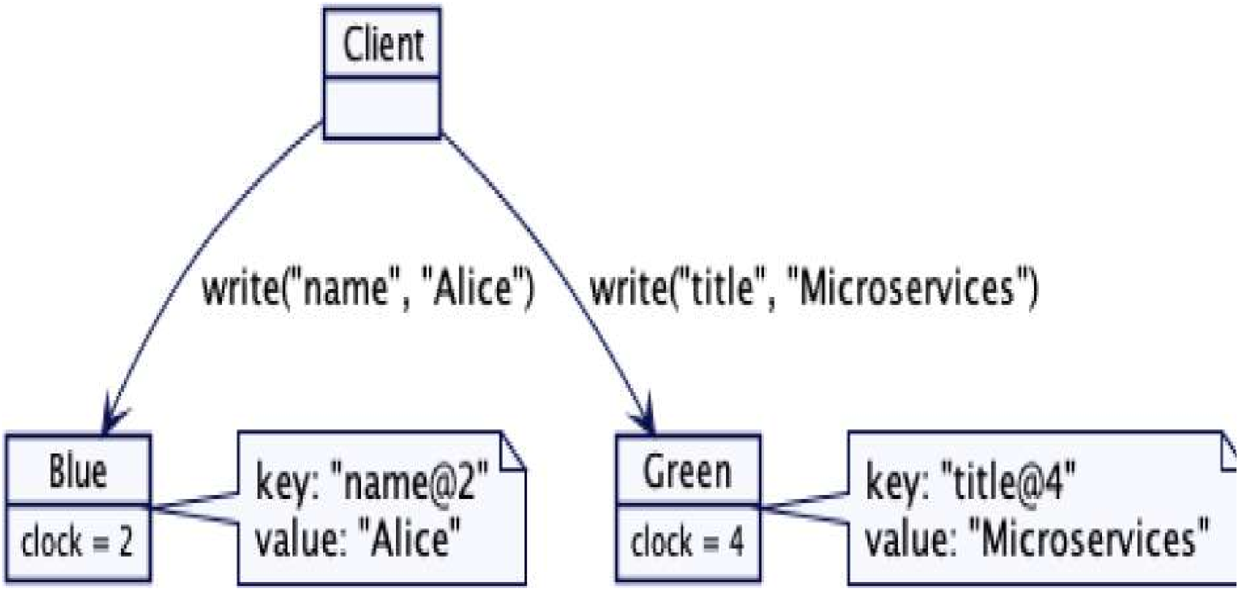
Causality, Time and Happens-Before

When an event A in a system happens before another event B, it might have a causal relationship. Causal relationship means that A might have some role in causing B. This ‘A happens before B’ relationship is established by attaching a timestamp to each event. If A happens before B, the timestamp attached to A will be lower than the timestamp attached to B. But because we can not rely on system time, we need some way to make sure that the happens-before relationship is maintained for the timestamp attached to the events. Leslie Lamport [bib-laslie-lamport] suggested a solution to use logical timestamps to track happens-before relationships, in his seminal paper Time, Clocks and Ordering Of Events [bib-time-clocks-ordering]. So this technique of using logical timestamps to track causality is named as the Lamport Timestamp.

It is useful to note that in a database, events are about storing data. So Lamport Timestamps are attached to the values stored. This also fits very well with versioned storage mechanism discussed in *Versioned Value*

An example key value store

Consider an example of a simple key value store with multiple server nodes. There are two servers, Blue and Green. Each server is responsible for storing a specific set of keys. This is a typical scenario when data is partitioned across a set of servers. Values are stored as *Versioned Value* with the Lamport Timestamp as a version number.



The receiving server compares and updates its own timestamp and uses it to write a versioned key value.

class Server…

```
public int write(String key, String value, int requestTimestamp) {
  //update own clock to reflect causality
  int writeAtTimestamp = clock.tick(requestTimestamp);
  mvccStore.put(new VersionedKey(key, writeAtTimestamp), value);
  return writeAtTimestamp;
}
```

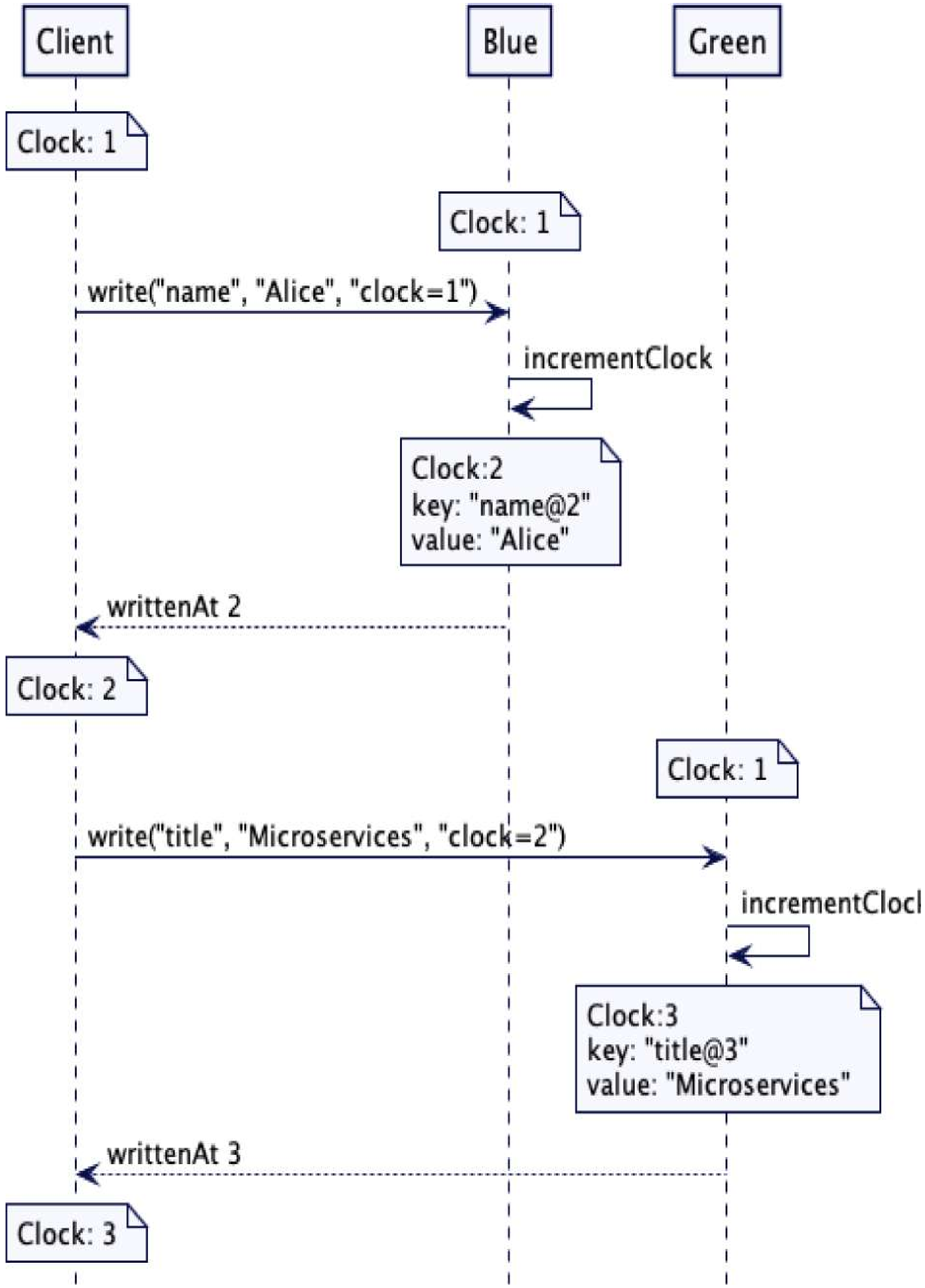
The timestamp used for writing the value is returned to the client. The client keeps track of the maximum timestamp, by updating its own timestamp. It uses this timestamp to issue any further writes.

class Client…

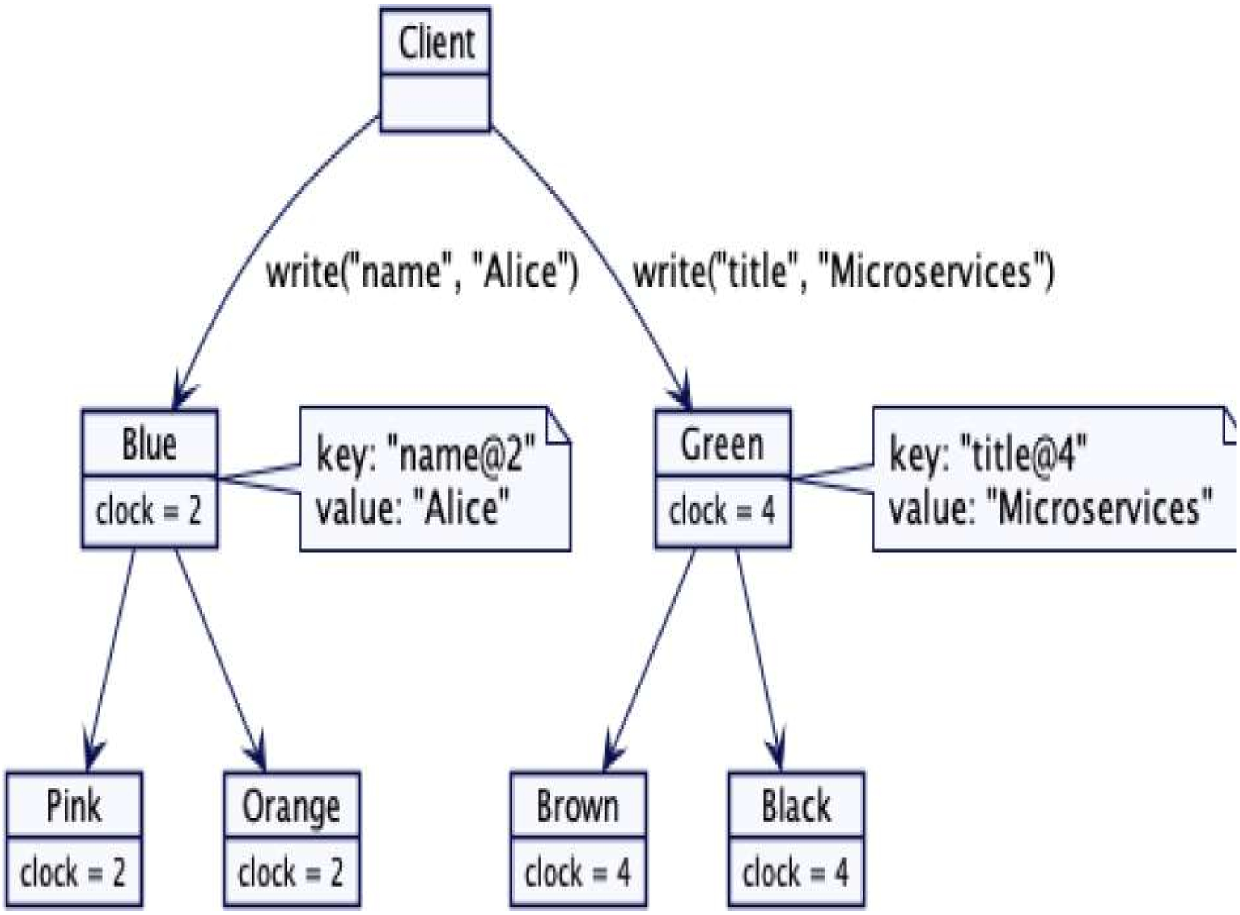
```
LamportClock clock = new LamportClock(1);
public void write() {
  int server1WrittenAt = server1.write("name", "Alice", clock.getLa
  clock.updateTo(server1WrittenAt);

  int server2WrittenAt = server2.write("title", "Microservices", cl
  clock.updateTo(server2WrittenAt);
```

```
  assertTrue(server2WrittenAt > server1WrittenAt);
}
```

The sequence of requests look like following:

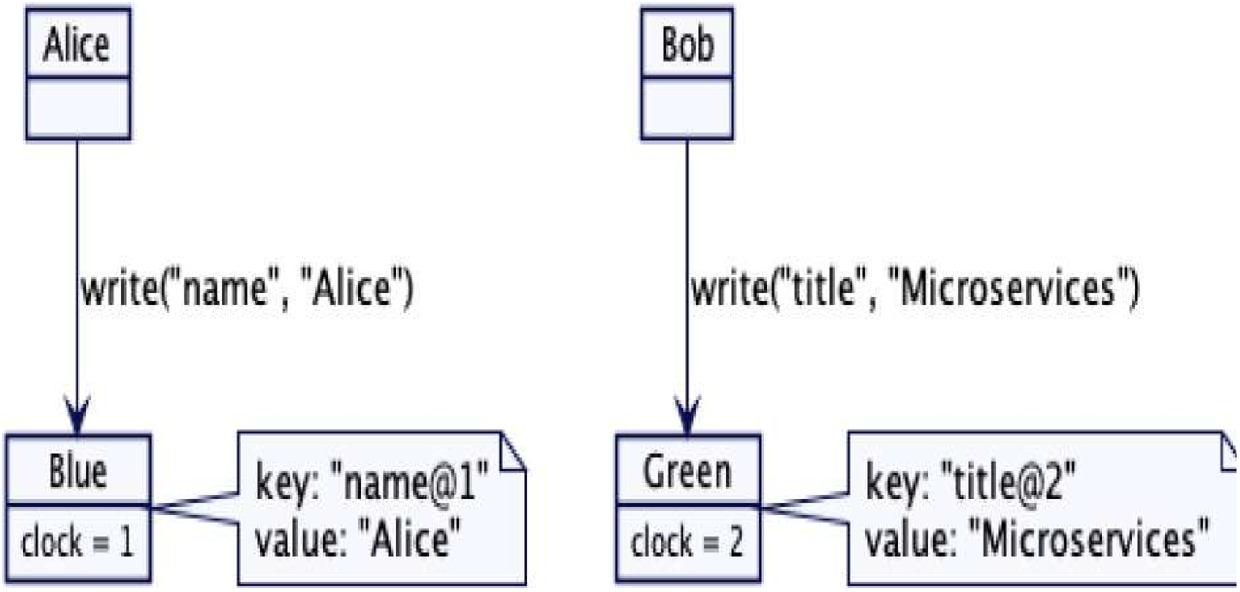

Client
Blue
Green
Clock: 1
Clock: 1
write("name", "Alice", "clock=1")
incrementClock
Clock:2
key: "name@2"
value: "Alice"
writtenAt 2
Clock: 2
Clock: 1
write("title", "Microservices", "clock=2")
incrementClocl
Clock:3
key: "title@3"
value: "Microservices"
writtenAt 3
Clock: 3

The same technique works even when the client is communicating with a leader with *Leader and Followers* groups, with each group responsible for specific keys. The client sends requests to the leader of the group as detailed above. The Lamport Clock instance is maintained by the leader of the group, and is updated exactly the same way as discussed in the previous section.



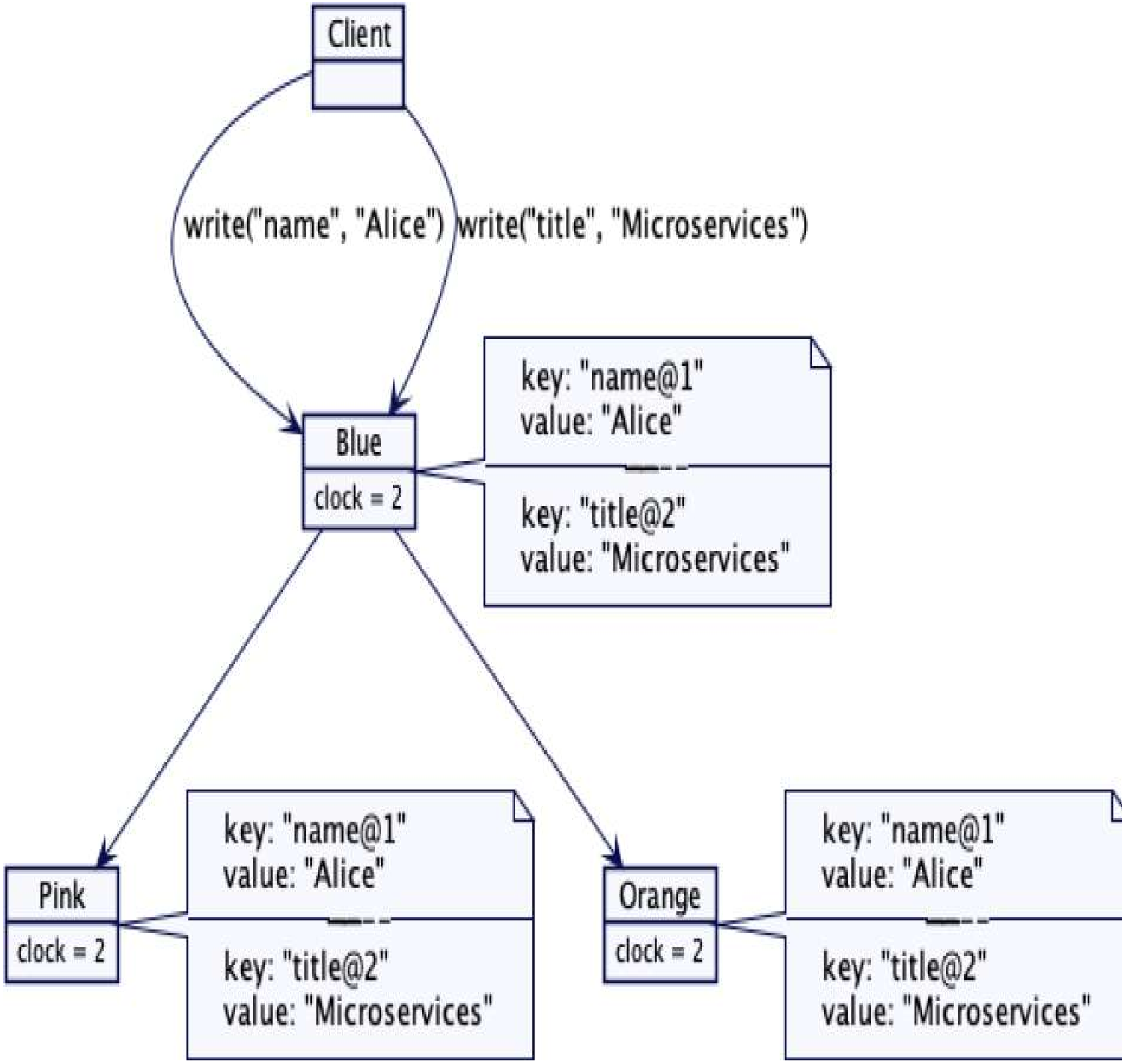
Partial Order

The values stored by Lamport Clock are only partially ordered [bib-partial-order]. If two clients store values in two separate servers, the timestamp values cannot be used to order the values across servers. In the following example, the title stored by Bob on server Green is at timestamp 2. But it can not be determined if Bob stored the title before or after Alice stored the name on server Blue.



A single server/leader updating values

For a single leader-follower group of servers, where a leader is always responsible for storing values, the basic implementation discussed in *Versioned Value* is enough to maintain causality.



In this case, the key value store keeps an integer version counter. It increments the version counter every time the key value write command is applied from the Write Ahead Log. It then constructs the new key with the incremented version counter. Only the leader is responsible for incrementing the version counter, and followers use the same version number.

class ReplicatedKVStore…

```
int version = 0;
MVCCStore mvccStore = new MVCCStore();
```

```
@Override
public CompletableFuture<Response> put(String key, String value) {
  return replicatedLog.propose(new SetValueCommand(key, value));
}

private Response applySetValueCommand(SetValueCommand setValueComma
  getLogger().info("Setting key value " + setValueCommand);
  version = version + 1;
  mvccStore.put(new VersionedKey(setValueCommand.getKey(), version)
  Response response = Response.success(version);
  return response;
}
```

Examples

Databases like MongoDB [bib-mongodb] and CockroachDB [bib-cockroachdb] use variants of the Lamport Clock to implement [mvcc] [bib-mvcc] storage

Generation Clock is an example of a Lamport Clock

Chapter 23. Hybrid Clock

Use a combination of system timestamp and logical timestamp to have versions as date-time, which can be ordered

Problem

When *Lamport Clock* is used as a version in *Versioned Value*, clients do not know the actual date-time when the particular versions are stored. It’s useful for clients to access versions using date-time like 01-01-2020 instead of using integers like 1, 2, 3.

Solution

Hybrid Logical Clock [bib-hybrid-clock] provides a way to have a version which is monotonically increasing just like a simple integer, but also has relation with the actual date time. Hybrid clocks are used in practice by databases like mongodb [bib-mongodb-hybridclock] or cockroachdb [bib-cockroachdb-hybridclock].

A Hybrid Logical Clock is implemented as follows:

class HybridClock…

```
public class HybridClock {
  private final SystemClock systemClock;
  private HybridTimestamp latestTime;
  public HybridClock(SystemClock systemClock) {
    this.systemClock = systemClock;
    this.latestTime = new HybridTimestamp(systemClock.now(), 0);
  }
```

It maintains the latest time as an instance of the hybrid timestamp, which is constructed by using system time and an integer counter.

class HybridTimestamp…

```
public class HybridTimestamp implements Comparable<HybridTimestamp>
  private final long wallClockTime;
  private final int ticks;

  public HybridTimestamp(long systemTime, int ticks) {
    this.wallClockTime = systemTime;
    this.ticks = ticks;
  }

  public static HybridTimestamp fromSystemTime(long systemTime) {
    return new HybridTimestamp(systemTime, -1); //initializing with
}

public HybridTimestamp max(HybridTimestamp other) {
  if (this.getWallClockTime() == other.getWallClockTime()) {
     return this.getTicks() > other.getTicks()? this:other;
  }
  return this.getWallClockTime() > other.getWallClockTime()?this:ot
}

public long getWallClockTime() {
  return wallClockTime;
}

public HybridTimestamp addTicks(int ticks) {
  return new HybridTimestamp(wallClockTime, this.ticks + ticks);
}

public int getTicks() {
  return ticks;
}

@Override
```

```
public int compareTo(HybridTimestamp other) {
  if (this.wallClockTime == other.wallClockTime) {
     return Integer.compare(this.ticks, other.ticks);
  }
  return Long.compare(this.wallClockTime, other.wallClockTime);
}
```

Hybrid clocks can be used exactly the same way as the Lamport Clock versions. Every server holds an instance of a hybrid clock.

class Server…

```
HybridClockMVCCStore mvccStore;
HybridClock clock;

public Server(HybridClockMVCCStore mvccStore) {
  this.clock = new HybridClock(new SystemClock());
  this.mvccStore = mvccStore;
}
```

Every time a value is written, a hybrid timestamp is associated with it. The trick is to check if the system time value is going back in time, if so increment another number representing a logical part of the component to reflect clock progress.

class HybridClock…

```
public synchronized HybridTimestamp now() {
  long currentTimeMillis = systemClock.now();
  if (latestTime.getWallClockTime() >= currentTimeMillis) {
      latestTime = latestTime.addTicks(1);
  } else {
     latestTime = new HybridTimestamp(currentTimeMillis, 0);
  }
  return latestTime;
}
```

Every write request that a server receives from the client carries a timestamp. The receiving server compares its own timestamp to that of the request and sets it’s own timestamp to the higher of the two

class Server…

```
public HybridTimestamp write(String key, String value, HybridTimest
  //update own clock to reflect causality
  HybridTimestamp writeAtTimestamp = clock.tick(requestTimestamp);
  mvccStore.put(key, writeAtTimestamp, value);
  return writeAtTimestamp;
}
```

class HybridClock…

```
public synchronized HybridTimestamp tick(HybridTimestamp requestTim
  long nowMillis = systemClock.now();
  //set ticks to -1, so that, if this is the max, the next addTicks
  HybridTimestamp now = HybridTimestamp.fromSystemTime(nowMillis);
  latestTime = max(now, requestTime, latestTime);
  latestTime = latestTime.addTicks(1);
  return latestTime;
}

private HybridTimestamp max(HybridTimestamp ...times) {
  HybridTimestamp maxTime = times[0];
  for (int i = 1; i < times.length; i++) {
    maxTime = maxTime.max(times[i]);
  }
  return maxTime;
}
```

The timestamp used for writing the value is returned to the client. The requesting client updates its own timestamp and then uses this timestamp to issue any further writes.

class Client…

```
HybridClock clock = new HybridClock(new SystemClock());
public void write() {
  HybridTimestamp server1WrittenAt = server1.write("name", "Alice",
  clock.tick(server1WrittenAt);

  HybridTimestamp server2WrittenAt = server2.write("title", "Micros

  assertTrue(server2WrittenAt.compareTo(server1WrittenAt) > 0);
}
```

Multiversion storage with Hybrid Clock

A hybrid timestamp can be used as a version when the value is stored in a key value store. The values are stored as discussed in *Versioned Value*.

class HybridClockReplicatedKVStore…

```
private Response applySetValueCommand(VersionedSetValueCommand setV
  mvccStore.put(setValueCommand.getKey(), setValueCommand.timestamp
  Response response = Response.success(setValueCommand.timestamp);
  return response;
}
```

class HybridClockMVCCStore…

```
ConcurrentSkipListMap<HybridClockKey, String> kv = new ConcurrentSk

public void put(String key, HybridTimestamp version, String value)
  kv.put(new HybridClockKey(key, version), value);
}
```

class HybridClockKey…

```
public class HybridClockKey implements Comparable<HybridClockKey> {
  private String key;
  private HybridTimestamp version;

  public HybridClockKey(String key, HybridTimestamp version) {
    this.key = key;
    this.version = version;
  }

  public String getKey() {
    return key;
  }

  public HybridTimestamp getVersion() {
    return version;
  }

  @Override
  public int compareTo(HybridClockKey o) {
    int keyCompare = this.key.compareTo(o.key);
    if (keyCompare == 0) {
       return this.version.compareTo(o.version);
    }
    return keyCompare;
  }
```

The values are read exactly as discussed in the Ordering of the versioned keys [versioned-value.xhtml#OrderingOfVersionedKeys]. The versioned keys are arranged in such a way as to form a natural ordering by using hybrid timestamps as a suffix to the key. This implementation enables us to get values for a specific version using the navigable map API.

class HybridClockMVCCStore…

```
public Optional<String> get(String key, HybridTimestamp atTimestamp
  Map.Entry<HybridClockKey, String> versionKeys = kv.floorEntry(new
  getLogger().info("Available version keys " + versionKeys + ". Rea
  return (versionKeys == null)? Optional.empty(): Optional.of(versi
}
```

Using timestamp to read values

Storing values with hybrid timestamp allows users to read using system timestamps in the past. For example, CockroachDB [bib-cockroachdb] allows executing queries with ‘AS OF SYSTEM TIME’ clause to specify date and time like ‘2016-10-03 12:45:00’ The values can be easily read as following:

class HybridClockMVCCStore…

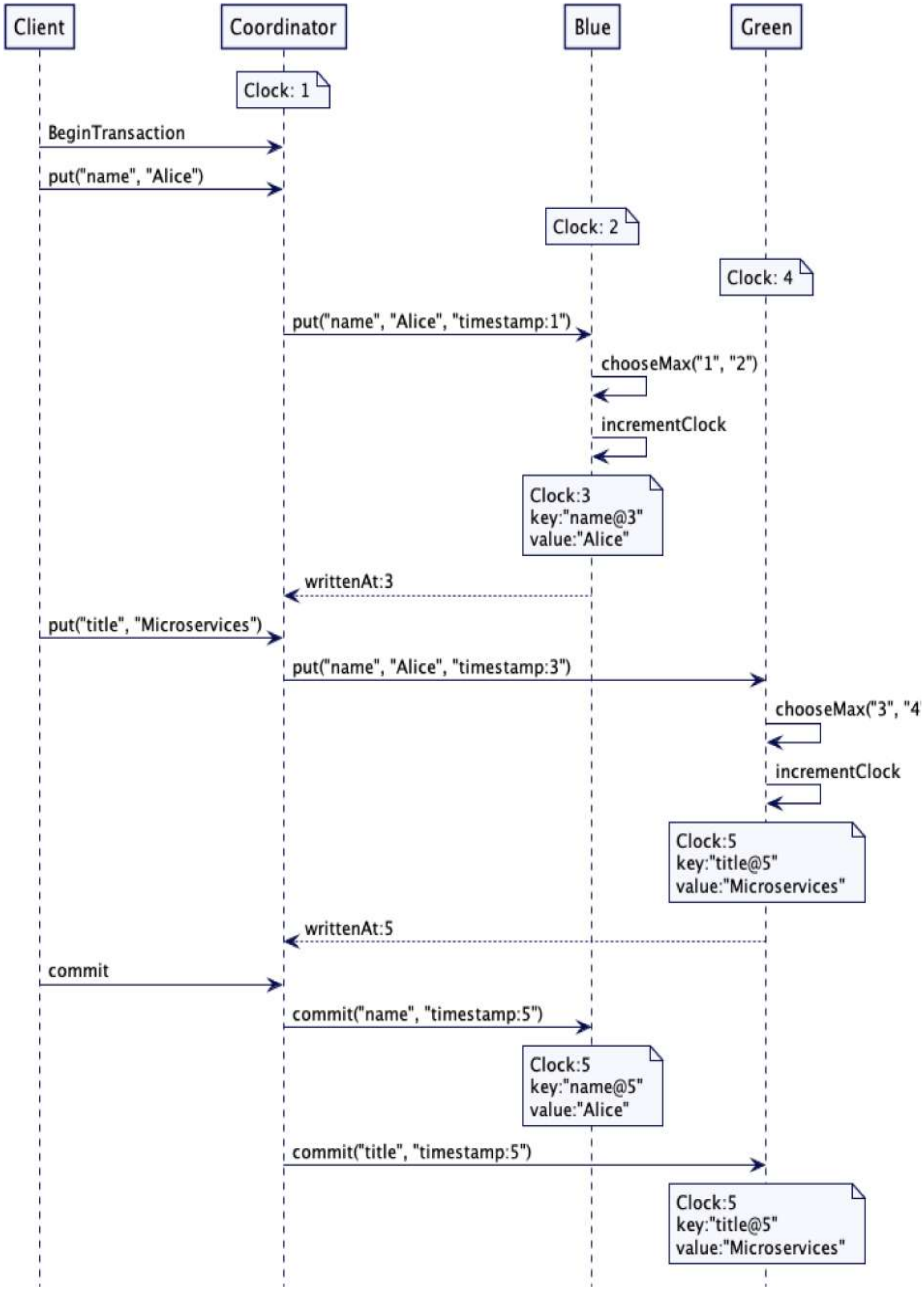
```
public Optional<String> getAtSystemTime(String key, String asOfSyst
  long time = Utils.parseDateTime(asOfSystemTimeClause);
  HybridTimestamp atTimestamp = new HybridTimestamp(time, 0);
  return get(key, atTimestamp);
}
```

Assigning timestamp to distributed transactions

Databases like MongoDB [bib-mongodb] and CockroachDB [bib-cockroachdb] use a *Hybrid Clock* to maintain causality with distributed transactions. With distributed transactions, it’s important to note that, all the values stored as part of the transaction should be stored at the same timestamp across the servers when the transaction commits. The requesting server might know about a higher timestamp in the later write requests. So the requesting server communicates with all the participating servers about the highest timestamp it received when the transaction commits. This fits very well with the standard [two-phase-commit] [bib-two-phase-commit] protocol for implementing transactions.

Following is the example of how the highest timestamp is determined at transaction commit. Assume that there are three servers. Server Blue stores names and Server Green stores titles. There is a separate server which acts as a coordinator. As can be seen, each server has a different local clock value. This can be a single integer or hybrid clock. The server acting as a coordinator starts writing to server Blue with the clock value known to it, which is 1. But Blue’s clock is at 2, so it increments that and writes the value at timestamp 3. Timestamp 3 is returned to the coordinator in the response.

For all subsequent requests to other servers, the coordinator uses timestamp of 3. Server Green, receiving the timestamp value 3 in the request, but it’s clock is at 4. So it picks up the highest value, which is 4. Increments it and writes the value at 5 and returns timestamp 5 to the coordinator. When the transaction commits, the coordinator uses the highest timestamp it received to commit the transaction. All the values updated in the transaction will be stored at this highest timestamp.


Client
Coordinator
Blue
Green
Clock: 1
BeginTransaction
put("name", "Alice")
Clock: 2
Clock: 4
put("name", "Alice", "timestamp:1")
chooseMax("1", "2")
incrementClock
Clock:3
key:"name@3"
value:"Alice"
writtenAt:3
put("title", "Microservices")
put("name", "Alice", "timestamp:3")
chooseMax("3", "4
incrementClock
Clock:5
key:"title@5"
value:"Microservices"
writtenAt:5
commit
commit("name", "timestamp:5")
Clock:5
key:"name@5"
value:"Alice"
commit("title", "timestamp:5")
Clock:5
key:"title@5"
value:"Microservices"

A very simplified code for timestamp handling with transactions looks like this:

class TransactionCoordinator…

```
public Transaction beginTransaction() {
  return new Transaction(UUID.randomUUID().toString());
}
public void putTransactionally() {
  Transaction txn = beginTransaction();
  HybridTimestamp coordinatorTime = new HybridTimestamp(1);
  HybridTimestamp server1WriteTime
     = server1.write("name", "Alice", coordinatorTime, txn);

  HybridTimestamp server2WriteTime = server2.write("title", "Micros

  HybridTimestamp commitTimestamp = server1WriteTime.max(server2Wri
  commit(txn, commitTimestamp);
}

private void commit(Transaction txn, HybridTimestamp commitTimestam
  server1.commitTxn("name", commitTimestamp, txn);
  server2.commitTxn("title", commitTimestamp, txn);
}
```

Transaction implementation can also use the prepare phase of the two phase commit protocol to learn about the highest timestamp used by each participating server.

Examples

MongoDB [bib-mongodb] uses hybrid timestamp to maintain versions in its MVCC storage.

CockroachDB [bib-cockroachdb] and YugabyteDB [bib-yugabyte] use hybrid timestamp to maintain causality with distributed transactions.

Chapter 24. Clock-Bound Wait

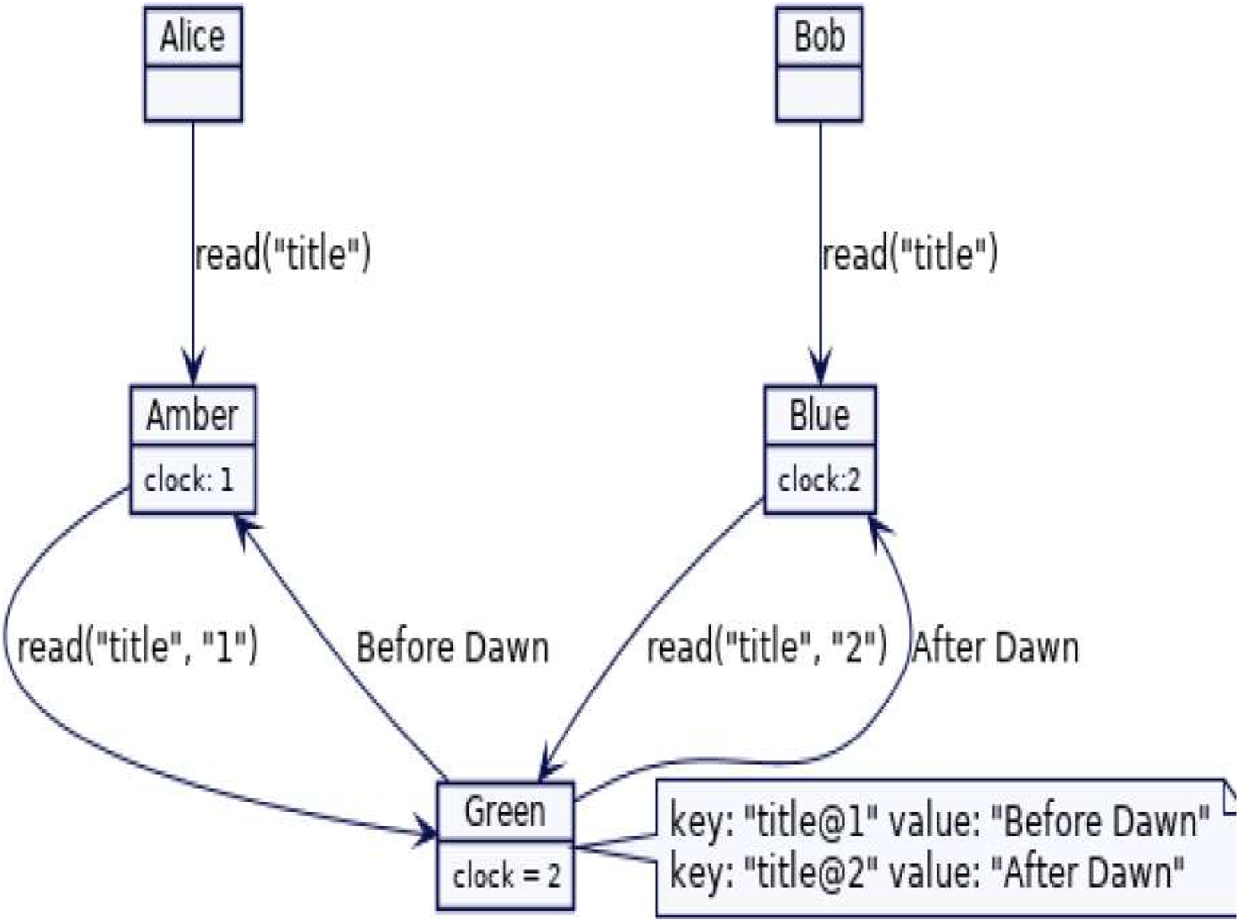
Wait to cover the uncertainty in time across cluster nodes before reading and writing values so values can be correctly ordered across cluster nodes.

Problem

> Both Alice and Bob can ask server Green for the latest version timestamp of the key they are trying to read. But that requires one extra round.
>
> If Alice and Bob are trying to read multiple keys, across a set of servers, they will need to ask the latest version for each and pick up the maximum value.

Consider a key-value store where values are stored with a timestamp to designate each version. Any cluster node that handles the client request will be able to read the latest version using the current timestamp at the request processing node.

In the following example, the value ‘Before Dawn’ is updated to value "After Dawn" at time 2, as per Green’s clock. Both Alice and Bob are trying to read the latest value for ‘title’. While Alice’s request is processed by cluster node Amber, Bob’s request is processed by cluster node Blue. Amber has its clock lagging at 1; which means that when Alice reads the latest value, it delivers the value ‘Before Dawn’. Blue has its clock at 2; when Bob reads the latest value, it returns the value as "After Dawn"



This violates a consistency known as external consistency [bib-external-consistency]. If Alice and Bob now make a phone call, Alice will be confused; Bob will tell that the latest value is "After Dawn", while her cluster node is showing "Before Dawn".

The same is true if Green's clock is fast and the writes happen in 'future' compared to Amber's clock.

This is a problem if system's timestamp is used as a version for storing values, because wall clocks are not monotonic [time-bound-lease.xhtml#wall-clock-not-monotonic]. Clock values from two different servers cannot and should not be compared. When *Hybrid Clock* is used as a version in *Versioned Value*, it allows values to be ordered on a single server as well as on different servers which are causally [bib-causal-consistency] related. However, Hybrid Clocks (or any *Lamport Clock* based clocks) can only give partial order. [lamport-clock.xhtml#PartialOrder] This means that any values which are not causally related and stored by two different clients across different nodes cannot be ordered. This creates a problem when using a timestamp to read the values across cluster nodes. If the read request originates on cluster nodes with lagging clocks, it probably won’t be able to read the most up to date versions of given values.

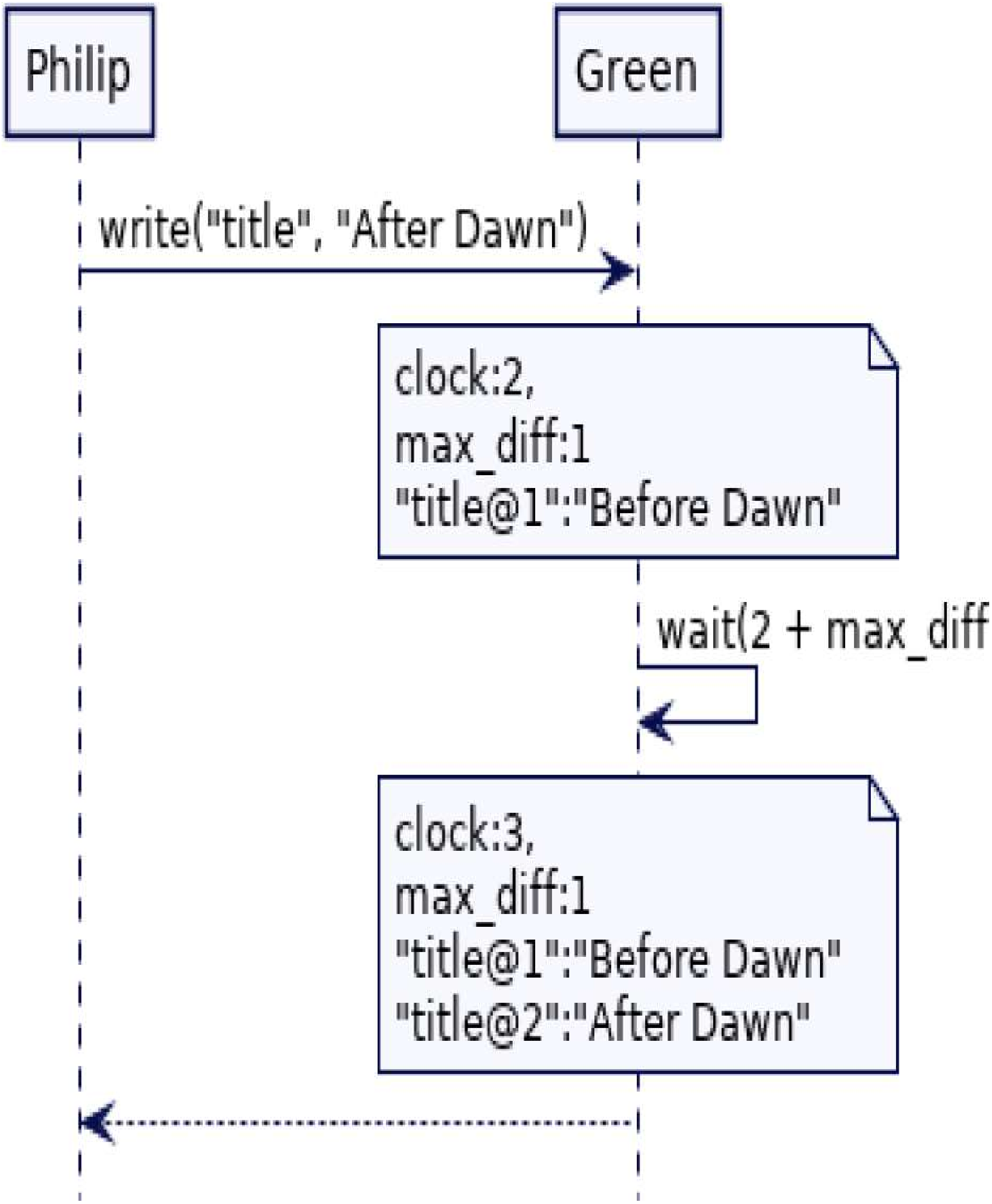
Solution

Cluster nodes wait until the clock values on every node in the cluster are guaranteed to be above the timestamp assigned to the value while reading or writting.

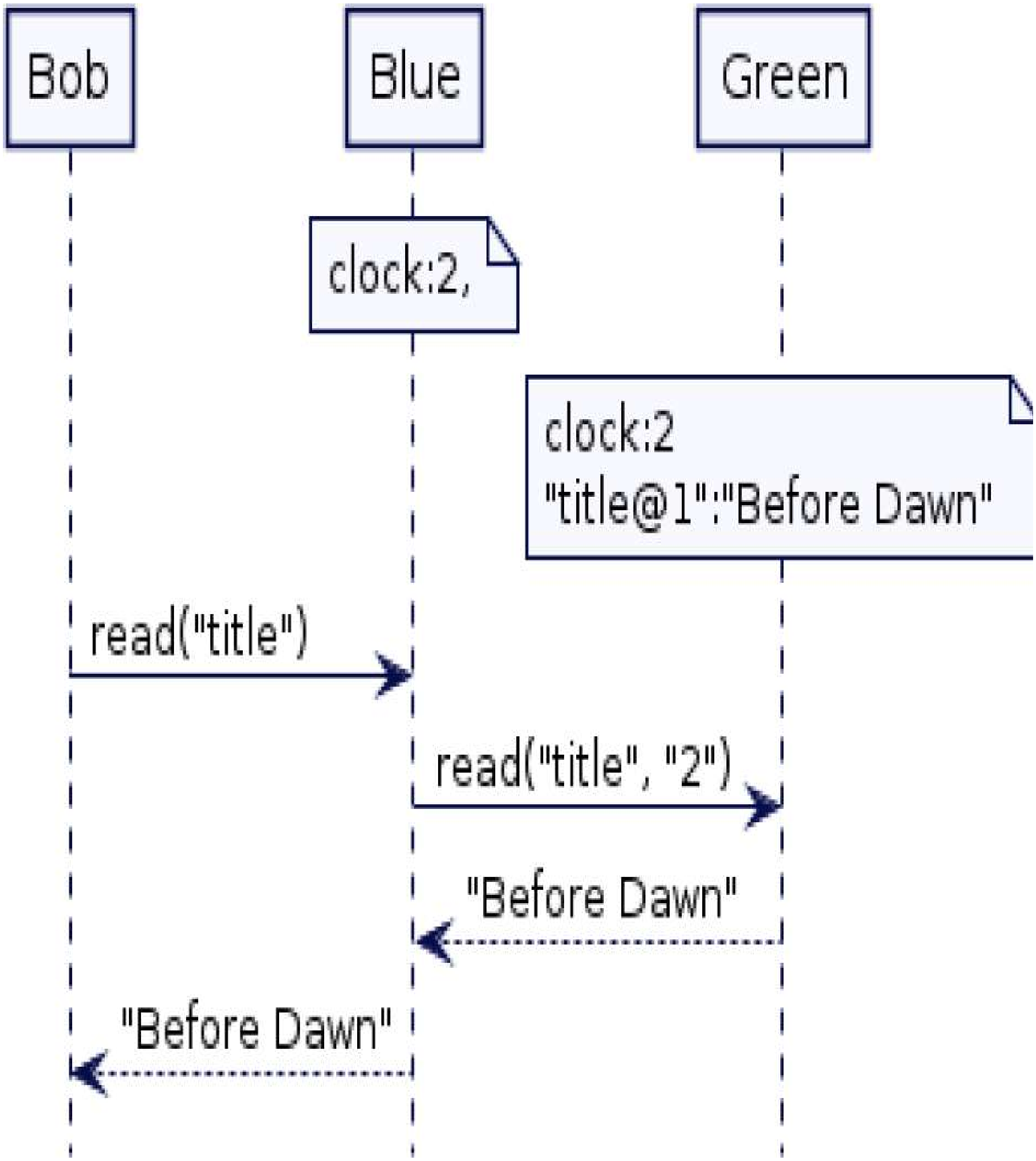
If the difference betweeen clocks is very small, write requests can wait without adding a great deal of overhead. As an example, assume the maximum clock offset across cluster nodes is 10ms. (This means that, at any given point in time, the slowest clock in the cluster is lagging behind t - 10ms.) To guarantee that every other cluster node has its clock set past t, the cluster node that handle any write operation will have to wait for t + 10ms before storing the value.

Consider a key value store with Versioned Value where each update is added as a new value, with a timestamp used as a version. In the Alice and Bob example mentioned above the write operation storing the title@2, will wait until all the clocks in the cluster are at 2. This makes sure that Alice will always see the latest value of the title even if the clock at the cluster node of Alice is lagging behind.

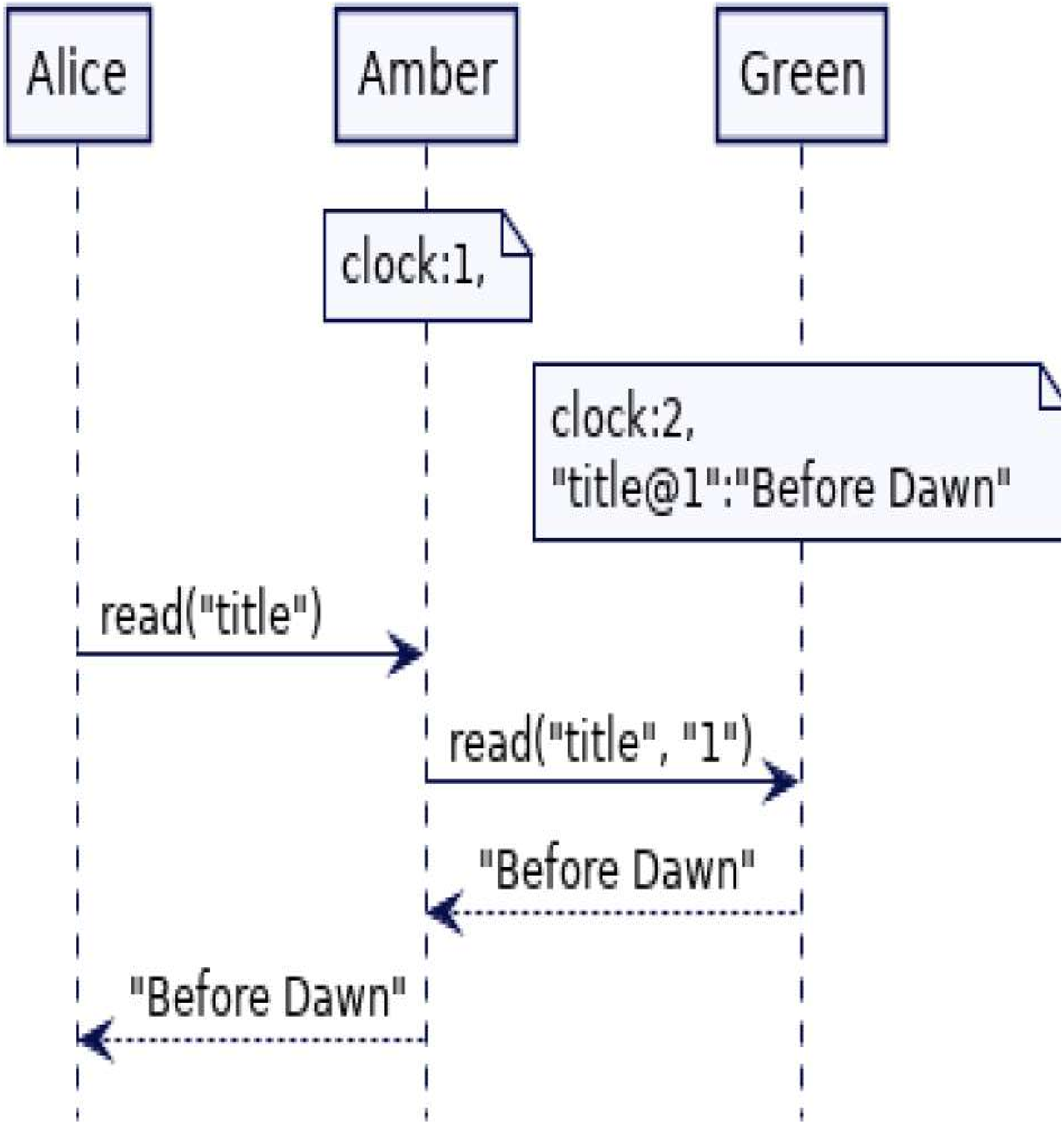
Consider a slightly different scenario. Philip is updating the title to ‘After Dawn’. Green’s clock has its time at 2. But Green knows that there might be a server with a clock lagging behind upto 1 unit. It will therefore have to wait in the write operation for a duration of 1 unit.



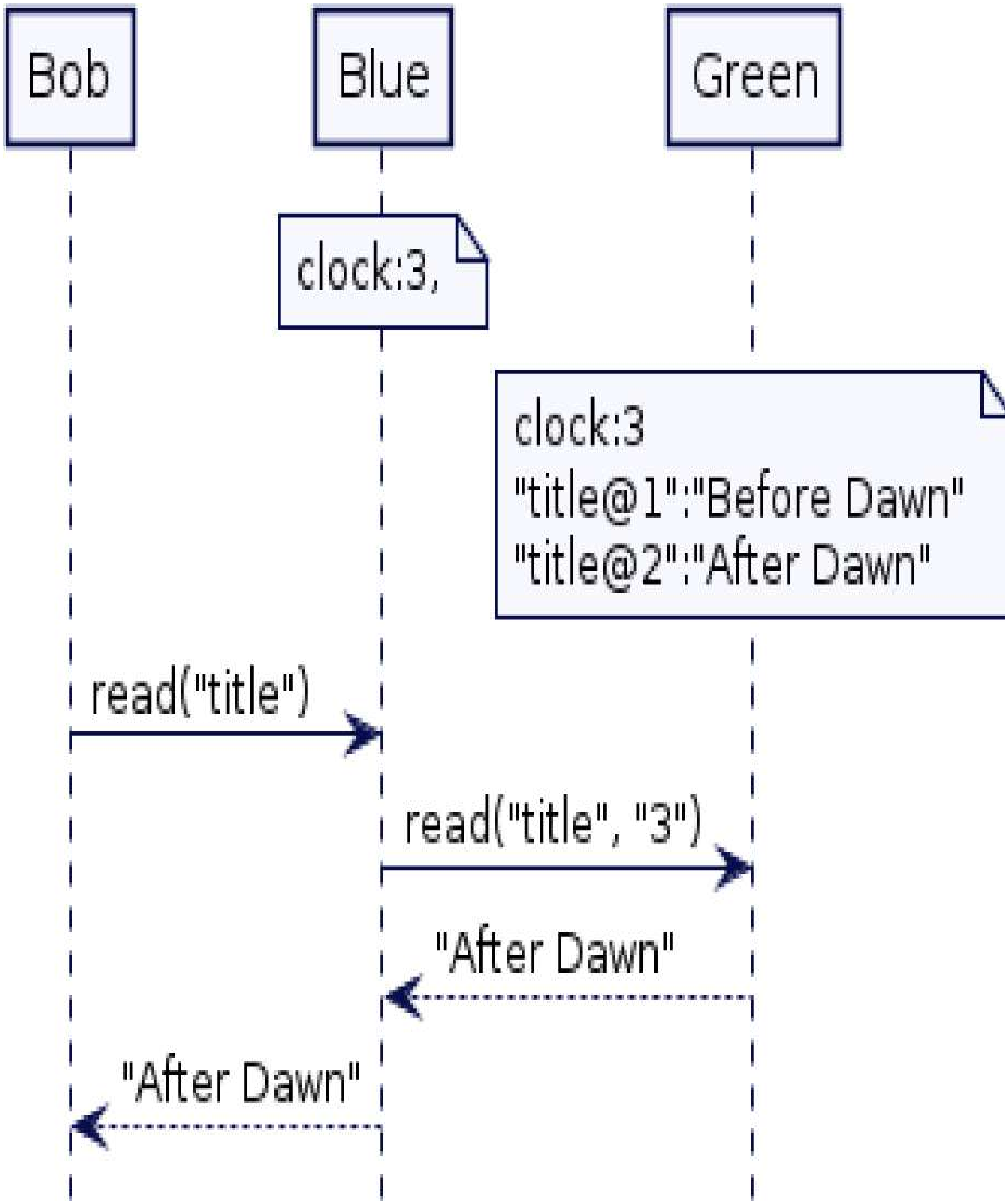
While Philip is updating the title, Bob's read request is handled by server Blue. Blue's clock is at 2, so it tries to read the title at timestamp 2. At this point Green has not yet made the value available. This means Bob gets the value at the highest timestamp lower than 2, which is 'Before Dawn'



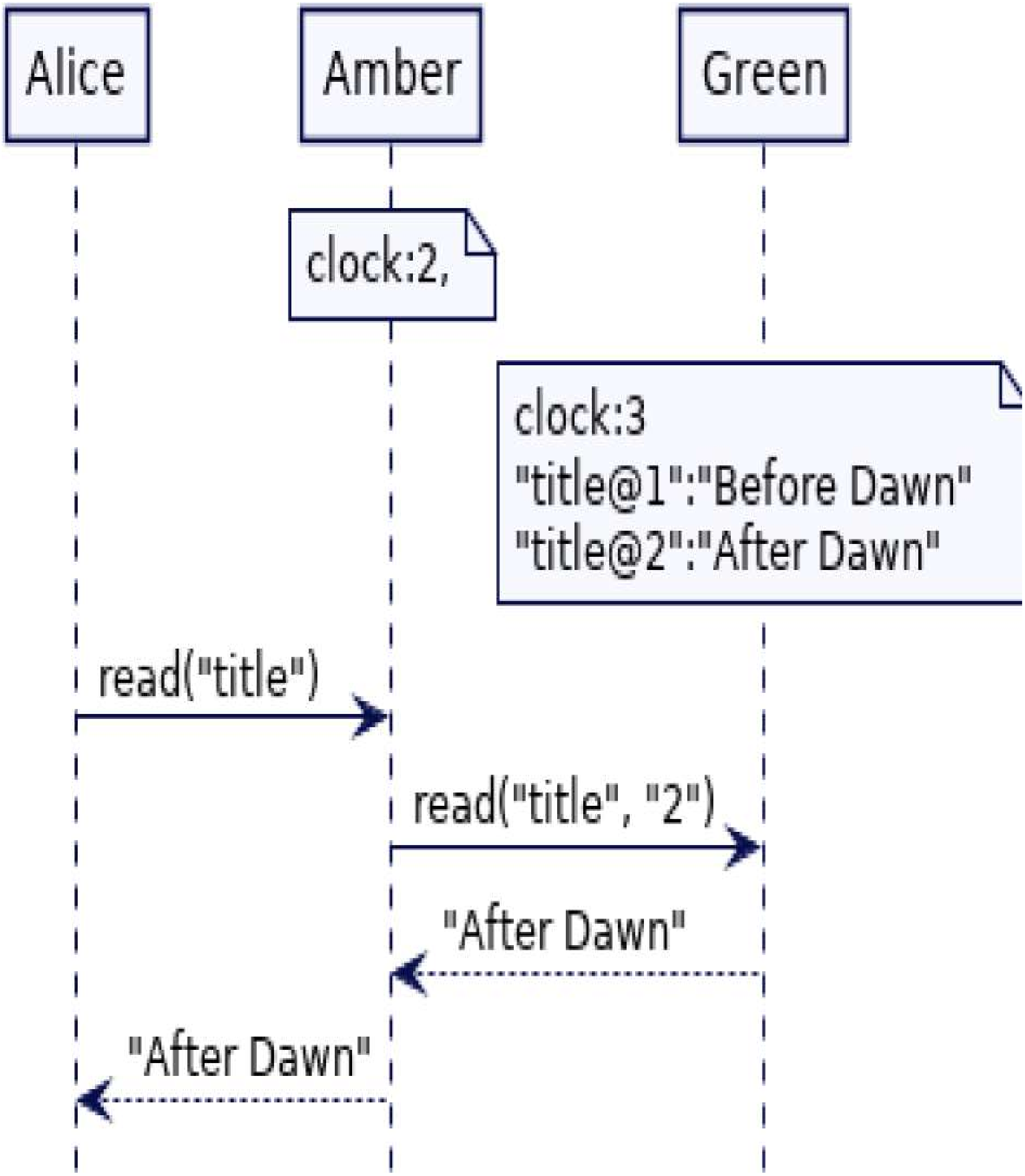
Alice's read request is handled by server Amber. Amber's clock is at 1 so it tries to read the title at timestamp 1. Alice gets the value 'Before Dawn'



Once Philip’s write request completes - after the wait of max_diff is over - if Bob now sends a new read request, server Blue will try to read the latest value according to its clock (which has advanced to 3); this will return the value "After Dawn"



If Alice initializes a new read request, server Amber will try to read the latest value as per its clock - which is now at 2. It will therefore, also return the value "After Dawn"



The main problem when trying to implement this solution is that getting the exact time difference across cluster nodes is simply not possible with the date/time hardware and operating systems APIs that are currently available. Such is the nature of the challenge that Google has its own specialized date time API called True Time [bib-external-consistency]. Similarly Amazon has AWS Time Sync Service [bib-aws-time-sync-service] and a library called

ClockBound [bib-clock-bound]. However, these APIs are very specific to Google and Amazon, so can't really be scaled beyond the confines of those organizations

Typically key value stores use Hybrid Clock to implement Versioned Value. While it is not possible to get the exact difference between clocks, a sensible default value can be chosen based on historical observations. Observed values for maximum clock drift on servers across datacenters is generally 200 to 500ms.

The key-value store waits for configured max-offset before storing the value.

class KVStore…

```
int maxOffset = 200;
NavigableMap<HybridClockKey, String> kv = new ConcurrentSkipListMap
public void put(String key, String value) {
  HybridTimestamp writeTimestamp = clock.now();
  waitTillSlowestClockCatchesUp(writeTimestamp);
  kv.put(new HybridClockKey(key, writeTimestamp), value);
}

private void waitTillSlowestClockCatchesUp(HybridTimestamp writeTim
  var waitUntilTimestamp = writeTimestamp.add(maxOffset, 0);
  sleepUntil(waitUntilTimestamp);
}

private void sleepUntil(HybridTimestamp waitUntil) {
  HybridTimestamp now = clock.now();
  while (clock.now().before(waitUntil)) {
    var waitTime = (waitUntil.getWallClockTime() - now.getWallClock
    Uninterruptibles.sleepUninterruptibly(waitTime, TimeUnit.MILLIS
    now = clock.now();
  }
}

public String get(String key, HybridTimestamp readTimestamp) {
  return kv.get(new HybridClockKey(key, readTimestamp));
}
```

Read Restart

200ms is too high an interval to wait for every write request. This is why databases like CockroachDB [bib-cockroachdb] or YugabyteDB [bib-yugabyte] implement a check in the read requests instead.

While serving a read request, cluster nodes check if there is a version available in the interval of readTimestamp and readTimestamp + maximum clock drift. If the version is available - assuming the reader's clock might be lagging - it is then asked to restart the read request with that version.

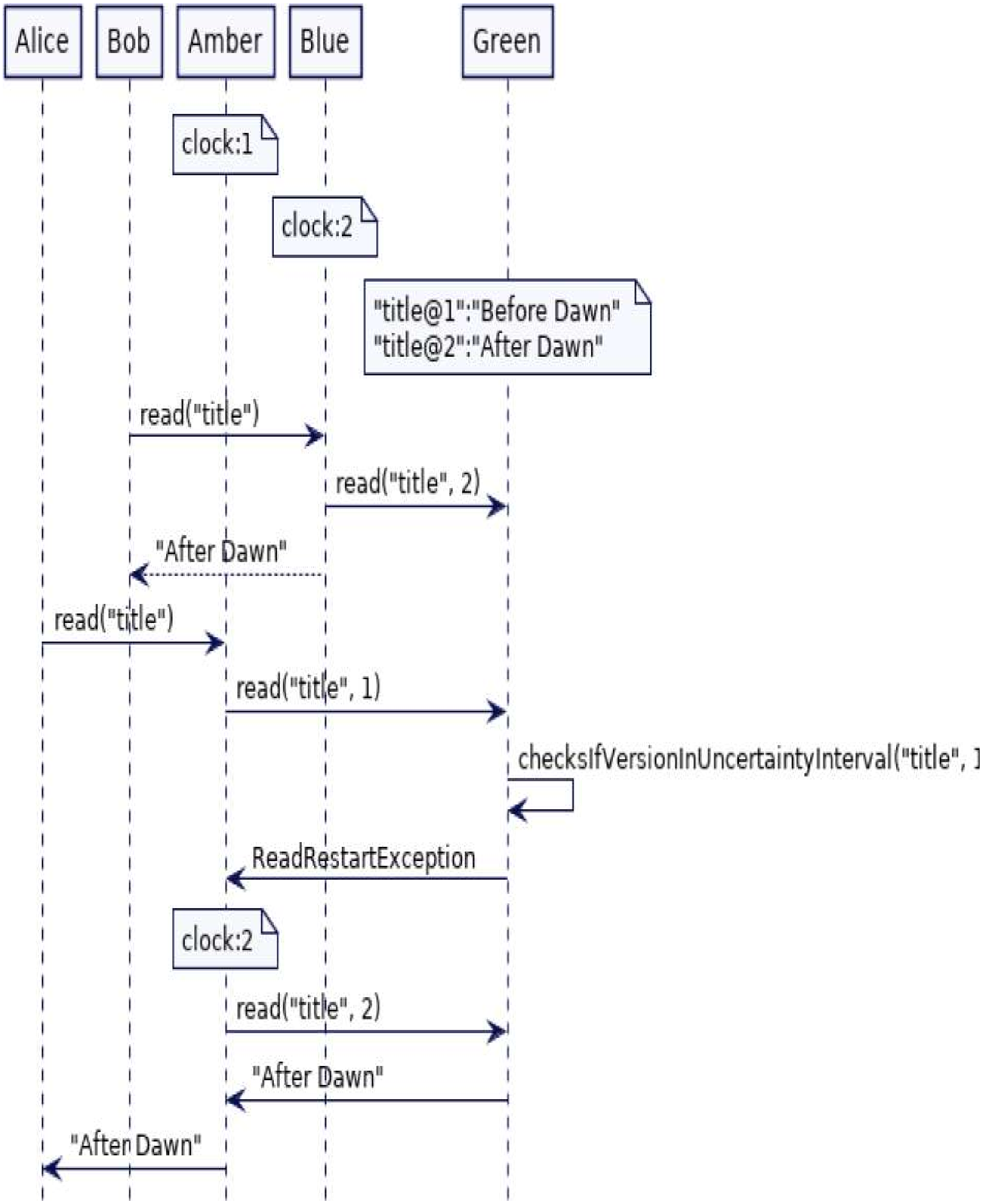
class KVStore…

```
public void put(String key, String value) {
  HybridTimestamp writeTimestamp = clock.now();
  kv.put(new HybridClockKey(key, writeTimestamp), value);
}

public String get(String key, HybridTimestamp readTimestamp) {
  checksIfVersionInUncertaintyInterval(key, readTimestamp);
  return kv.floorEntry(new HybridClockKey(key, readTimestamp)).getV
}

private void checksIfVersionInUncertaintyInterval(String key, Hybri
  HybridTimestamp uncertaintyLimit = readTimestamp.add(maxOffset, 0
  HybridClockKey versionedKey = kv.floorKey(new HybridClockKey(key,
  if (versionedKey == null) {
     return;
}

  HybridTimestamp maxVersionBelowUncertainty = versionedKey.getVers
  if (maxVersionBelowUncertainty.after(readTimestamp)) {
     throw new ReadRestartException(readTimestamp, maxOffset, maxVe
  }
  ;
}
```

class Client…

```
String read(String key) {
  int attemptNo = 1;
  int maxAttempts = 5;
  while(attemptNo < maxAttempts) {
    try {
      HybridTimestamp now = clock.now();
      return kvStore.get(key, now);
    } catch (ReadRestartException e) {
      logger.info(" Got read restart error " + e + "Attempt No. " +
      Uninterruptibles.sleepUninterruptibly(e.getMaxOffset(), TimeU
      attemptNo++;
    }
  }
  throw new ReadTimeoutException("Unable to read after " + attemptN
}
```

In the Alice and Bob example above, if there is a version for "title" available at timestamp 2, and Alice sends a read request with read timestamp 1, a ReadRestartException will be thrown asking Alice to restart the read request at readTimestamp 2.



Read restarts only happen if there is a version written in the uncertainty interval. Write request do not need to wait.

It’s important to remember that the configured value for maximum clock drift is an assumption, it is not guaranteed. In some cases, a bad server can have a clock drift more than the assumed value. In such cases, the problem will persist. [bib-ydb-causal-reverse]

Using Clock Bound APIs

Cloud providers like Google and Amazon, implement clock machinery with atomic clocks and GPS to make sure that the clock drift across cluster nodes is kept below a few milliseconds. As we’ve just discussed, Google has True Time [bib-external-consistency]. AWS has AWS Time Sync Service [bib-aws-time-sync-service] and ClockBound [bib-clock-bound].

There are two key requirements for cluster nodes to make sure these waits are implemented correctly.

- The clock drift across cluster nodes is kept to a minimum. Google’s True-Time keeps it below 1ms in most cases (7ms in the worst cases)
- The possible clock drift is always available in the date-time API, this ensures programmers don’t need to guess the value.

The clock machinery on cluster nodes computes error bounds for date-time values. Considering there is a possible error in timestamps returned by the local system clock, the API makes the error explicit. It will give the lower as well as the upper bound on clock values. The real time value is guaranteed to be within this interval.

```
public class ClockBound {
  public final long earliest;
  public final long latest;

  public ClockBound(long earliest, long latest) {
    this.earliest = earliest;
    this.latest = latest;
  }

  public boolean before(long timestamp) {
    return timestamp < earliest;
  }

  public boolean after(long timestamp)  {
    return timestamp > latest;
}
```

As explained in this AWS blog [bib-aws-clock-accuracy] the error is calculated at each cluster node as ClockErrorBound. The real time values will always be somewhere between local clock time and +-ClockErrorBound.

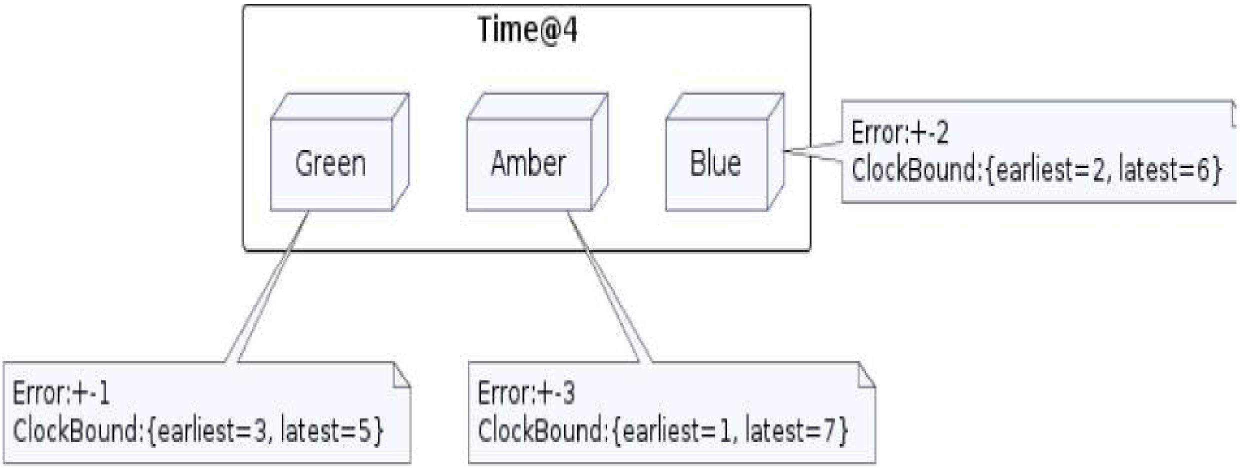
The error bounds are returned whenever date-time values are asked for.

```
public ClockBound now() {
  return now;
}
```

There are two properties guaranteed by the clock-bound API

- Clock bounds should overlap across cluster nodes
- For two time values t1 and t2, if t1 is less than t2, then clock_bound(t1).earliest is less than clock_bound(t2).latest across all cluster nodes

Imagine we have three cluster nodes: Green, Blue and Amber. Each node might have a different error bound. Let’s say the error on Green is 1, Blue is 2 and Amber is 3. At time=4, the clock bound across cluster nodes will look like this:


Time@4
Green
Amber
Blue
Error:+-2
ClockBound:{earliest=2, latest=6}
Error:+-1
ClockBound:{earliest=3, latest=5}
Error:+-3
ClockBound:{earliest=1, latest=7}

In this scenario, two rules need to be followed to implement the commit-wait.

- For any write operation, the clock bound's latest value should be picked as the timestamp. This will ensure that it is always higher than any timestamp assigned to previous write operations (considering the second rule below).
- The system must wait until the write timestamp is less than the clock bound's earliest value, before storing the value.

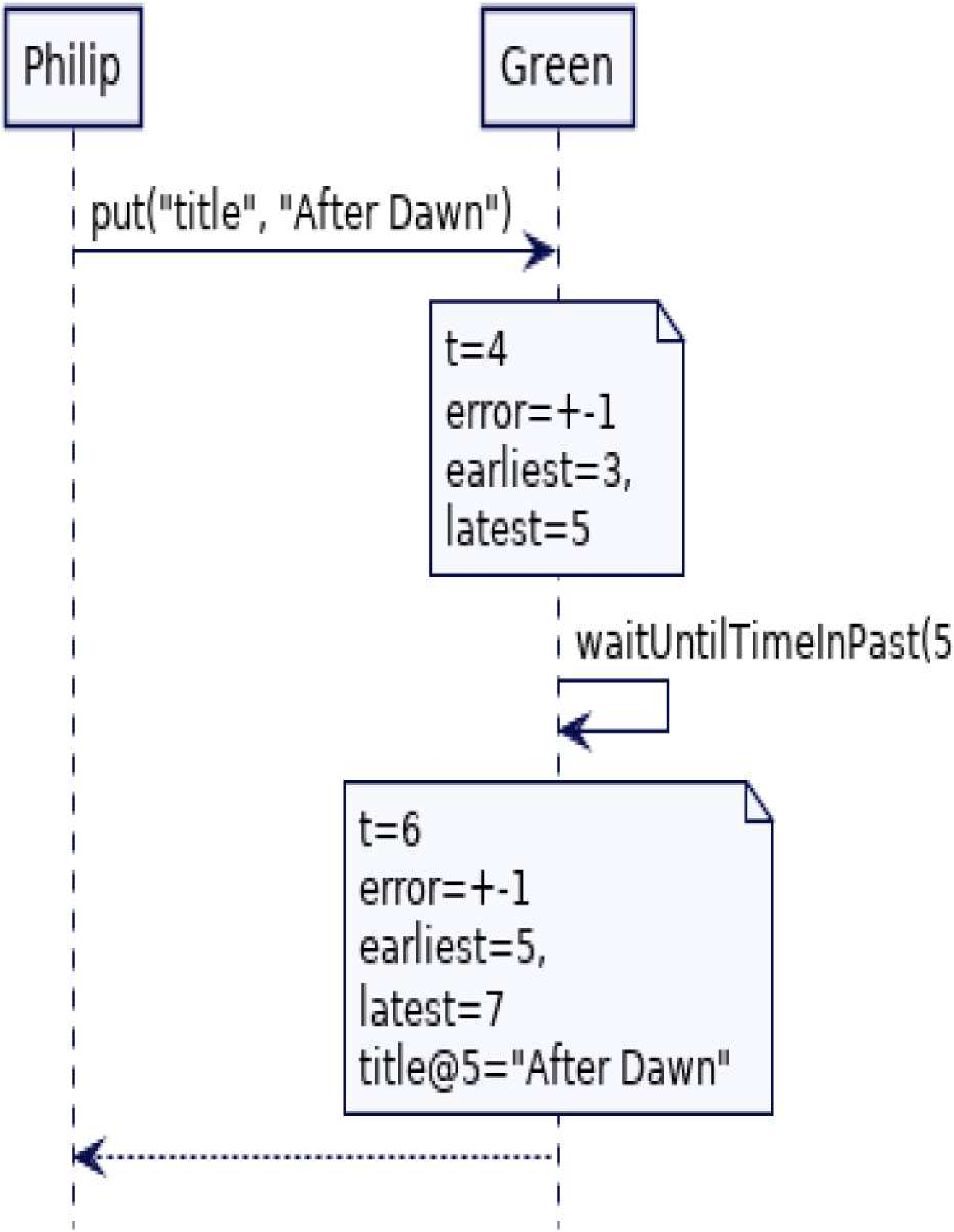
This is Because the earliest value is guaranteed to be lower than clock bound's latest values across all cluster nodes. This write operation will be accessible to anyone reading with the clock-bound's latest value in future. Also, this value is guaranteed to be ordered before any other write operation happen in future.

class KVStore…

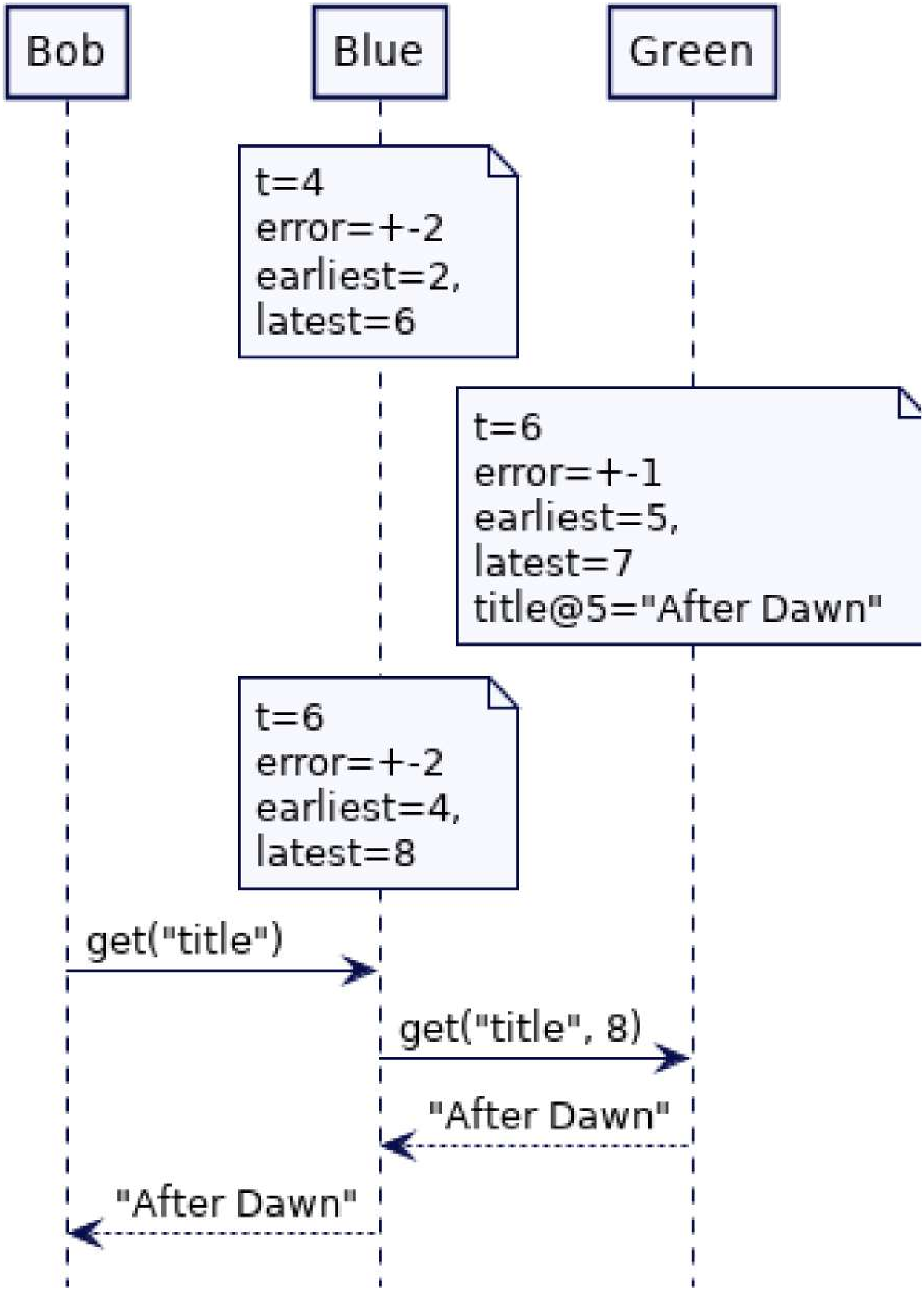
```
public void put(String key, String value) {
  ClockBound now = boundedClock.now();
  long writeTimestamp = now.latest;
  addPending(writeTimestamp);
  waitUntilTimeInPast(writeTimestamp);
  kv.put(new VersionedKey(key, writeTimestamp), value);
  removePending(writeTimestamp);
}
```

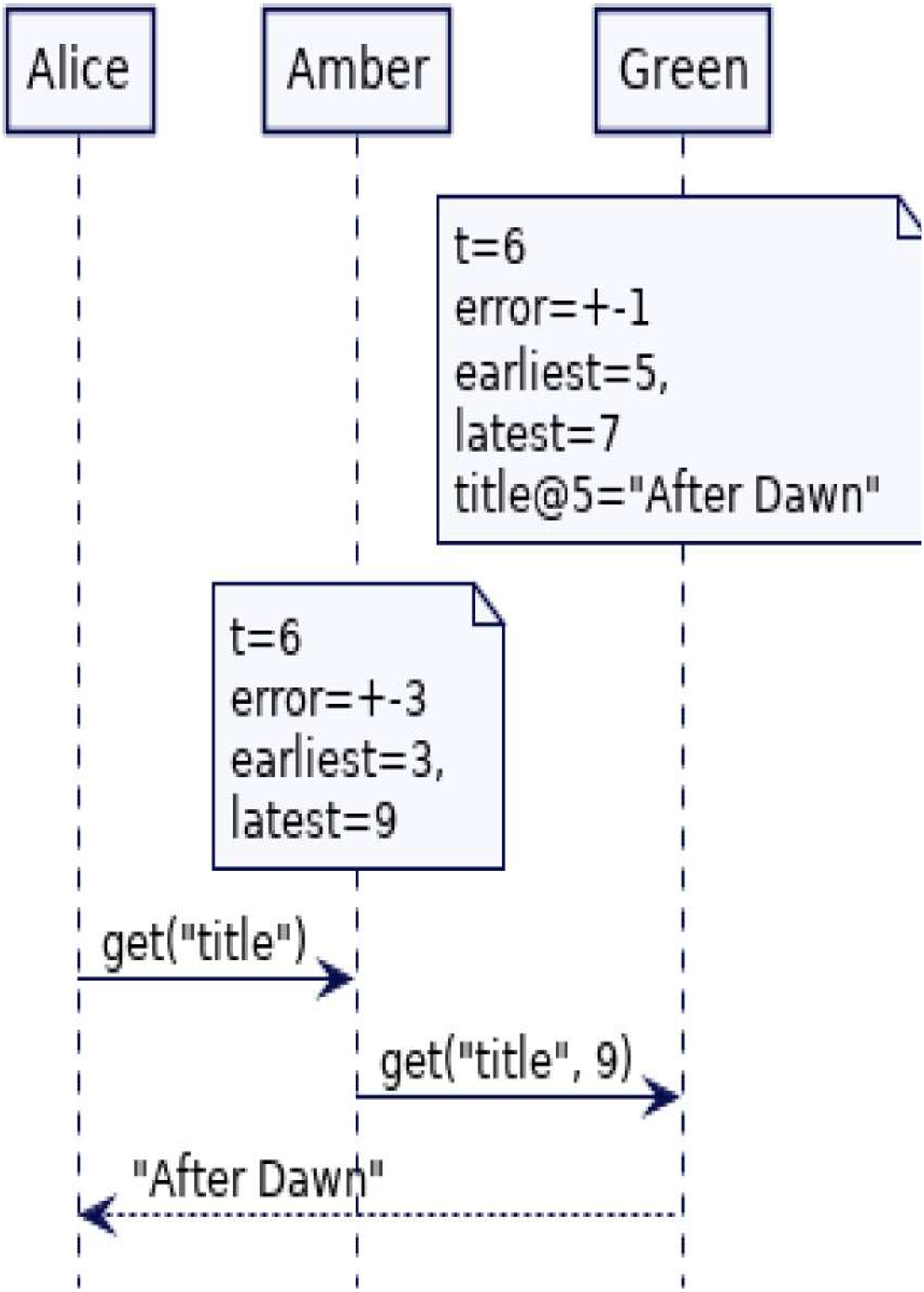
```
private void waitUntilTimeInPast(long writeTimestamp) {
  ClockBound now = boundedClock.now();
  while(now.earliest < writeTimestamp) {
    Uninterruptibles.sleepUninterruptibly(now.earliest - writeTimes
    now = boundedClock.now();
  }
}


private void removePending(long writeTimestamp) {
  pendingWriteTimestamps.remove(writeTimestamp);
  try {
     lock.lock();
     cond.signalAll();
  } finally {
    lock.unlock();
  }
}
private void addPending(long writeTimestamp) {
  pendingWriteTimestamps.add(writeTimestamp);
}
```

If we return to the Alice and Bob example above, when the value for "title"- "After Dawn" - is written by Philip on server Green, the put operation on Green waits until the chosen write timestamp is below the earliest value of the clock bound. This guarantees that every other cluster node is guaranteed to have a higher timestamp for the latest value of the clock bound. To illustrate, considering this scenario. Green has error bound of +-1. So, with a put operation which starts at time 4, when it stores the value, Green will pick up the latest value of clock bound which is 5. It then waits until the earliest value of the clock bound is more than 5. Essentially, Green waits for the uncertainty interval before actually storing the value in the key-value store.

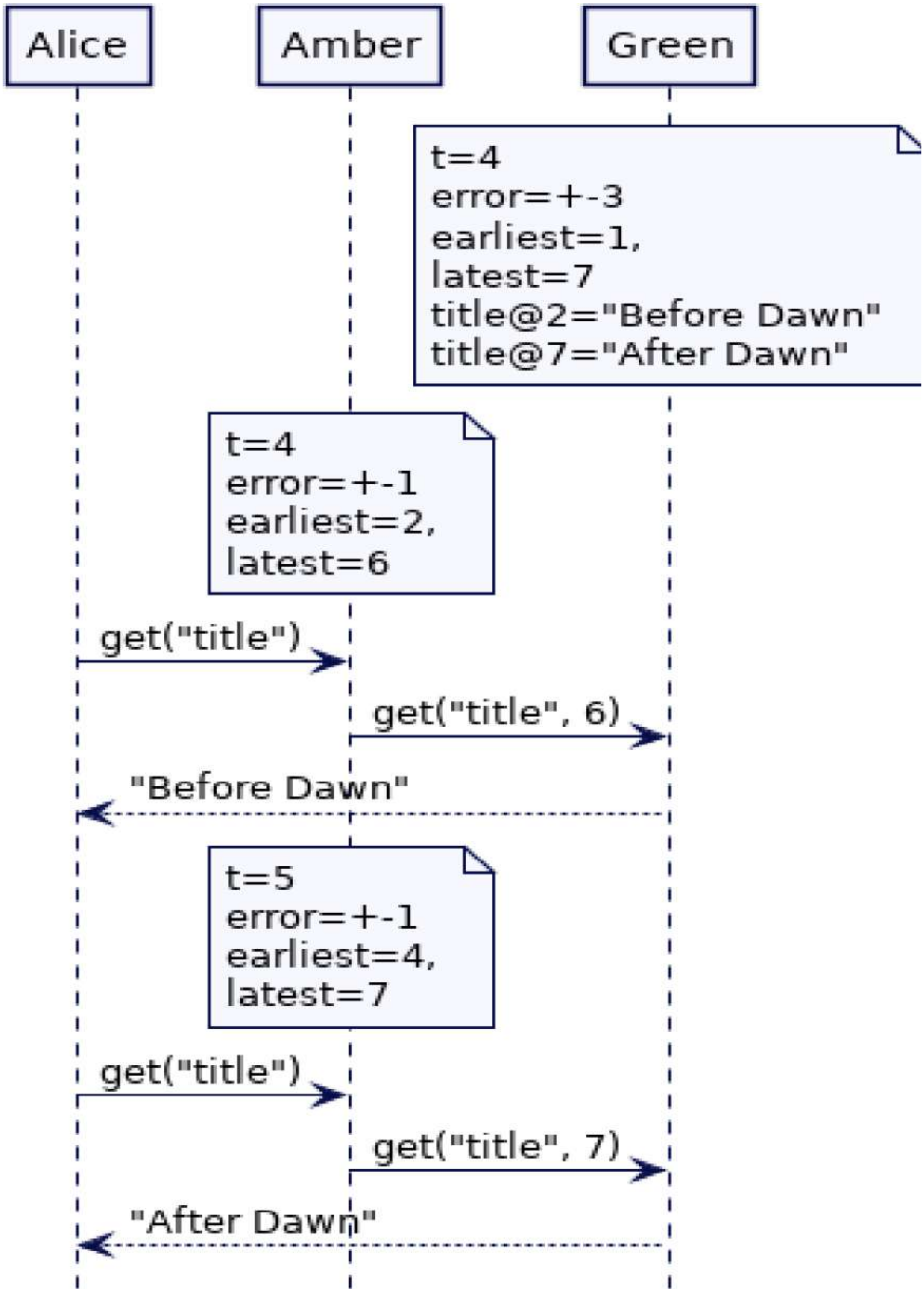


When the value is made available in the key value store, that the clock bound’s latest value is guaranteed to be higher than 5 on each and every cluster node. This means that Bob’s request handled by Blue as well as Alice’s request handled by Amber, are guaranteed to get the latest value of the title.


Bob
Blue
Green
t=4
error=+-2
earliest=2,
latest=6
t=6
error=+-1
earliest=5,
latest=7
title@5="After Dawn"
t=6
error=+-2
earliest=4,
latest=8
get("title")
get("title", 8)
"After Dawn"
"After Dawn"


Alice
Amber
Green
t=6
error=+-1
earliest=5,
latest=7
title@5="After Dawn"
t=6
error=+-3
earliest=3,
latest=9
get("title")
get("title", 9)
"After Dawn"

We will get the same result if Green has ‘wider’ time bounds. The greater the error bound, the longer the wait. If Green’s error bound is maximum, it will continue to wait before making the values available in the key-value store. Neither Amber nor Blue will be able to get the value until their latest time value is past 7. When Alice gets the most up-to-date value of title at latest time 7, every other cluster node will be guaranteed to get it at it’s latest time value.


Alice
Amber
Green
t=4
error=+-3
earliest=1,
latest=7
title@2="Before Dawn"
title@7="After Dawn"
t=4
error=+-1
earliest=2,
latest=6
get("title")
get("title", 6)
"Before Dawn"
t=5
error=+-1
earliest=4,
latest=7
get("title")
get("title", 7)
"After Dawn"

Read-Wait

When reading the value, the client will always pick the maximum value from the clock bound from its cluster node.

The cluster node that is receiving the request needs to make sure that once a response is returned at the specific request timestamp, there are no values written at that timestamp or the lower timestamp.

If the timestamp in the request is higher than the timestamp at the server, the cluster node will wait until the clock catches up, before returning the response.

It will then check if there are any pending write requests at the lower timestamp, which are not yet stored. If there are, then the read requests will pause until the requests are complete.

The server will then read the values at the request timestamp and return the value. This ensures that once a response is returned at a particular timestamp, no values will ever be written at the lower timestamp. This guarantee is called Snapshot Isolation [bib-snapshot-isolation]

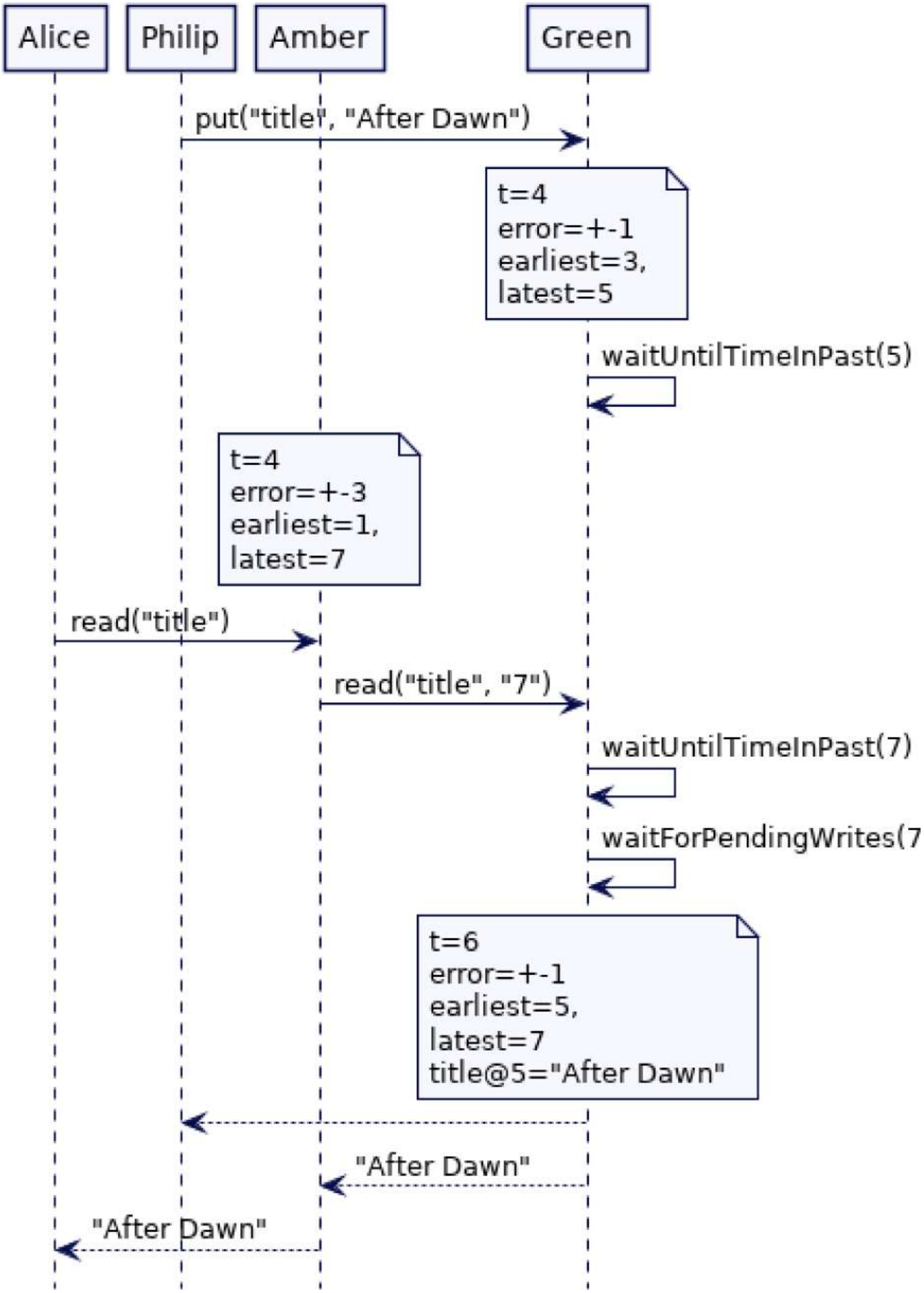
class KVStore…

```
final Lock lock = new ReentrantLock();
Queue<Long> pendingWriteTimestamps = new ArrayDeque<>();
final Condition cond  = lock.newCondition();

public Optional<String> read(long readTimestamp) {
  waitUntilTimeInPast(readTimestamp);
  waitForPendingWrites(readTimestamp);
  Optional<VersionedKey> max = kv.keySet().stream().max(Comparator.
  if(max.isPresent()) {
     return Optional.of(kv.get(max.get()));
  }
  return Optional.empty();
}
```

```
private void waitForPendingWrites(long readTimestamp) {
  try {
    lock.lock();
    while (pendingWriteTimestamps.stream().anyMatch(ts -> ts <= rea
      cond.awaitUninterruptibly();
    }
  } finally {
    lock.unlock();
  }
}
```

Consider this final scenario: Alice’s read request is handled by server Amber with error bound of 3. It picks up the latest time as 7 to read the title. Meanwhile, Philip’s write request is handled by Green (with an error bound of +-1), it picks up 5 to store the value.

Alice’s read request waits until the earliest time at Green is past 7 and the pending write request. It then returns the latest value with a timestamp below 7.


Alice
Philip
Amber
Green
put("title", "After Dawn")
t=4
error=+-1
earliest=3,
latest=5
waitUntilTimeInPast(5)
t=4
error=+-3
earliest=1,
latest=7
read("title")
read("title", "7")
waitUntilTimeInPast(7)
waitForPendingWrites(7
t=6
error=+-1
earliest=5,
latest=7
title@5="After Dawn"
"After Dawn"
"After Dawn"

Examples

Google’s TrueTime API [bib-external-consistency] provides us with a clock bound. Spanner [bib-spanner] uses it to implement commit-wait

AWS Time Sync Service [bib-aws-time-sync-service]ensures minimal clock drifts. It is possible to use the ClockBound [bib-clock-bound] API to implement waits to order the events across the cluster.

CockroachDB [bib-cockroachdb] implements read restart. It also has an experimental option to use commit-wait based on the configured maximum clock drift value.

YugabyteDB [bib-yugabyte] implements read restart based on the configured maximum clock drift value.

Part V: Patterns of Cluster Management

Chapter 25. Consistent Core

Maintain a smaller cluster providing stronger consistency to allow large data cluster to coordinate server activities without implementing quorum based algorithms.

Problem

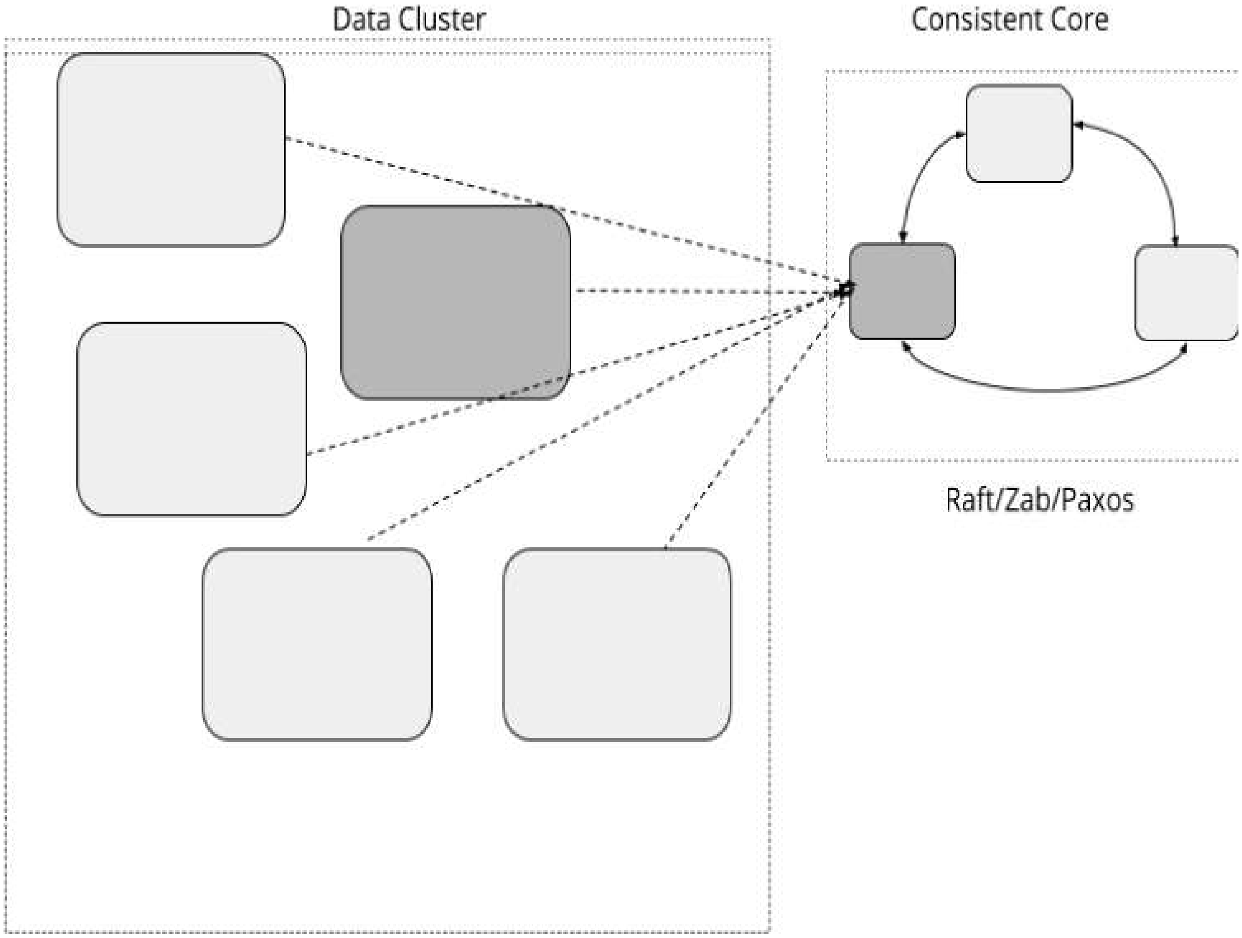
> Linearizability [bib-Linearizable] is the strongest consistency guarantee where all the clients are guaranteed to see latest committed updates to data. Providing linearizability along with fault tolerance needs consensus [bib-consensus] algorithms like Raft [bib-raft], Zab [bib-zab] or Paxos [bib-paxos] to be implemented on the servers.
>
> While consensus algorithm is an essential requirement to implement a Consistent Core, there are various aspects of client interaction - such as how a client finds the leader, how duplicate requests are handled, etc - which are important implementation decisions. There are also some important implementation considerations regarding safety and liveness. Paxos defines only the consensus algorithm, but these other implementation aspects are not well documented in the Paxos literature. Raft very clearly documents various implementation aspects, along with a reference implementation [bib-logcabin-raft] and therefore is the most widely used algorithm today.

When a cluster needs to handle a lot of data, it needs more and more servers. For a cluster of servers, there are some common requirements, such as selecting a specific server to be the master for a particular task, managing group membership information, mapping of data partitions to the servers etc.

These requirements need strong consistency guarantee, namely linearizability. The implementation also needs to be fault tolerant. A common approach is to use a fault-tolerant consensus algorithms based on *Quorum*. But in quorum-based systems throughput degrades with the size of the cluster.

Solution

Implement a smaller, 3 to 5 node cluster which provides linearizability guarantee as well as fault tolerance. [1] A separate data cluster can use the small consistent cluster to manage metadata and for taking cluster wide decisions with primitives like *Lease*. This way, the data cluster can grow to a large number of servers, but can still do certain actions which need strong consistency guarantees using the smaller metadata cluster.



A typical interface of consistent core looks like this:

```
public interface ConsistentCore {
  CompletableFuture put(String key, String value);

  List<String> get(String keyPrefix);

  CompletableFuture registerLease(String name, long ttl);

  void refreshLease(String name);

  void watch(String name, Consumer<WatchEvent> watchCallback);
}
```

At the minimum, Consistent Core provides a simple key value storage mechanism. It is used to store metadata.

Metadata Storage

The storage is implemented using consensus algorithms such as Raft. It is an example of Replicated Write Ahead Log implementation, where replication is handled by *Leader and Followers* and *High-Water Mark* is used to track the successful replication using *Quorum*

Supporting hierarchical storage

Consistent Core is generally used to store data for things like: group membership or task distribution across servers. A common usage pattern is to scope the type of metadata with a prefix. e.g. for group membership, the keys will all be stored like / servers/1, servers/2 etc. For tasks assigned to servers the keys can be /tasks/task1, / tasks/task2. This data is generally read with all the keys with a specific prefix. For example, to get information about all the servers in the cluster, all the keys with prefix / servers are read.

An example usage is as following:

[1] Because the entire cluster depends on the Consistent Core, it is critical to be aware of the details of the consensus algorithm used. Consensus implementations can run into liveness issues in some tricky network partition situations. For example, a Raft cluster can be disrupted by a partitioned server, which can continuously trigger leader election, unless special care is taken. This recent incident at Cloudflare [bib-cloudflare-outage] is a good example to learn from.

The servers can register themselves with the Consistent Core by creating their own key with prefix /servers.

```
client1.setValue("/servers/1", "{address:192.168.199.10, port:8000}

client2.setValue("/servers/2", "{address:192.168.199.11, port:8000}

client3.setValue("/servers/3", "{address:192.168.199.12, port:8000}
```

The clients can then get to know about all the servers in the cluster by reading with key prefix /servers as following:

```
assertEquals(client1.getValue("/servers"), Arrays.asList("{address:
                                              "{address:192.168.199.11, port
                                              "{address:192.168.199.10, port
```

Because of this hierarchical nature of data storage, products like Zookeeper [bib-zookeeper], [chubby] [bib-chubby] provide a file system like interface, where users create directories and files, or nodes, with the concept of parent and child nodes. etc3d [bib-etcd3] has a flat key space with the ability to get a range of keys.

Handling Client Interactions

One of the key requirements for Consistent Core functionality is how a client interacts with the core. The following aspects are critical for the clients to work with the Consistent Core.

Finding the leader

Serializability and Linearizability

When read requests are handled by follower servers, it is possible that clients can get stale data, as the latest commits from the leader have not reached the followers. The order in which the updates are received by the client is still maintained but the updates might not be most recent. This is the [serializability] [bib-serializability] guarantee as opposed to linearizability [bib-Linearizable]. Linearizability guarantees that every client gets the most recent updates. Clients can work with serializability guarantee when they just need to read metadata and can tolerate stale metadata for a while. For operations like *Lease*, linearizability is strictly needed.

If the leader is partitioned from the rest of the cluster, clients can get stale values from the leader, Raft describes a mechanism to provide linearizable reads. See for example etcd [bib-etcd-readindex-impl] implementation of readIndex. YugabyteDB [bib-yugabyte] uses a technique called [yugabyte-leader-lease] [bib-yugabyte-leader-lease] to achieve the same.

A similar situation can happen with followers which are partitioned. The follower may be partitioned and might not return the latest values to the client. To make sure that the followers are not partitioned and are up-to-date with the leader, they need to query the leader to know the latest updates, and wait till they receive the latest updates before responding to the client, See the proposed kafka design [bib-kafka-enhanced-raft] for example.

It’s important that all the operations are executed on the leader, so a client library needs to find the leader server first. There are two approaches possible to fulfil this requirement.

Products like zookeeper and etcd implement this approach because they allow some read-only requests to be handled by the follower servers; this avoids a bottleneck on the leader when a large number of clients are read-only. This reduces complexity in the clients to connect to either leader or follower based on the type of the request.

- The follower servers in the consistent core know about the current leader, so if the client connects to a follower, it can return the address of the leader. The client can then directly connect to the leader identified in the response. It should be noted that the servers might be in the middle of leader election when the client tries to connect. In that case, servers cannot return the leader address and the client needs to wait and try another server.

- Servers can implement a forwarding mechanism and forward all the client requests to the leader. This allows clients to connect to any server. Again, if servers are in the middle of leader election, then clients need to retry until the leader election is successful and a legitimate leader is established.

A simple mechanism to find the leader is to try to connect to each server and try to send a request, the server responds with a redirect response if it’s not the leader.

```
private void establishConnectionToLeader(List<InetAddressAndPort> s
  for (InetAddressAndPort server : servers) {
    try {
       SingleSocketChannel socketChannel = new SingleSocketChannel(
       logger.info("Trying to connect to " + server);
       RequestOrResponse response = sendConnectRequest(socketChanne
       if (isRedirectResponse(response)) {
          redirectToLeader(response);
          break;
       } else if (isLookingForLeader(response)) {
         logger.info("Server is looking for leader. Trying next ser
         continue;
       } else { //we know the leader
         logger.info("Found leader. Establishing a new connection."
         newPipelinedConnection(server);
         break;
       }
    } catch (IOException e) {
       logger.info("Unable to connect to " + server);
       //try next server
    }
  }
}

private boolean isLookingForLeader(RequestOrResponse requestOrRespo
  return requestOrResponse.getRequestId() == RequestId.LookingForLe
}

private void redirectToLeader(RequestOrResponse response) {
  RedirectToLeaderResponse redirectResponse = deserialize(response)
  newPipelinedConnection(redirectResponse.leaderAddress);

  logger.info("Connected to the new leader "
       + redirectResponse.leaderServerId
       + " " + redirectResponse.leaderAddress
       + ". Checking connection");
}


private boolean isRedirectResponse(RequestOrResponse requestOrRespo
  return requestOrResponse.getRequestId() == RequestId.RedirectToLe
}
```

Just establishing TCP connection is not enough, we need to know if the server can handle our requests. So clients send a special connection request for the server to acknowledge if it can serve the requests or else redirect to the leader server.

```
private RequestOrResponse sendConnectRequest(SingleSocketChannel so
  RequestOrResponse request
      = new RequestOrResponse(RequestId.ConnectRequest.getId(), Jso
  try {
     return socketChannel.blockingSend(request);
  } catch (IOException e) {
     resetConnectionToLeader();
     throw e;
  }
}
```

If an existing leader fails, the same technique is used to identify the newly elected leader from the cluster.

Once connected, the client maintains a *Single Socket Channel* to the leader server

Handling duplicate requests

In cases of failure, clients may try to connect to the new leader, resending the requests. But if those requests were already handled by the failed leader prior to failure, it might result in duplicates. Therefore, it’s important to have a mechanism on the servers to ignore duplicate requests. *Idempotent Receiver* pattern is used to implement duplicate detection.

Coordinating tasks across a set of servers can be done by using *Lease*. The same can be used to implement group membership and failure detection mechanism.

State Watch is used to get notifications of changes to the metadata or time bound leases.

Examples

Google is known to use [chubby] [bib-chubby] lock service for coordination and metadata management.

Kafka [bib-kafka] uses Zookeeper [bib-zookeeper] to manage metadata and take decisions like leader election for cluster master. The proposed architecture change [bib-kip-500] in Kafka will replace zookeeper with its own Raft [bib-raft] based controller cluster.

[bookkeeper] [bib-bookkeeper] uses Zookeeper to manage cluster metadata.

Kubernetes [bib-kubernetes] uses etcd [bib-etcd] for coordination, manage cluster metadata and group membership information.

All the big data storage and processing systems like [hdfs] [bib-hdfs], [spark] [bib-spark], [flink] [bib-flink] use Zookeeper [bib-zookeeper] for high availability and cluster coordination.

Chapter 26. Lease

Use time bound leases for cluster nodes to coordinate their activities.

Problem

Cluster nodes need exclusive access to certain resources. But nodes can crash; they can be temporarily disconnected or experiencing a process pause. Under these error scenarios, they should not keep the access to a resource indefinitely.

Solution

Wall Clocks are not monotonic

Computers have two different mechanisms to represent time. The wall clock time, which represents the time of the day, is measured by a clock machinery generally built with an crystal oscillator. The known problem with this mechanism is that it can drift away from the actual time of the day, based on how fast or slow the crystals oscillate. To fix this, computers typically have a service like NTP [bib-ntp] setup, which checks the time of the day with well known time sources over the internet and fixes the local time. Because of this, two consecutive readings of the wall clock time in a given server can have time going backwards. This makes the wall clock time unsuitable for measuring the time elapsed between some events. Computers have a different mechanism called monotonic clock, which indicates elapsed time. The values of monotonic clock are not affected by services like NTP. Two consecutive calls of monotonic clock are guaranteed to get the elapsed time. So for measuring timeout values monotonic clocks are always used. This works well on a single server. But monotonic clocks on two different servers cannot be compared. All programming languages have an api to read both the wall clock and the monotonic clock. e.g. In Java System.currentMillis gives wall clock time and System.nanoTime gives monotonic clock time.

A cluster node can ask for a lease for a limited period of time, after which it expires. The node can renew the lease before it expires if it wants to extend the access. Implement the lease mechanism with *Consistent Core* to provide fault tolerance, and consistency. Have a ‘time to live’ value associated with the lease. Cluster nodes can create keys in a Consistent Core with a lease attached to it. The leases are replicated with the *Leader and Followers* to provide fault tolerance. It’s the responsibility of the node which owns the lease to periodically refresh it. *HeartBeat* is used by the clients to refresh the time to live value in the consistent core. The leases are created on all the nodes in the Consistent Core, but only the leader tracks the lease timeouts. [1] The timeouts are not tracked on the followers in the Consistent Core. This is done because we need the leader to decide when the lease expires using its own monotonic clock, and then let the followers know when the lease expires. This makes sure that, like any other decision in the Consistent Core, nodes also reach consensus about lease expiration.

When a node from a consistent core becomes a leader, it starts tracking leases.

class ReplicatedKVStore…

```
public void onBecomingLeader() {
  leaseTracker = new LeaderLeaseTracker(this, new SystemClock(), lo
  leaseTracker.start();
}
```

[1] LogCabin, the reference implementation of Raft [bib-logcabin-raft-clustertime] has an interesting concept of ClusterTime, which is a logical clock maintained for the whole Raft cluster. With all the nodes in the cluster agreeing on the time, they can independently remove expired sessions. But it needs heartbeat entries from leader to followers to be replicated and committed like any other log entries.

Leader starts a scheduled task to periodically check for lease expiration

class LeaderLeaseTracker…

```
private ScheduledThreadPoolExecutor executor = new ScheduledThreadP
private ScheduledFuture<?> scheduledTask;
@Override
public void start() {
  scheduledTask = executor.scheduleWithFixedDelay(this::checkAndExp
      leaseCheckingInterval,
      leaseCheckingInterval,
      TimeUnit.MILLISECONDS);
}

@Override
public void checkAndExpireLeases() {
  remove(expiredLeases());
}

private void remove(Stream<String> expiredLeases) {
  expiredLeases.forEach((leaseId)->{
    //remove it from this server so that it doesnt cause trigger ag
    expireLease(leaseId);
    //submit a request so that followers know about expired leases
    submitExpireLeaseRequest(leaseId);
  });
}

private Stream<String> expiredLeases() {
  long now = System.nanoTime();
  Map<String, Lease> leases = kvStore.getLeases();
  return  leases.keySet().stream()
       .filter(leaseId -> {
  Lease lease = leases.get(leaseId);
  return lease.getExpiresAt() < now;
  });
}
```

Followers start a no-op lease tracker.

class ReplicatedKVStore…

```
public void onCandidateOrFollower() {
  if (leaseTracker != null) {
     leaseTracker.stop();
  }
  leaseTracker = new FollowerLeaseTracker(this, leases);
}
```

The lease is represented simply as following:

```
public class Lease implements Logging {
  String name;
  long ttl;
  //Time at which this lease expires
  long expiresAt;

  //The keys from kv store attached with this lease
  List<String> attachedKeys = new ArrayList<>();

  public Lease(String name, long ttl, long now) {
    this.name = name;
    this.ttl = ttl;
    this.expiresAt = now + ttl;
  }

  public String getName() {
    return name;
  }

  public long getTtl() {
    return ttl;
  }

  public long getExpiresAt() {
    return expiresAt;
  }

  public void refresh(long now) {
    expiresAt = now + ttl;
    getLogger().info("Refreshing lease " + name + " Expiration time
  }

  public void attachKey(String key) {
    attachedKeys.add(key);
  }
  public List<String> getAttachedKeys() {
    return attachedKeys;
  }
}
```

When a node wants to create a lease, it connects with the leader of the Consistent Core and sends a request to create a lease. The register lease request is replicated and handled similar to other requests in Consistent Core. The request is complete only when the *High-Water Mark* reaches the log index of the request entry in the replicated log.

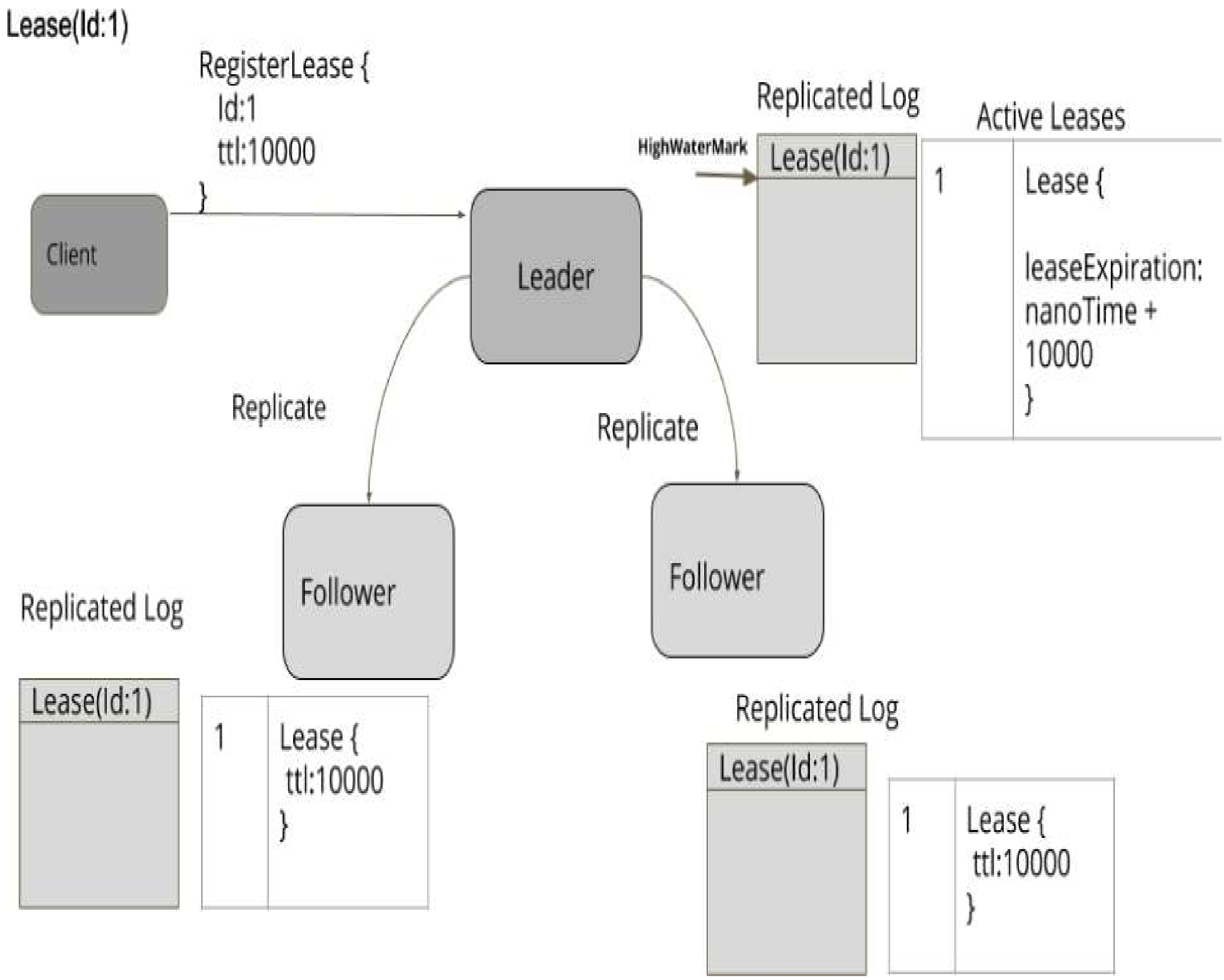
class ReplicatedKVStore…

```
private ConcurrentHashMap<String, Lease> leases = new ConcurrentHas
@Override
public CompletableFuture<Response> registerLease(String name, long
  if (leaseExists(name)) {
     return CompletableFuture
          .completedFuture(
              Response.error(DUPLICATE_LEASE_ERROR,
               "Lease with name " + name + " already exists"));
  }
  return log.propose(new RegisterLeaseCommand(name, ttl));
}

private boolean leaseExists(String name) {
  return leases.containsKey(name);
}
```

An important thing to note is where to validate for duplicate lease registration. Checking it before proposing the request is not enough, as there can be multiple in-flight requests. So the server also checks for duplicates when the lease is registered after successful replication.

class LeaderLeaseTracker…

```
private Map<String, Lease> leases;
@Override
public void addLease(String name, long ttl) throws DuplicateLeaseEx
  if (leases.get(name) != null) {
     throw new DuplicateLeaseException(name);
  }
  Lease lease = new Lease(name, ttl, clock.nanoTime());
  leases.put(name, lease);
}
```



Like any heartbeating mechanism, there is an assumption here that the server's monotonic clock is not faster than the client's monotonic clock. To take care of any possible rate difference, clients need to be conservative and send multiple heartbeats to the server within the timeout interval.

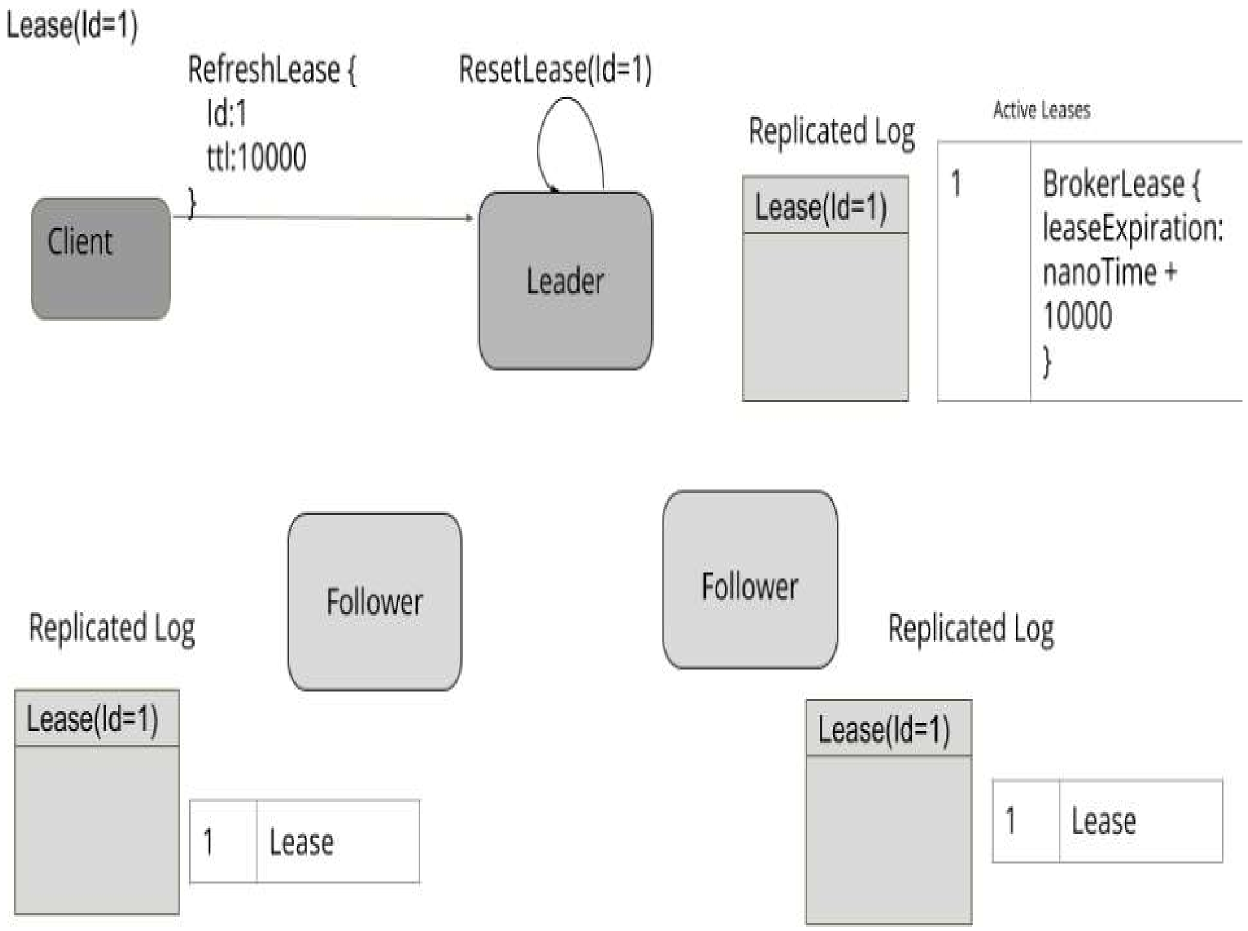
For example, Zookeeper [bib-zookeeper] has a default session timeout of 10 seconds, and uses 1/3 of the session timeout to send heartbeats. Apache Kafka, in its new architecture [bib-kip-631-configurations] uses 18 seconds as lease expiration time, and heartbeat is sent every 3 seconds.

The node responsible for the lease connects to the leader and refreshes the lease before it expires. As discussed in HeartBeat, it needs to consider the network round trip time to decide on the ‘time to live’ value, and send refresh requests before the lease expires. The node can send refresh requests multiple times within the ‘time to live’ time interval, to ensure that lease is refreshed in case of any issues. But the node also needs to make sure that too many refresh requests are not sent. It’s reasonable to send a request after about half of the lease time is elapsed. This results in up to two refresh requests within the lease time. The client node tracks the time with its own monotonic clock.

class LeaderLeaseTracker…

```
@Override
public void refreshLease(String name) {
  Lease lease = leases.get(name);
  lease.refresh(clock.nanoTime());
}
```

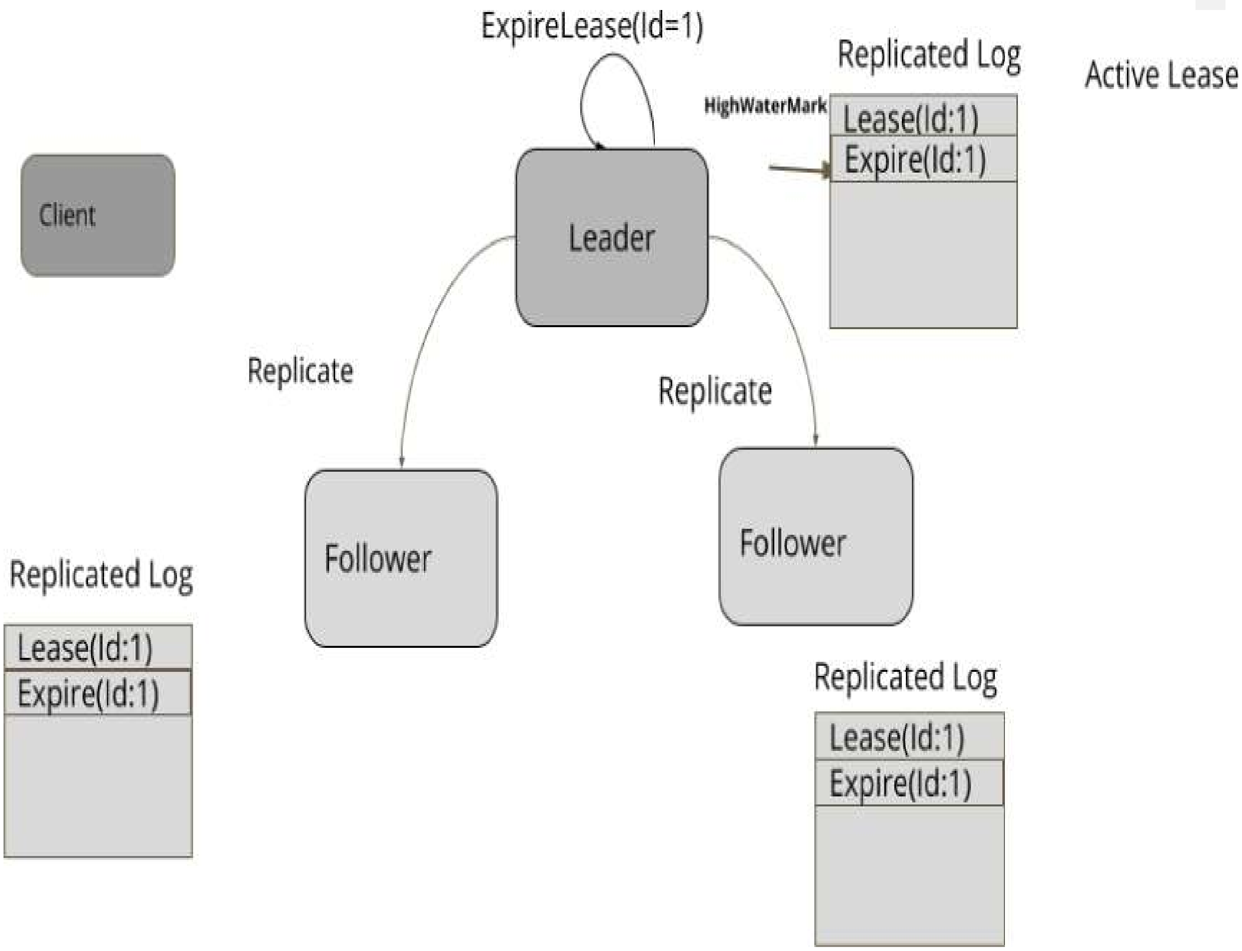
Refresh requests are sent only to the leader of the Consistent Core, because only the leader is responsible for deciding when the lease expires.



When the lease expires, it is removed from the leader. It’s also critical for this information to be committed to the Consistent Core. So the leader sends a request to expire the lease, which is handled like other requests in Consistent Core. Once the High-Water Mark reaches the proposed expire lease request, it’s removed from all the followers.

class LeaderLeaseTracker…

```
public void expireLease(String name) {
  getLogger().info("Expiring lease " + name);
  Lease removedLease = leases.remove(name);
  removeAttachedKeys(removedLease);
}

@Override
public Lease getLeaseDetails(String name) {
  return leases.get(name);
}
```



Attaching the lease to keys in the key value storage

> Zookeeper [bib-zookeeper] has a concept of sessions and ephemeral nodes. The sessions are implemented with the similar mechanism explained in this pattern. Ephemeral nodes are attached to the session. Once the session expires, all the ephemeral nodes are removed from the storage.

A cluster needs to know if one of its nodes fails. It can do that by having the node take a lease from the Consistent Core, and then attach it to a self-identifying key that it stores within the Consistent Core. If the cluster node is running, it should renew the lease at regular intervals. Should the lease expire, the associated keys are removed. When the key is removed, an event indicating the node failure is sent to the interested cluster node as discussed in the *State Watch* pattern.

The cluster node using the Consistent Core, creates a lease by making a network call, like following:

```
consistentCoreClient.registerLease("server1Lease", Duration.ofSecon
```

It can then attach this lease to the self-identifying key it stores in the Consistent Core.

```
consistentCoreClient.setValue("/servers/1", "{address:192.168.199.1
```

When the Consistent Core receives the message to save the key in its key-value store, it also attaches the key to the specified lease.

class ReplicatedKVStore…

```
private ConcurrentHashMap<String, Lease> leases = new ConcurrentHas
```

class ReplicatedKVStore…

```
private Response applySetValueCommand(Long walEntryId, SetValueComm
  getLogger().info("Setting key value " + setValueCommand);
  if (setValueCommand.hasLease()) {
     Lease lease = leases.get(setValueCommand.getAttachedLease());
     if (lease == null) {
        //The lease to attach is not available with the Consistent
        return Response.error(Errors.NO_LEASE_ERROR,
            "No lease exists with name "
                + setValueCommand.getAttachedLease(), 0);
    }

    lease.attachKey(setValueCommand.getKey());
  }
  kv.put(setValueCommand.getKey(), new StoredValue(setValueCommand.
```

Once the lease expires, the Consistent Core also removes the attached keys from its key-value store.

class LeaderLeaseTracker…

```
public void expireLease(String name) {
  getLogger().info("Expiring lease " + name);
  Lease removedLease = leases.remove(name);
  removeAttachedKeys(removedLease);
}
```

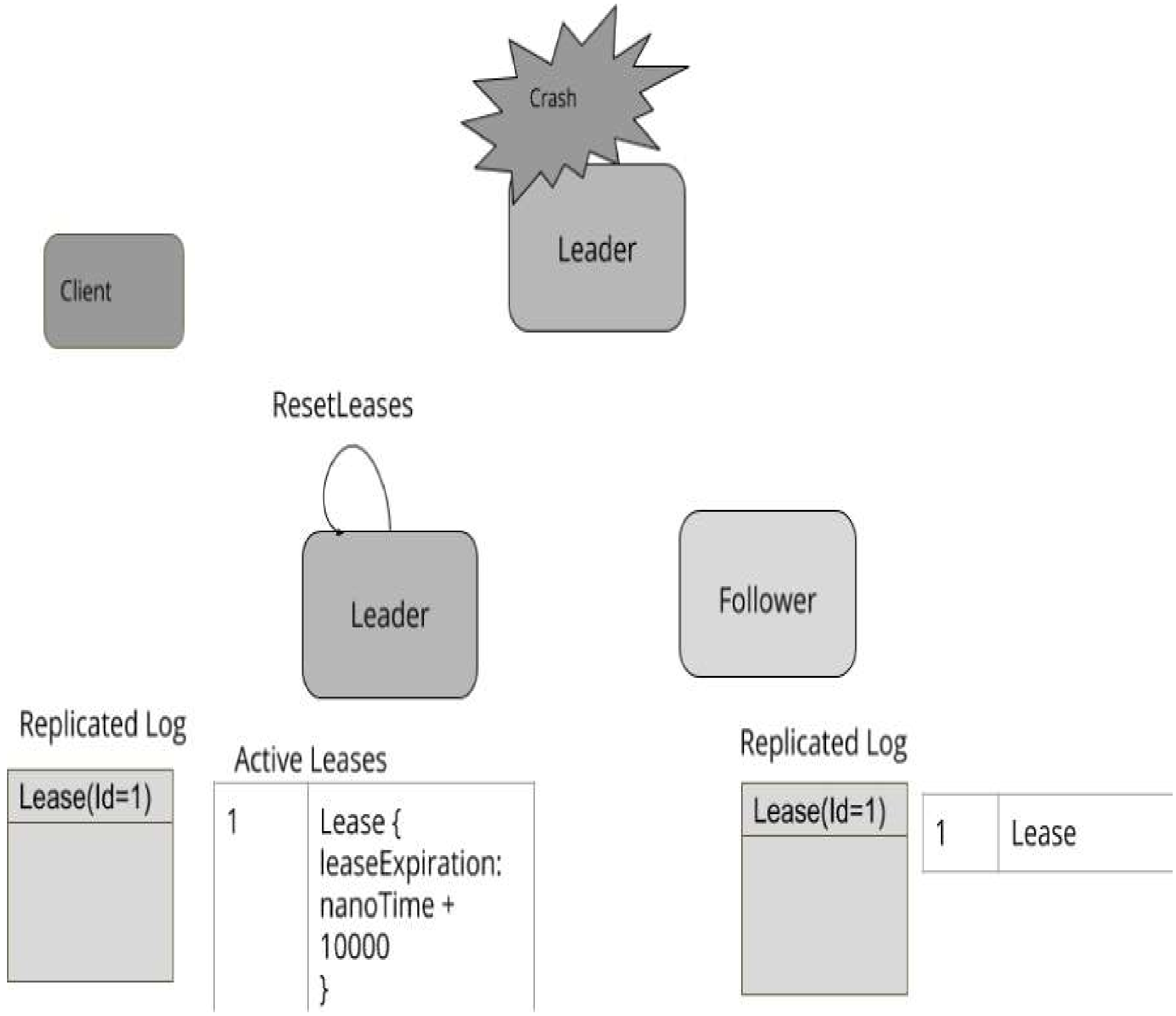
```
@Override
public Lease getLeaseDetails(String name) {
  return leases.get(name);
}

private void removeAttachedKeys(Lease removedLease) {
  if (removedLease == null) {
     return;
  }
  List<String> attachedKeys = removedLease.getAttachedKeys();
  for (String attachedKey : attachedKeys) {
     getLogger().trace("Removing " + attachedKey + " with lease " +
     kvStore.remove(attachedKey);
  }
}
```

Handling leader failure

When the existing leader fails, a new leader for *Consistent Core* is elected. Once elected, the new leader starts tracking the leases.

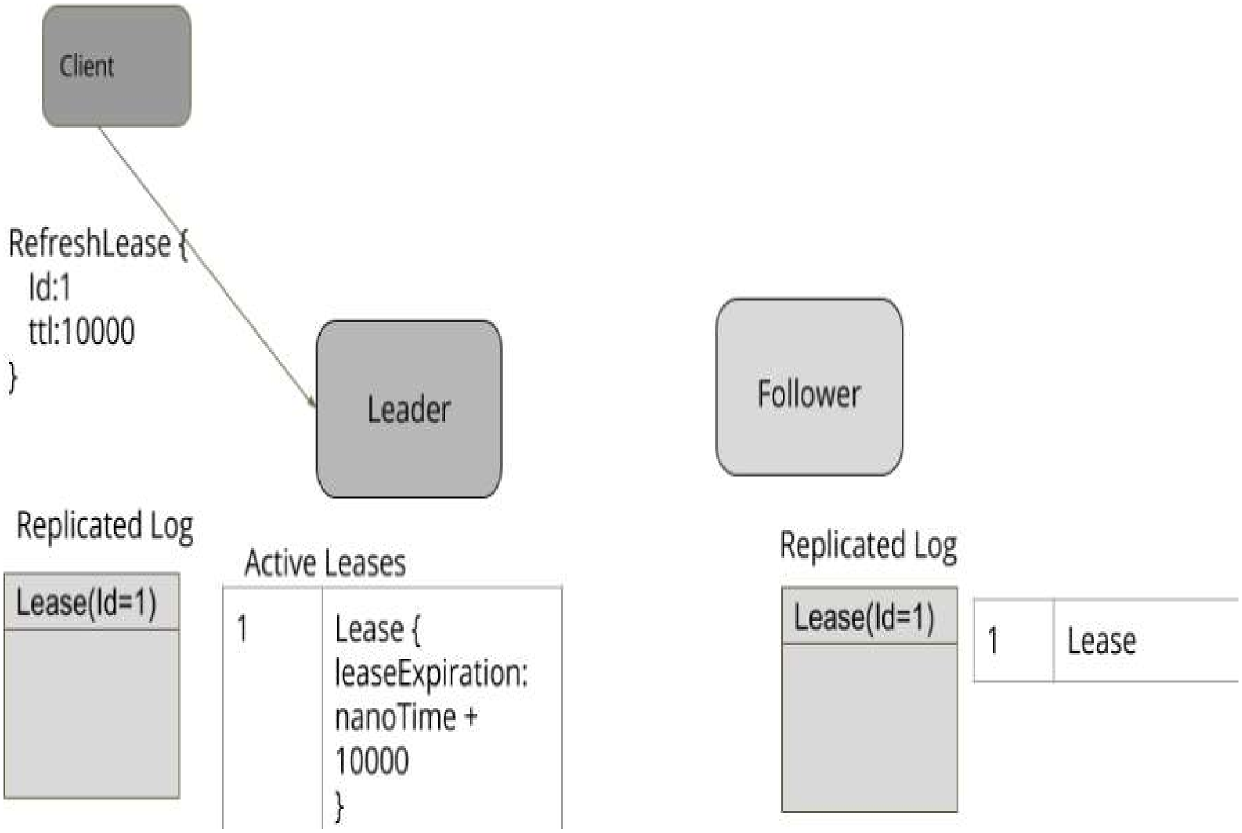
The new leader refreshes all the leases it knows about. Note that the leases which were about to expire on the old leader get extended by the ‘time to live’ value. This is not a problem, as it gives the chance for the client to reconnect with the new leader and continue the lease.

```
private void refreshLeases() {
  long now = clock.nanoTime();
  this.kvStore.getLeases().values().forEach(l -> {
    l.refresh(now);
  });
}
```



1	Lease {
	leaseExpiration:
	nanoTime +
	10000
	}

1	Lease



Examples

Google's [chubby] [bib-chubby] service implements the time-bound lease mechanism in similar way

Zookeeper [bib-zookeeper] sessions are managed with similar mechanisms as that of replicated leases.

The KIP-631 [bib-kip-631] in Kafka proposes use of time-bound leases to manage group membership and failure detection of Kafka brokers.

etcd [bib-etcd] provides time bound lease facility, which is used by clients to coordinate their activities as well as for group membership and failure detection.

dhcp [bib-dhcp] protocol allows connecting devices to lease an IP address. The failover protocol [bib-dhcp-failover] with multiple DHCP servers works similar to the implementation explained here.

Chapter 27. State Watch

Notify clients when specific values change on the server

Problem

Clients are interested in changes to the specific values on the server. It’s difficult for clients to structure their logic if they need to poll the server continuously to look for changes. If clients open too many connections to the server for watching changes, it can overwhelm the server.

Solution

Allow clients to register their interest with the server for specific state changes. The server notifies the interested clients when state changes happen. The client maintains a *Single Socket Channel* with the server. The server sends state change notifications on this channel. Clients might be interested in multiple values, but maintaining a connection per watch can overwhelm the server. So clients can use *Request Pipeline*.

Considering a simple key value store example used in *Consistent Core*: a client can be interested when a value changes for a particular key or a key is removed. There are two parts to the implementation, a client side implementation and a server side implementation.

Client side implementation

The client accepts the key and the function to be invoked when it gets watch events from the server. The client stores the function object for later invocation. It then sends the request to register the watch to the server.

```
ConcurrentHashMap<String, Consumer<WatchEvent>> watches = new Conc

public void watch(String key, Consumer<WatchEvent> consumer) {
  watches.put(key, consumer);
  sendWatchRequest(key);
}

private void sendWatchRequest(String key) {
  requestSendingQueue.submit(new RequestOrResponse(RequestId.Watch
       JsonSerDes.serialize(new WatchRequest(key)),
       correlationId.getAndIncrement()));
}
```

When a watch event is received on the connection, a corresponding consumer is invoked

```
this.pipelinedConnection = new PipelinedConnection(address, reques
  logger.info("Received response on the pipelined connection " + r
  if (r.getRequestId() == RequestId.WatchRequest.getId()) {
     WatchEvent watchEvent = JsonSerDes.deserialize(r.getMessageBo
     Consumer<WatchEvent> watchEventConsumer = getConsumer(watchEv
     watchEventConsumer.accept(watchEvent);
     lastWatchedEventIndex = watchEvent.getIndex(); //capture last
  }
  completeRequestFutures(r);
});
```

Server side implementation

When the server receives a watch registration request, it keeps the mapping of the pipelined connection on which the request is received, and the keys.

```
private Map<String, ClientConnection> watches = new HashMap<>();
private Map<ClientConnection, List<String>> connection2WatchKeys =

public void watch(String key, ClientConnection clientConnection) {
  logger.info("Setting watch for " + key);
  addWatch(key, clientConnection);
}

private synchronized void addWatch(String key, ClientConnection cl
  mapWatchKey2Connection(key, clientConnection);
  watches.put(key, clientConnection);
}

private void mapWatchKey2Connection(String key, ClientConnection c
  List<String> keys = connection2WatchKeys.get(clientConnection);
  if (keys == null) {
     keys = new ArrayList<>();
     connection2WatchKeys.put(clientConnection, keys);
  }
  keys.add(key);
}
```

The ClientConnection wraps the socket connection to the client. It has the following structure. This structure remains the same for both, the blocking-IO based server and Non-blocking-IO-based server.

```
public interface ClientConnection {
  void write(RequestOrResponse response);
  void close();
}
```

There can be multiple watches registered on a single connection. So it is important to store the mapping of connections to the list of watch keys. It is needed when the client connection is closed, to remove all the associated watches as following:

```
public void close(ClientConnection connection) {
  removeWatches(connection);
}
private synchronized void removeWatches(ClientConnection clientCon
  List<String> watchedKeys = connection2WatchKeys.remove(clientCon
  if (watchedKeys == null) {
     return;
  }
  for (String key : watchedKeys) {
     watches.remove(key);
  }
}
```

Using Reactive Streams [bib-reactive-streams]

The example here shows writing the events directly to the pipelined connection. Some type of backpressure at the application level is useful to have. If there are a lot many events getting generated, it's important to control the rate at which they can be sent. Keeping producers and consumers of the events in sync is an important consideration. This [bib-etcd-watch-channel-issue] issue in etcd is an example of how these considerations matter in production.

[reactive-streams] [bib-reactive-streams] API makes it easier to write code with backpressure as a first class concept. Protocols like rsocket [bib-rsocket] provide a structured way to implement this.

When the specific events like setting a value for key happen on the server, the server notifies all the registered clients by constructing a relevant WatchEvent

```
private synchronized void notifyWatchers(SetValueCommand setValueC
  logger.info("Looking for watches for " + setValueCommand.getKey(
  if (!hasWatchesFor(setValueCommand.getKey())) {
     return;
  }
  String watchedKey = setValueCommand.getKey();
  WatchEvent watchEvent = new WatchEvent(watchedKey,
                 setValueCommand.getValue(),
                 EventType.KEY_ADDED, entryId);
  notify(watchEvent, watchedKey);
}

private void notify(WatchEvent watchEvent, String watchedKey) {
  List<ClientConnection> watches = getAllWatchersFor(watchedKey);
  for (ClientConnection pipelinedClientConnection : watches) {
    try {
       getLogger().info("Notifying watcher of event "
           + watchEvent +
           " from "
           + log.getServerId());
       pipelinedClientConnection
            .write(new RequestOrResponse(RequestId.WatchRequest.ge
                 JsonSerDes.serialize(watchEvent)));
  } catch (NetworkException e) {
    removeWatches(pipelinedClientConnection); //remove watch if ne
  }
  }
}
```

One of the critical things to note is that the state related to watches can be accessed concurrently from client request handling code and from the client connection handling code to close the connection. So all the methods accessing watch state needs to be protected by locks.

Watches on hierarchical storage

Consistent Core mostly supports hierarchical storage. The watches can be set on the parent nodes or prefix of a key. Any changes to the child node triggers the watches set on the parent node. For each event, the Consistent Core walks the path to check if there are watches setup on the parent path and send events to all those watches.

```
List<ClientConnection> getAllWatchersFor(String key) {
  List<ClientConnection> affectedWatches = new ArrayList<>();
  String[] paths = key.split("/");
  String currentPath = paths[0];
  addWatch(currentPath, affectedWatches);
  for (int i = 1; i < paths.length; i++) {
    currentPath = currentPath + "/" + paths[i];
    addWatch(currentPath, affectedWatches);
  }
  return affectedWatches;
}

private void addWatch(String currentPath, List<ClientConnection> a
  ClientConnection clientConnection = watches.get(currentPath);
  if (clientConnection != null) {
    affectedWatches.add(clientConnection);
  }
}
```

This allows a watch to be set up on a key prefix like "servers". Any key created with this prefix like "servers/1", "servers/2" will trigger this watch.

Because the mapping of the function to be invoked is stored with the key prefix, it’s important to walk the hierarchy to find the function to be invoked for the received event on the client side as well. An alternative can be to send the path for which the event triggered along with the event, so that the client knows which watch caused the event to be sent.

Handling Connection Failures

The connection between client and server can fail at any time. For some use cases this is problematic as the client might miss certain events when it’s disconnected. For example, a cluster controller might be interested in knowing if some nodes have failed, which is indicated by events for removal of some keys. The client needs to tell the server about the last event it received. The client sends the last received event number when it resets the watch again. The server is expected to send all the events it has recorded from that event number onwards.

In the *Consistent Core* client, it can be done when the client re-establishes the connection to the leader.

Pull based design in Kafka

In a typical design for watches, the server pushes watch events to clients. Kafka [bib-kafka] follows end-to-end pull-based design. In its new architecture [bib-kip-500], the Kafka brokers are going to periodically pull metadata log from a Controller Quorum [bib-kip-631] (which itself is an example of Consistent Core). The offset-based pull mechanism allows clients to read events from the last known offset, like any other Kafka consumer, avoiding loss of events.

```
private void connectToLeader(List<InetAddressAndPort> servers) {
  while (isDisconnected()) {
    logger.info("Trying to connect to next server");
    waitForPossibleLeaderElection();
    establishConnectionToLeader(servers);
  }
  setWatchesOnNewLeader();
}

private void setWatchesOnNewLeader() {
  for (String watchKey : watches.keySet()) {
     sendWatchResetRequest(watchKey);
  }
}

private void sendWatchResetRequest(String key) {
  pipelinedConnection.send(new RequestOrResponse(RequestId.SetWatc
       JsonSerDes.serialize(new SetWatchRequest(key, lastWatchedEv
}
```

The server numbers every event that occurs. For example, if the server is the Consistent Core, it stores all the state changes in a strict order and every change is numbered with the log index as discussed in *Write-Ahead Log*, It is then possible for clients to ask for events starting from the specific index.

Deriving events from the key value store

The events can be generated looking at the current state of the key value store, if it also numbers every change that happens and stores that number with each value.

When the client re-establishes the connection to the server, it can set the watches again also sending the last seen change number. The server can then compare it with the one stored with the value and if it’s more than the one client sent, it can resend the events to the client. Deriving events from the key value store can be a bit awkward as events need to be guessed. It might miss some events. - for instance, If a key is created and then deleted - while the client was disconnected, the create event will be missed.

```
private synchronized void eventsFromStoreState(String key, long st
  List<StoredValue> values = getValuesForKeyPrefix(key);
  for (StoredValue value : values) {
    if (values == null) {
       //the key was probably deleted send deleted event
       notify(new WatchEvent(key, EventType.KEY_DELETED), key);
    } else if (value.index > stateChangesSince) {
       //the key/value was created/updated after the last event cl
       notify(new WatchEvent(key, value.getValue(), EventType.KEY_
    }
  }
}
```

Zookeeper [bib-zookeeper] uses this approach. The watches in zookeeper are also one-time triggers by default. Once the event is triggered, clients need to set the watch again if they want to receive further events. Some events can be missed, before the watch is set again, so clients need to ensure they read the latest state, so that they don’t miss any updates.

Storing Event History

It’s easier to keep a history of past events and reply to clients from the event history. The problem with this approach is that the event history needs to be limited, say to 1,000 events. If the client is disconnected for a longer duration, it might miss on events which are beyond the 1,000 events window.

A simple implementation using google guava’s EvictingQueue is as following:

```
public class EventHistory implements Logging {
  Queue<WatchEvent> events = EvictingQueue.create(1000);
  public void addEvent(WatchEvent e) {
    getLogger().info("Adding " + e);
    events.add(e);
  }

  public List<WatchEvent> getEvents(String key, Long stateChangesS
    return this.events.stream()
          .filter(e -> e.getIndex() > stateChangesSince && e.getKey
          .collect(Collectors.toList());
  }
}
```

When the client re-establishes the connection and resets watches, the events can be sent from history.

```
private void sendEventsFromHistory(String key, long stateChangesSi
  List<WatchEvent> events = eventHistory.getEvents(key, stateChang
  for (WatchEvent event : events) {
    notify(event, event.getKey());
  }
}
```

Using multi-version storage

To keep track of all the changes, it is possible to use multi-version storage. It keeps track of all the versions for every key, and can easily get all the changes from the version asked for.

etcd [bib-etcd] version 3 onwards uses this approach

Examples

Zookeeper [bib-zookeeper] has the ability to set up watches on nodes. This is used by products like Kafka [bib-kafka] for group membership and failure detection of cluster members.

etcd [bib-etcd] has watch implementation which is heavily used by Kubernetes [bib-kubernetes] for its resource watch [bib-kubernetes-api] implementation.

Chapter 28. Gossip Dissemination

Use random selection of nodes to pass on information to ensure it reaches all the nodes in the cluster without flooding the network

Problem

In a cluster of nodes, each node needs to pass metadata information it has, to all the other nodes in the cluster, without depending on a shared storage. In a large cluster, if all servers communicate with all the other servers, a lot of network bandwidth can be consumed. Information should reach all the nodes even when some network links are experiencing issues.

Solution

Cluster nodes use gossip style communication to propagate state updates. Each node selects a random node to pass the information it has. This is done at a regular interval, say every 1 second. Each time, a random node is selected to pass on the information.

Epidemics, Rumours and Computer Communication

Currently we are all experiencing how quickly a pandemic like Covid19 spreads across the entire globe. There are mathematical properties of epidemics which describe why they spread so fast. The mathematical branch of epidemiology [bib-epidemiology] studies how an epidemic or rumours spread in a society. Gossip Dissemination is based on the mathematical models from epidemiology. The key characteristics of an epidemic or rumours is that they spread very fast even if each person comes into contact with only a few individuals at random. An entire population can become infected even with very few total interactions. More specifically, if n is the total number of people in a given population, it takes interactions proportional to log(n) per individual. As discussed by Professor Indranil Gupta in his Gossip Analysis [bib-gossip-analysis], log(n) can be almost treated as a constant.

This property of epidemic spread is very useful to spread the information across a set of processes. Even if a given process communicates with only a few processes at random, in a very few communication rounds, all the nodes in the cluster will have the same information. Hashicorp has a very nice convergence simulator [bib-serf-convergence-simulator] to demonstrate how quickly the information spreads the entire cluster, even with some network loss and node failures.

In large clusters, the following things need to be considered:

- Put a fixed limit on the number of messages generated per server
- The messages should not consume a lot of network bandwidth. There should be an upper bound of say a few hundred Kbs, making sure that the applications' data transfer is not impacted by too many messages across the cluster.
- The metadata propagation should tolerate network and a few server failures. It should reach all the cluster nodes even if a few network links are down, or a few servers have failed.

As discussed in the sidebar, Gossip-style communication fulfills all these requirements.

Each cluster nodes stores the metadata as a list of key value pairs associated with each node in the cluster as following:

class Gossip…

```
Map<NodeId, NodeState> clusterMetadata = new HashMap<>();
```

class NodeState…

```
Map<String, VersionedValue> values = new HashMap<>();
```

At startup, each cluster node adds the metadata about itself, which needs to be propagated to other nodes. An example of metadata can be the IP address and port the node listens on, the partitions it's responsible for, etc. The Gossip instance needs to know about at least one other node to start the gossip communication. The well known cluster node, which is used to initialize the Gossip instance is called as a seed node or an introducer. Any node can act as an introducer.

class Gossip…

```
public Gossip(InetAddressAndPort listenAddress,
        List<InetAddressAndPort> seedNodes,
        String nodeId) throws IOException {
  this.listenAddress = listenAddress;
  //filter this node itself in case its part of the seed nodes
  this.seedNodes = removeSelfAddress(seedNodes);
  this.nodeId = new NodeId(nodeId);
  addLocalState(GossipKeys.ADDRESS, listenAddress.toString());

  this.socketServer = new NIOSocketListener(newGossipRequestConsum
}

private void addLocalState(String key, String value) {
  NodeState nodeState = clusterMetadata.get(listenAddress);
  if (nodeState == null) {
     nodeState = new NodeState();
     clusterMetadata.put(nodeId, nodeState);
  }
  nodeState.add(key, new VersionedValue(value, incremenetVersion()
}
```

Each cluster node schedules a job to transmit the metadata it has to other nodes at regular intervals.

class Gossip…

```
private ScheduledThreadPoolExecutor gossipExecutor = new Scheduled'
private long gossipIntervalMs = 1000;
private ScheduledFuture<?> taskFuture;
public void start() {
  socketServer.start();
  taskFuture = gossipExecutor.scheduleAtFixedRate(()-> doGossip(),
         gossipIntervalMs,
         gossipIntervalMs,
         TimeUnit.MILLISECONDS);
}
```

When the scheduled task is invoked, it picks up a small set of random nodes from the list of servers from the metadata map. A small constant number, defined as Gossip fanout, determines how many nodes to pick up as gossip targets. If nothing is known yet, it picks up a random seed node and sends the metadata map it has to that node.

class Gossip…

```
public void doGossip() {
  List<InetAddressAndPort> knownClusterNodes = liveNodes();
  if (knownClusterNodes.isEmpty()) {
     sendGossip(seedNodes, gossipFanout);
  } else {
    sendGossip(knownClusterNodes, gossipFanout);
  }
}

private List<InetAddressAndPort> liveNodes() {
  Set<InetAddressAndPort> nodes
       = clusterMetadata.values()
       .stream()
```

```
        .map(n -> InetAddressAndPort.parse(n.get(GossipKeys.ADDRESS
        .collect(Collectors.toSet());
  return removeSelfAddress(nodes);
}
```

Using UDP or TCP

Gossip communication assumes unreliable networks, so it can use UDP as a transport mechanism. But cluster nodes generally need some guarantee of quick convergence of state, and therefore use TCP-based transport to exchange the gossip state.

```
private void sendGossip(List<InetAddressAndPort> knownClusterNodes
  if (knownClusterNodes.isEmpty()) {
     return;
  }

  for (int i = 0; i < gossipFanout; i++) {
    InetAddressAndPort nodeAddress = pickRandomNode(knownClusterNo
    sendGossipTo(nodeAddress);
  }
}

private void sendGossipTo(InetAddressAndPort nodeAddress) {
  try {
     getLogger().info("Sending gossip state to " + nodeAddress);
     SocketClient<RequestOrResponse> socketClient = new SocketClie
     GossipStateMessage gossipStateMessage
         = new GossipStateMessage(this.nodeId, this.clusterMetadat
     RequestOrResponse request
         = createGossipStateRequest(gossipStateMessage);
     RequestOrResponse response = socketClient.blockingSend(reques
     GossipStateMessage responseState = deserialize(response);
     merge(responseState.getNodeStates());
   } catch (IOException e) {
      getLogger().error("IO error while sending gossip state to " +
    }
}

private RequestOrResponse createGossipStateRequest(GossipStateMess
  return new RequestOrResponse(RequestId.PushPullGossipState.getId
       JsonSerDes.serialize(gossipStateMessage), correlationId++);
}
```

The cluster node receiving the gossip message inspects the metadata it has and finds three things.

- The values which are in the incoming message but not available in this node's state map
- The values which it has but the incoming Gossip message does not have
- The higher version value is chosen when the node has the values present in the incoming message

It then adds the missing values to its own state map. Whatever values were missing from the incoming message, are returned as a response.

The cluster node sending the Gossip message adds the values it gets from the gossip response to its own state.

class Gossip…

```
private void handleGossipRequest(org.distrib.patterns.common.Messa
  GossipStateMessage gossipStateMessage = deserialize(request.getR
  Map<NodeId, NodeState> gossipedState = gossipStateMessage.getNod
  getLogger().info("Merging state from " + request.getClientConnec
  merge(gossipedState);

  Map<NodeId, NodeState> diff = delta(this.clusterMetadata, gossip
  GossipStateMessage diffResponse = new GossipStateMessage(this.no
  getLogger().info("Sending diff response " + diff);
  request.getClientConnection().write(new RequestOrResponse(Reques
            JsonSerDes.serialize(diffResponse),
            request.getRequest().getCorrelationId()));
}

public Map<NodeId, NodeState> delta(Map<NodeId, NodeState> fromMap
  Map<NodeId, NodeState> delta = new HashMap<>();
  for (NodeId key : fromMap.keySet()) {
    if (!toMap.containsKey(key)) {
       delta.put(key, fromMap.get(key));
       continue;
    }
    NodeState fromStates = fromMap.get(key);
    NodeState toStates = toMap.get(key);
    NodeState diffStates = fromStates.diff(toStates);
    if (!diffStates.isEmpty()) {
       delta.put(key, diffStates);
    }
  }
  return delta;
}

public void merge(Map<NodeId, NodeState> otherState) {
  Map<NodeId, NodeState> diff = delta(otherState, this.clusterMeta
  for (NodeId diffKey : diff.keySet()) {
    if(!this.clusterMetadata.containsKey(diffKey)) {
       this.clusterMetadata.put(diffKey, diff.get(diffKey));
    } else {
      NodeState stateMap = this.clusterMetadata.get(diffKey);
      stateMap.putAll(diff.get(diffKey));
    }
  }
}
```

This process happens every one second at each cluster node, each time selecting a different node to exchange the state.

Avoiding unnecessary state exchange

The above code example shows that the complete state of the node is sent in the Gossip message. This is fine for a newly joined node, but once the state is up to date, it's unnecessary to send the complete state. The cluster node just needs to send the state changes since the last gossip. For achieving this, each node maintains a version number which is incremented every time a new metadata entry is added locally.

class Gossip…

```
private int gossipStateVersion = 1;


private int incremenetVersion() {
  return gossipStateVersion++;
}
```

Each value in the cluster metadata is maintained with a version number. This is an example of pattern *Versioned Value*.

class VersionedValue…

```
long version;
String value;

public VersionedValue(String value, long version) {
  this.version = version;
  this.value = value;
}

public long getVersion() {
  return version;
}

public String getValue() {
  return value;
}
```

Each Gossip cycle can then exchange states from a specific version.

class Gossip…

```
private void sendKnownVersions(InetAddressAndPort gossipTo) throws
  Map<NodeId, Long> maxKnownNodeVersions = getMaxKnownNodeVersions
  RequestOrResponse knownVersionRequest = new RequestOrResponse(Re
      JsonSerDes.serialize(new GossipStateVersions(maxKnownNodeVer
  SocketClient<RequestOrResponse> socketClient = new SocketClient(
  socketClient.blockingSend(knownVersionRequest);
}

private Map<NodeId, Long> getMaxKnownNodeVersions() {
  return clusterMetadata.entrySet()
       .stream()
       .collect(Collectors.toMap(e -> e.getKey(), e -> e.getValue(
}
```

class NodeState…

```
public long maxVersion() {
  return values.values().stream().map(v -> v.getVersion()).max(Com
}
```

The receiving node can then send the values only if the versions are greater than the ones in the request.

class Gossip…

```
Map<NodeId, NodeState> getMissingAndNodeStatesHigherThan(Map<NodeI
  Map<NodeId, NodeState> delta = new HashMap<>();
  delta.putAll(higherVersionedNodeStates(nodeMaxVersions));
  delta.putAll(missingNodeStates(nodeMaxVersions));
  return delta;
}

private Map<NodeId, NodeState> missingNodeStates(Map<NodeId, Long>
  Map<NodeId, NodeState> delta = new HashMap<>();
  List<NodeId> missingKeys = clusterMetadata.keySet().stream().fil
  for (NodeId missingKey : missingKeys) {
    delta.put(missingKey, clusterMetadata.get(missingKey));
  }
  return delta;
}

private Map<NodeId, NodeState> higherVersionedNodeStates(Map<NodeI
  Map<NodeId, NodeState> delta = new HashMap<>();
  Set<NodeId> keySet = nodeMaxVersions.keySet();
  for (NodeId node : keySet) {
    Long maxVersion = nodeMaxVersions.get(node);
  NodeState nodeState = clusterMetadata.get(node);
  if (nodeState == null) {
     continue;
  }
  NodeState deltaState = nodeState.statesGreaterThan(maxVersion);
  if (!deltaState.isEmpty()) {
     delta.put(node, deltaState);
  }
 }
 return delta;
}
```

Gossip implementation in Cassandra [bib-cassandra] optimizes state exchange with a three-way handshake, where the node receiving the gossip message also sends the versions it needs from the sender, along with the metadata it returns. The sender can then immediately respond with the requested metadata. This avoids an extra message that otherwise would have been required.

Gossip protocol used in CockroachDB [bib-cockroachdb] maintains state for each connected node. For each connection, it maintains the last version sent to that node, and the version received from that node. This is so that it can send ‘state since the last sent version’ and ask for ‘state from the last received version’.

Some other efficient alternatives can be used as well, sending a hash of the entire Map and if the hash is the same, then doing nothing.

Criteria for node selection to Gossip

Cluster nodes randomly select the node to send the Gossip message. An example implementation in Java can use java.util.Random as following:

class Gossip…

```
private Random random = new Random();
private InetAddressAndPort pickRandomNode(List<InetAddressAndPort>
  int randomNodeIndex = random.nextInt(knownClusterNodes.size());
  InetAddressAndPort gossipTo = knownClusterNodes.get(randomNodeIn
  return gossipTo;
}
```

There can be other considerations such as the node that is least contacted with. For example, Gossip protocol in Cockroachdb [bib-cockroachdb-design] selects nodes this way.

There are network-topology-aware [bib-wan-gossip] ways of Gossip target selection that exist as well.

Any of these can be implemented modularly inside the pickRandomNode() method.

Group Membership and Failure Detection

Eventual Consistency

Information exchange with Gossip protocols is eventually consistent by nature. Even if its Gossip state converges very fast, there will be some delay before a new node is recognized by the entire cluster or a node failure is detected. The implementations using Gossip protocol for information exchange, need to tolerate eventual consistency.

For operations which require strong consistency, *Consistent Core* needs to be used.

It’s a common practice to use both in the same cluster. For example, Consul [bib-consul] uses Gossip protocol for group membership and failure detection, but uses a Raft-based Consistent Core to store a strongly consistent service catalogue.

Maintaining the list of available nodes in the cluster is one of the most common usage of Gossip protocols. There are two approaches in use.

- [swim-gossip] [bib-swim-gossip] uses a separate probing component which continuously probes different nodes in the cluster to detect if they are available. If it detects that the node is alive or dead, that result is propagated to the entire cluster with Gossip communication. The prober randomly selects a node to send the Gossip message. If the receiving node detects that this is new information, it immediately sends the message to a randomly selected node. This way, the failure of a node or newly joined node in the cluster is quickly known to the entire cluster.
- The cluster node can periodically update its own state to reflect its heartbeat. This state is then propagated to the entire cluster through the gossip messages exchanged. Each cluster node can then check if it has received any update for a particular cluster node in a fixed amount of time or else mark that node as down. In this case, each cluster node independently determines if a node is up or down.

Handling node restarts

The versioned values does not work well if the node crashes or restarts, as all the in-memory state is lost. More importantly, the node can have different values for the same key. For example, the cluster node can start with a different IP address and port, or can start with a different configuration. *Generation Clock* can be used to mark generation with every value, so that when the metadata state is sent to a random cluster node, the receiving node can detect changes not just by the version number, but also with the generation.

It is useful to note that this mechanism is not necessary for the core Gossip protocol to work. But it’s implemented in practice to make sure that the state changes are tracked correctly.

Examples

Cassandra [bib-cassandra] uses Gossip protocol for the group membership and failure detection of cluster nodes. Metadata for each cluster node such as the tokens assigned to each cluster node, is also transmitted using Gossip protocol.

Consul [bib-consul] uses [swim-gossip] [bib-swim-gossip] protocol for group membership and failure detection of consul agents.

CockroachDB [bib-cockroachdb] uses Gossip protocol to propagate node metadata.

Blockchain implementations such as Hyperledger Fabric [bib-hyperledger-fabric-gossip] use Gossip protocol for group membership and sending ledger metadata.

Chapter 29. Emergent Leader

Order cluster nodes based on their age whithin the cluster to allow nodes to select a leader without running an explicit election.

Problem

Peer-to-peer systems treat each cluster node as equal; there is no strict leader. This means there is no explicit leader election process as happens in the *Leader and Followers* pattern. Sometimes the cluster also doesn’t want to depend on a separate *Consistent Core* to achieve better availability. However, there still needs to be one cluster node acting as cluster coordinator for tasks such as assigning data partitions to other cluster nodes and tracking when new cluster nodes join or fail and take corrective actions.

Solution

One of the common techniques used in peer-to-peer systems is to order cluster nodes according to their ‘age’. The oldest member of the cluster plays the role of the coordinator for the cluster. The coordinator is responsible for deciding on membership changes as well as making decisions such as where *Fixed Partitions* should be placed across cluster nodes.

To form the cluster, one of the cluster nodes acts as a seed node or an introducer node. All the cluster nodes join the cluster by contacting the seed node.

> Typically different discovery mechanisms are provided to find the node to join the cluster. For example JGroups [bib-jgroups] provides

> different discovery protocols [bib-jgroups-discovery-protocols]. Akka [bib-akka] provides several discovery mechanisms [bib-jgroups-discovery-protocols]. as well.

Every cluster node is configured with the seed node address. When a cluster node is started, it tries to contact the seed node to join the cluster.

class ClusterNode…

```
MembershipService membershipService;
public void start(Config config) {
  this.membershipService =  new MembershipService(config.getListenA
  membershipService.join(config.getSeedAddress());
}
```

The seed node could be any of the cluster nodes. It's configured with its own address as the seed node address and is the first node that is started. It immediately begins accepting requests. The age of the seed node is 1.

class MembershipService…

```
Membership membership;
public void join(InetAddressAndPort seedAddress) {
  int maxJoinAttempts = 5;
  for(int i = 0; i < maxJoinAttempts; i++){
    try {
       joinAttempt(seedAddress);
       return;
    } catch (Exception e) {
      logger.info("Join attempt " + i + "from " + selfAddress + " t
    }
  }
  throw new JoinFailedException("Unable to join the cluster after "
}

private void joinAttempt(InetAddressAndPort seedAddress) throws Exe
  if (selfAddress.equals(seedAddress)) {
     int membershipVersion = 1;
     int age = 1;
     updateMembership(new Membership(membershipVersion, Arrays.asLi
     start();
     return;
  }
  long id = this.messageId++;
  CompletableFuture<JoinResponse> future = new CompletableFuture<>(
  JoinRequest message = new JoinRequest(id, selfAddress);
  pendingRequests.put(id, future);
  network.send(seedAddress, message);

  JoinResponse joinResponse = Uninterruptibles.getUninterruptibly(f
  updateMembership(joinResponse.getMembership());
  start();
}

private void start() {
  heartBeatScheduler.start();
  failureDetector.start();
  startSplitBrainChecker();
  logger.info(selfAddress + " joined the cluster. Membership=" + me
}

private void updateMembership(Membership membership) {
   this.membership  = membership;
}
```

There can be more than one seed node. But seed nodes start accepting requests only after they themselves join the cluster. Also the cluster will be functional if the seed node is down, but no new nodes will be able to add to the cluster.

Non seed nodes then send the join request to the seed node. The seed node handles the join request by creating a new member record and assigning its age. It then updates its own membership list and sends messages to all the existing members with the new membership list. It then waits to make sure that the response is returned from every node, but will eventually return the join response even if the response is delayed.

class MembershipService…

```
public void handleJoinRequest(JoinRequest joinRequest) {
  handlePossibleRejoin(joinRequest);
  handleNewJoin(joinRequest);
}

private void handleNewJoin(JoinRequest joinRequest) {
  List<Member> existingMembers = membership.getLiveMembers();
  updateMembership(membership.addNewMember(joinRequest.from));
  ResultsCollector resultsCollector = broadcastMembershipUpdate(exi
  JoinResponse joinResponse = new JoinResponse(joinRequest.messageI
  resultsCollector.whenComplete((response, exception) -> {
    logger.info("Sending join response from " + selfAddress + " to
    network.send(joinRequest.from, joinResponse);
  });
}
```

class Membership…

```
public Membership addNewMember(InetAddressAndPort address) {
  var newMembership = new ArrayList<>(liveMembers);
  int age = yongestMemberAge() + 1;
  newMembership.add(new Member(address, age, MemberStatus.JOINED));
  return new Membership(version + 1, newMembership, failedMembers);
}

private int yongestMemberAge() {
  return liveMembers.stream().map(m -> m.age).max(Integer::compare)
}
```

If a node which was already part of the cluster is trying to rejoin after a crash, the failure detector state related to that member is cleared.

class MembershipService…

```
private void handlePossibleRejoin(JoinRequest joinRequest) {
  if (membership.isFailed(joinRequest.from)) {
     //member rejoining
     logger.info(joinRequest.from  + " rejoining the cluster. Remov
     membership.removeFromFailedList(joinRequest.from);
  }
}
```

It’s then added as a new member. Each member needs to be identified uniquely. It can be assigned a unique identifier at startup. This then provides a point of reference that makes it possible to check if it is an existing cluster node that is rejoining.

The membership class maintains the list of live members as well as failed members. The members are moved from live to failed list if they stop sending *HeartBeat* as explained in the failure detection section [#FailureDetection].

class Membership…

```
public class Membership {
  List<Member> liveMembers = new ArrayList<>();
  List<Member> failedMembers = new ArrayList<>();

  public boolean isFailed(InetAddressAndPort address) {
    return failedMembers.stream().anyMatch(m -> m.address.equals(ad
  }
```

Sending membership updates to all the existing members

Membership updates are sent to all the other nodes concurrently. The coordinator also needs to track whether all the members successfully received the updates.

A common technique is to send a one way request to all nodes and expect an acknowledgement message. The cluster nodes send acknowledgement messages to the coordinator to confirm receipt of the membership update. A ResultCollector object can track receipt of all the messages asynchronously, and is notified every time an acknowledgement is received for a membership update. It completes its future once the expected acknowledgement messages are received.

class MembershipService…

```
private ResultsCollector broadcastMembershipUpdate(List<Member> exi
  ResultsCollector resultsCollector = sendMembershipUpdateTo(existi
  resultsCollector.orTimeout(2, TimeUnit.SECONDS);
  return resultsCollector;
}

Map<Long, CompletableFuture> pendingRequests = new HashMap();
private ResultsCollector sendMembershipUpdateTo(List<Member> existi
  var otherMembers = otherMembers(existingMembers);
  ResultsCollector collector = new ResultsCollector(otherMembers.si
  if (otherMembers.size() == 0) {
     collector.complete();
     return collector;
  }
  for (Member m : otherMembers) {
    long id = this.messageId++;
    CompletableFuture<Message> future = new CompletableFuture();
    future.whenComplete((result, exception)->{
      if (exception == null){
      collector.ackReceived();
      }
    });
    pendingRequests.put(id, future);
    network.send(m.address, new UpdateMembershipRequest(id, selfAdd
  }
  return collector;
}
```

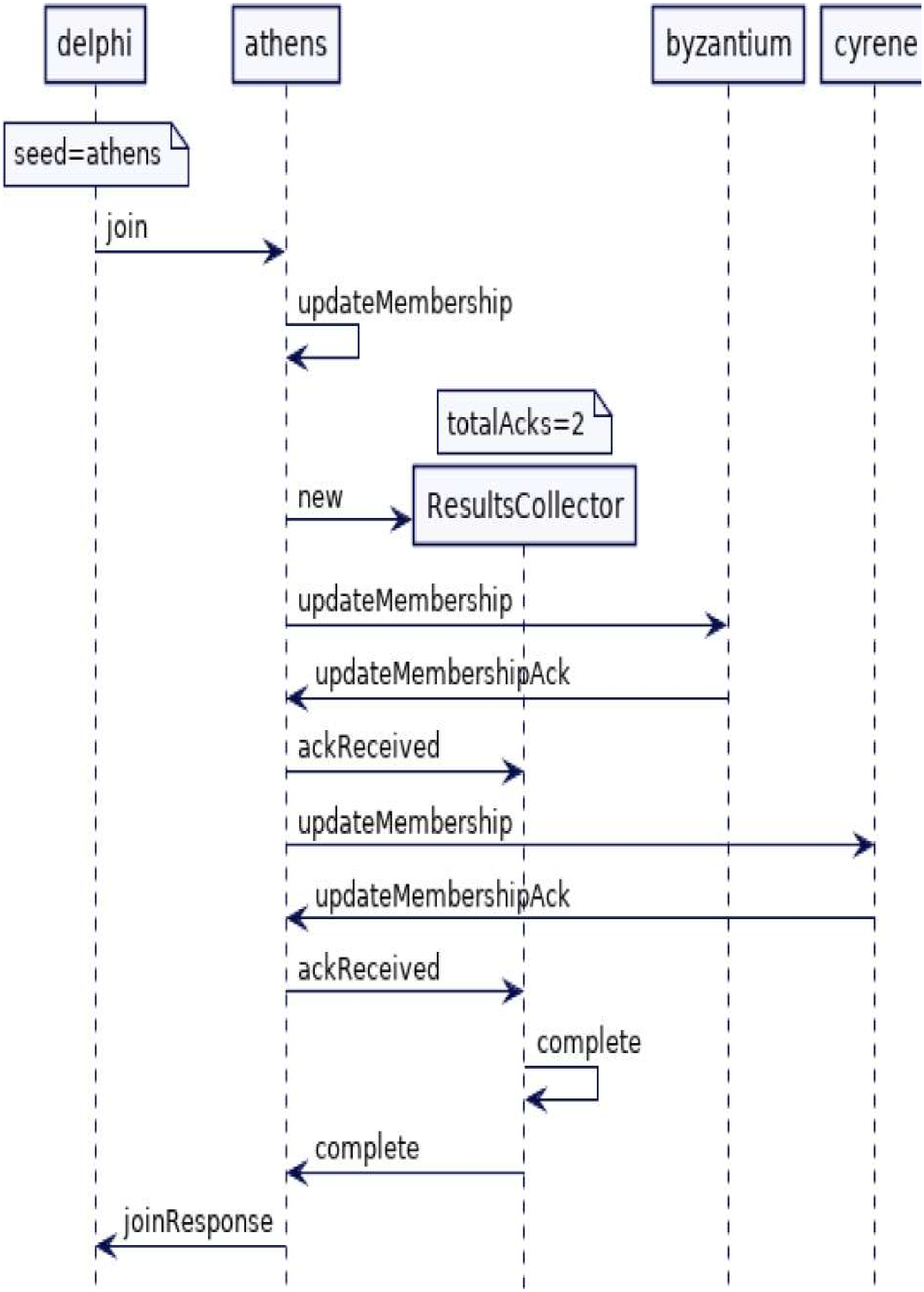
class MembershipService…

```
private void handleResponse(Message message) {
  completePendingRequests(message);
}

private void completePendingRequests(Message message) {
  CompletableFuture requestFuture = pendingRequests.get(message.mes
  if (requestFuture != null) {
     requestFuture.complete(message);
  }
}
```

class ResultsCollector…

```
class ResultsCollector {
  int totalAcks;
  int receivedAcks;
  CompletableFuture future = new CompletableFuture();

  public ResultsCollector(int totalAcks) {
    this.totalAcks = totalAcks;
  }

  public void ackReceived() {
    receivedAcks++;
    if (receivedAcks == totalAcks) {
       future.complete(true);
    }
  }

  public void orTimeout(int time, TimeUnit unit) {
    future.orTimeout(time, unit);
  }

  public void whenComplete(BiConsumer<? super Object, ? super Throw
    future.whenComplete(func);
  }

  public void complete() {
    future.complete("true");
  }
}
```

To see how ResultCollector works, consider a cluster with a set of nodes: let’s call them athens, byzantium and cyrene. athens is acting as a coordinator. When a new node - delphi - sends a join request to athens, athens updates the membership and sends the updateMembership request to byantium and cyrene. It also creates a ResultCollector object to track acknowledgements. It records each acknowledgement received with ResultCollector. When it receives acknowledgements from both byzantium and cyrene, it then responds to delphi.

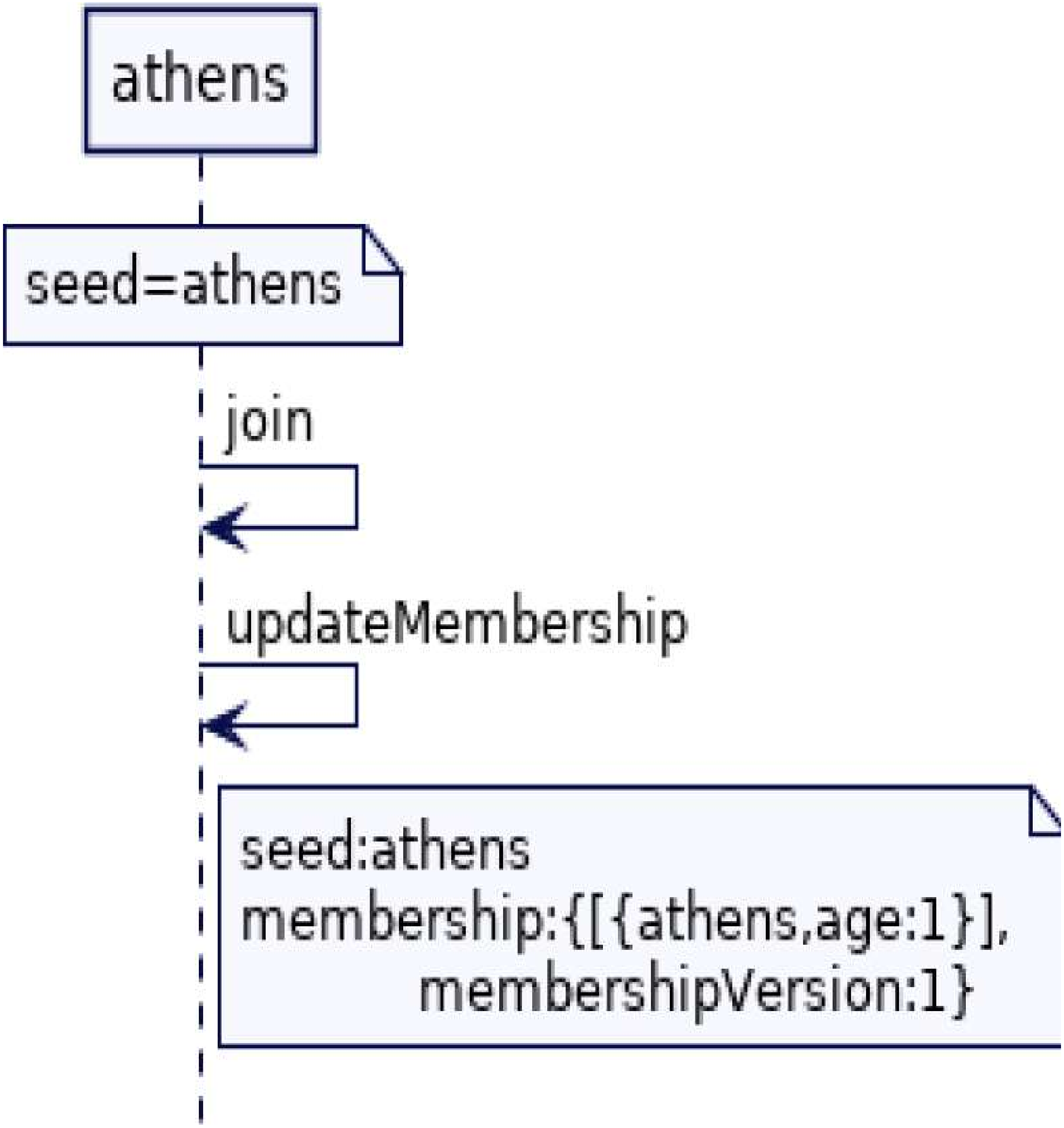

delphi
athens
byzantium
cyrene
seed=athens
join
updateMembership
totalAcks=2
new
ResultsCollector
updateMembership
updateMembershipAck
ackReceived
updateMembership
updateMembershipAck
ackReceived
complete
complete
joinResponse

Frameworks like Akka [bib-akka] use *Gossip Dissemination* and Gossip Convergence [bib-gossip-convergence] to track whether updates have reached all cluster nodes.

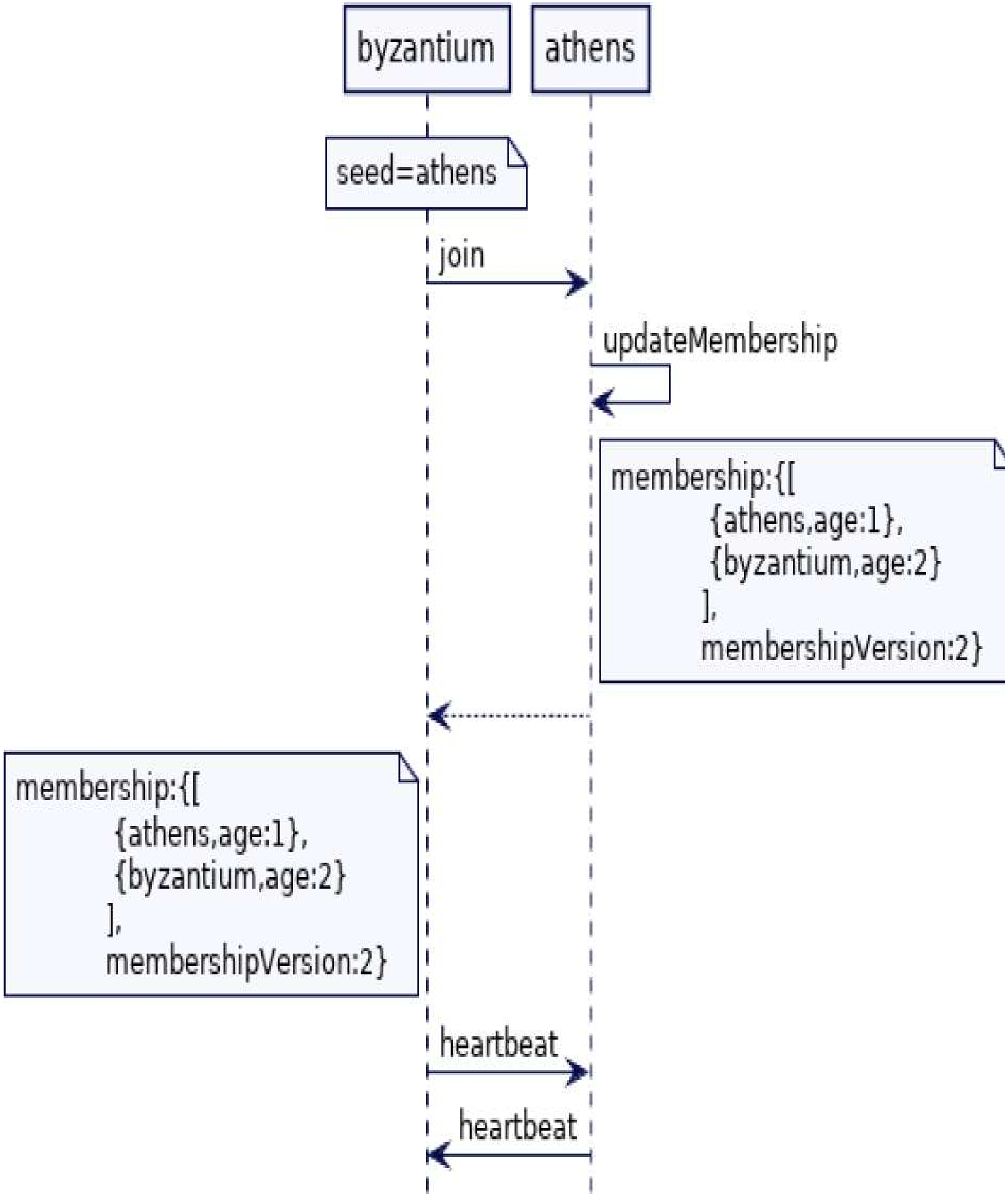
An example scenario

Consider another three nodes. Again, we'll call them athens, byzantium and cyrene. athens acts as a seed node; the other two nodes are configured as such.

When athens starts, it detects that it is itself the seed node. It immediately initializes the membership list and starts accepting requests.

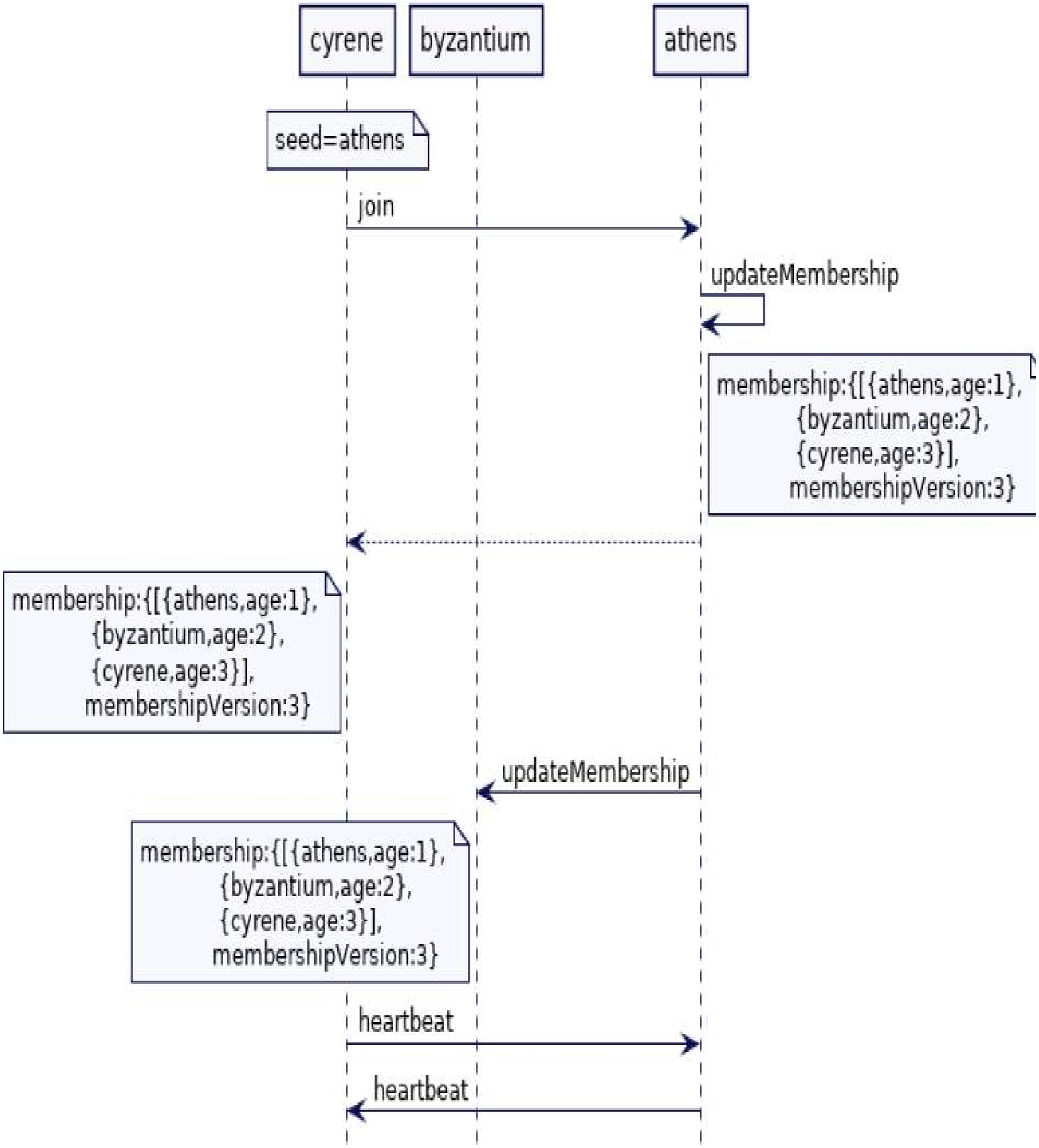


When byzantium starts, it sends a join request to athens. Note that even if byzantium starts before athens, it will keep trying to send join requests until it can connect to athens. Athens finally adds byzantium to the membership list and sends the updated membership list to byzantium. Once byzantium receives the response from athens, it can start accepting requests.



With all-to-all heartbeating, byzantium starts sending heartbeats to athens, and athens sends heartbeat to byzantium.

cyrene starts next. It sends join requests to athens. Athens updates the membership list and sends updated membership list to byantium. It then sends the join response with the membership list to cyrene.



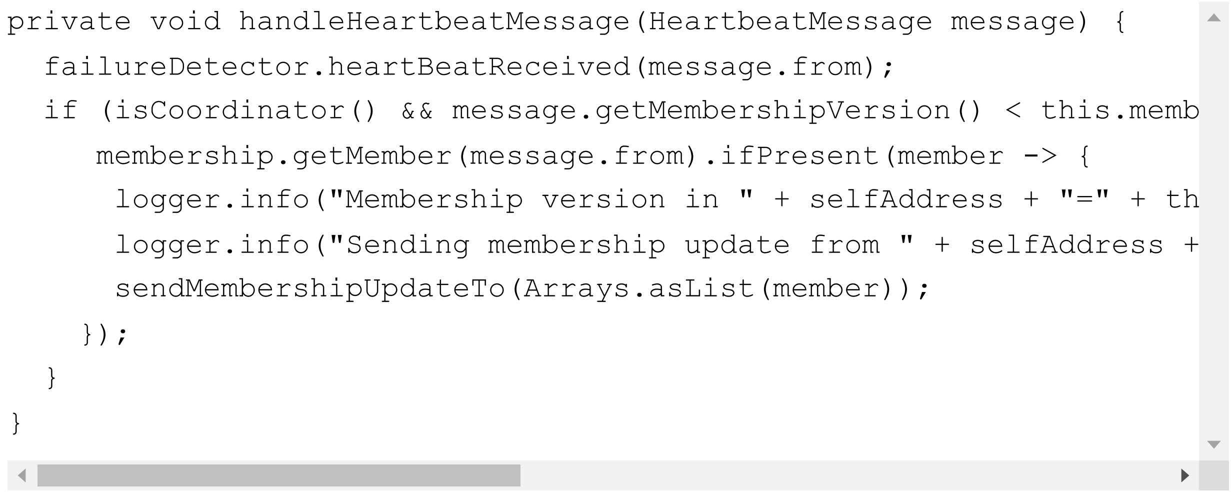
With all to all heartbeating, cyrene, athens and byzantium all send heartbeats to each other.

Handling missing membership updates

It’s possible that some cluster nodes miss membership updates. There are two solutions to handle this problem.

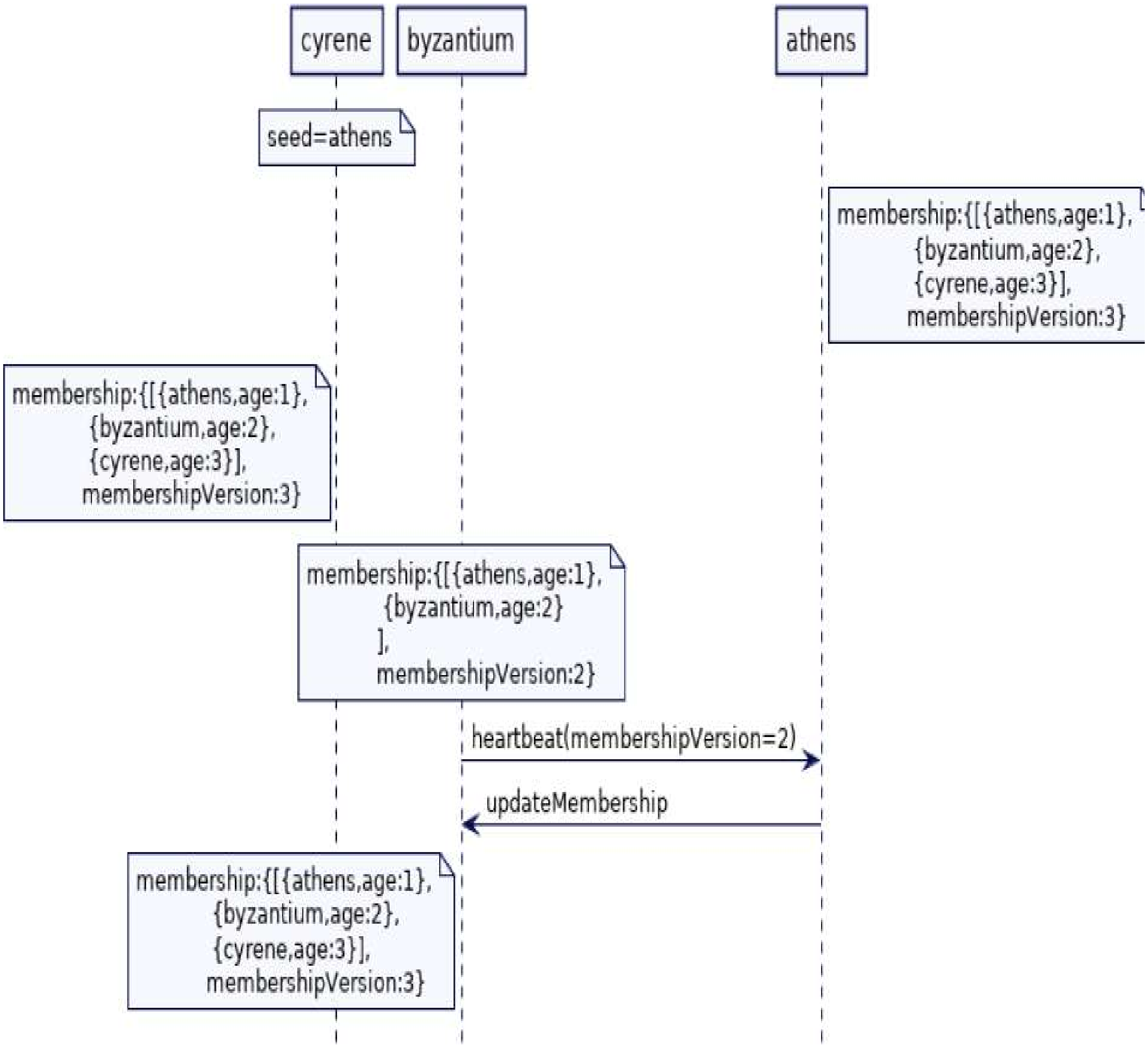
If all members are sending heartbeat to all other members, the membership version number can be sent as part of the heartbeat. The cluster node that handles the heartbeat can then ask for the latest membership. Frameworks like Akka [bib-akka] which use *Gossip Dissemination* track convergence [bib-akka-gossip-convergence] of the gossiped state.

class MembershipService…



```
private void handleHeartbeatMessage(HeartbeatMessage message) {
  failureDetector.heartBeatReceived(message.from);
  if (isCoordinator() && message.getMembershipVersion() < this.memb
     membership.getMember(message.from).ifPresent(member -> {
      logger.info("Membership version in " + selfAddress + "=" + th
      logger.info("Sending membership update from " + selfAddress +
      sendMembershipUpdateTo(Arrays.asList(member));
    });
  }
}
```

In the above example, if byzantium misses the membership update from athens, it will be detected when byzantine sends the heartbeat to athens. athens can then send the latest membership to byzantine.



Alternatively each cluster node can check the lastest membership list periodically, -say every one second - with other cluster nodes. If any of the nodes figure out that their member list is outdated, it can then ask for the latest membership list so it can update it. To be able to compare membership lists, generally a version number is maintained and incremented everytime there is a change.

Failure Detection

Each cluster also runs a failure detector to check if heartbeats are missing from any of the cluster nodes. In a simple case, all cluster nodes send heartbeats to all the other nodes. But only the coordinator marks the nodes as failed and communicates the updated membership list to all the other nodes. This makes sure that not all nodes unilaterally deciding if some other nodes have failed. Hazelcast [bib-hazelcast] is an example of this implementation.

class MembershipService…

```
private boolean isCoordinator() {
  Member coordinator = membership.getCoordinator();
  return coordinator.address.equals(selfAddress);
}

TimeoutBasedFailureDetector<InetAddressAndPort> failureDetector
    = new TimeoutBasedFailureDetector<InetAddressAndPort>(Duration.

private void checkFailedMembers(List<Member> members) {
  if (isCoordinator()) {
     removeFailedMembers();

  } else {
    //if failed member consists of coordinator, then check if this
    claimLeadershipIfNeeded(members);
  }
}

void removeFailedMembers() {
  List<Member> failedMembers = checkAndGetFailedMembers(membership.
  if (failedMembers.isEmpty()) {
     return;
  }
  updateMembership(membership.failed(failedMembers));
  sendMembershipUpdateTo(membership.getLiveMembers());
}
```

Avoiding all-to-all heartbeating

All-to-all heartbeating is not feasible in large clusters. Typically each node will receive heartbeats from only a few other nodes. If a failure is detected, it’s broadcasted to all the other nodes including the coordinator.

For example in Akka [bib-akka] a node ring is formed by sorting network addresses and each cluster node sends heartbeats to only a few cluster nodes. Ignite [bib-ignite] arranges all the nodes in the cluster in a ring and each node sends heartbeat only to the node next to it. Hazelcast [bib-hazelcast] uses all-to-all heartbeat.

Any membership changes, because of nodes being added or node failures need to be broadcast to all the other cluster nodes. A node can connect to every other node to send the required information. *Gossip Dissemination* can be used to broadcast this information.

Split Brain Situation

Even though a single coordinator node decides when to mark another nodes as down, there’s no explicit leader-election happening to select which node acts as a coordinator. Every cluster node expects a heartbeat from the existing coordinator node; if it doesn’t get a heartbeat in time, it can then claim to be the coordinator and remove the existing coordinator from the memberlist.

class MembershipService…

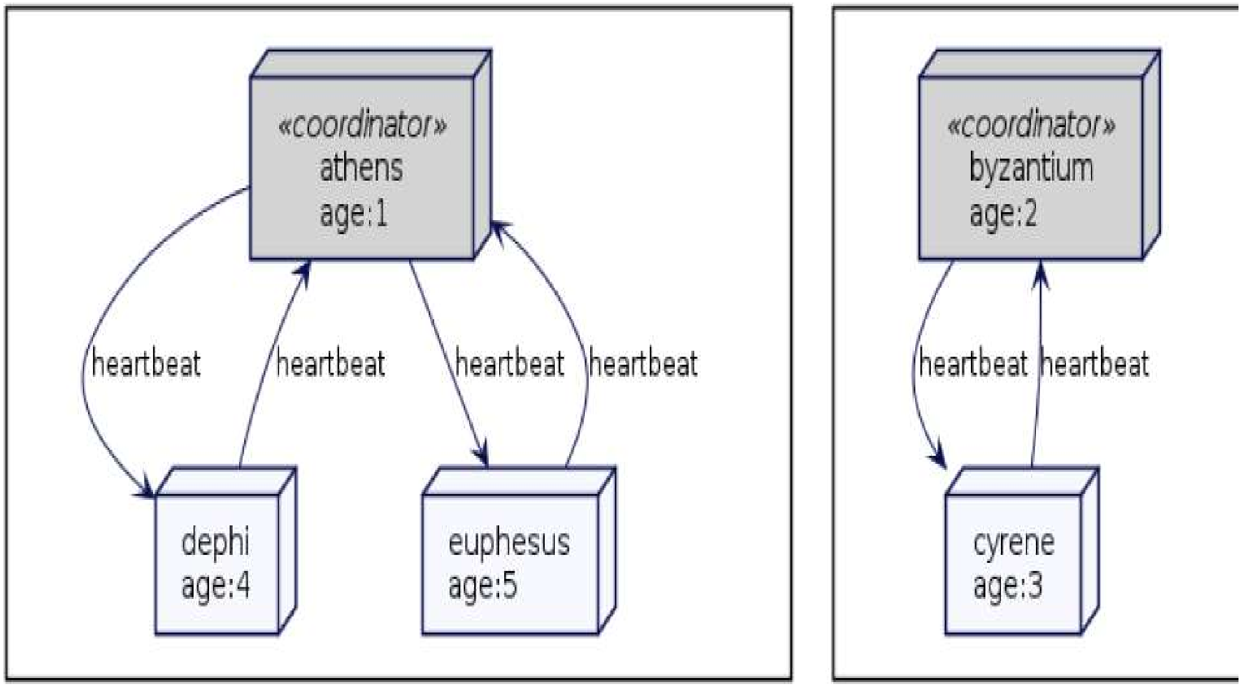
```
private void claimLeadershipIfNeeded(List<Member> members) {
  List<Member> failedMembers = checkAndGetFailedMembers(members);
  if (!failedMembers.isEmpty() && isOlderThanAll(failedMembers)) {
     var newMembership = membership.failed(failedMembers);
     updateMembership(newMembership);
     sendMembershipUpdateTo(newMembership.getLiveMembers());
  }
}

private boolean isOlderThanAll(List<Member> failedMembers) {
  return failedMembers.stream().allMatch(m -> m.age < thisMember().
}

private List<Member> checkAndGetFailedMembers(List<Member> members)
  List<Member> failedMembers = members
```

```
          .stream()
          .filter(member -> !member.address.equals(selfAddress) && fai
          .map(member -> new Member(member.address, member.age, member

   failedMembers.forEach(member->{
      failureDetector.remove(member.address);
      logger.info(selfAddress + " marking " + member.address + " as
   });
   return failedMembers;
}
```

This can create a situation where there are two or more subgroups formed in an existing cluster, each considering the others to have failed. This is called split-brain problem.

Consider a five node cluster, athens, byzantium, cyrene, delphi and euphesus. If athens receives heartbeats from dephi and euphesus, but stops getting heartbeats from byzantium, cyrene, it marks both byzantium and cyrene as failed.

byzantium and cyrene could send heartbeats to each other, but stop receiving heartbeats from cyrene, dephi and euphesus. byzantium being the second oldest member of the cluster, then becomes the coordinator. So two separate clusters are formed one with athens as the coordinator and the other with byzantium as the coordinator.



Handling split brain

One common way to handle split brain issue is to check whether there are enough members to handle any client request, and reject the request if there are not enough live members. For example, Hazelcast [bib-hazelcast] allows you to configure minimum cluster size to execute any client request.

```
public void handleClientRequest(Request request) {
  if (!hasMinimumRequiredSize()) {
      throw new NotEnoughMembersException("Requires minium 3 members
  }
}

private boolean hasMinimumRequiredSize() {
  return membership.getLiveMembers().size() > 3;
}
```

The part which has the majority of the nodes, continues to operate, but as explained in the Hazelcast documentation, there will always be a time window [bib-hazelcast-split-brain-time-window] in which this protection has yet to come into effect.

The problem can be avoided if cluster nodes are not marked as down unless it's guaranteed that they won't cause split brain. For example, Akka [bib-akka] recommends that you don't have nodes marked as down [bib-akka-auto-downing] through the failure detector; you can instead use its split brain resolver. [bib-akka-split-brain-resolver] component.

Recovering from split brain

The coordinator runs a periodic job to check if it can connect to the failed nodes. If a connection can be established, it sends a special message indicating that it wants to trigger a split brain merge.

If the receiving node is the coordinator of the subcluster, it will check to see if the cluster that is initiating the request is part of the minority group. If it is, it will send a merge request. The coordinator of the minority group, which receives the merge request, will then execute the merge request on all the nodes in the minority sub group.

class MembershipService…

```
splitbrainCheckTask = taskScheduler.scheduleWithFixedDelay(() -> {
          searchOtherClusterGroups();
      },
      1, 1, TimeUnit.SECONDS);
```

class MembershipService…

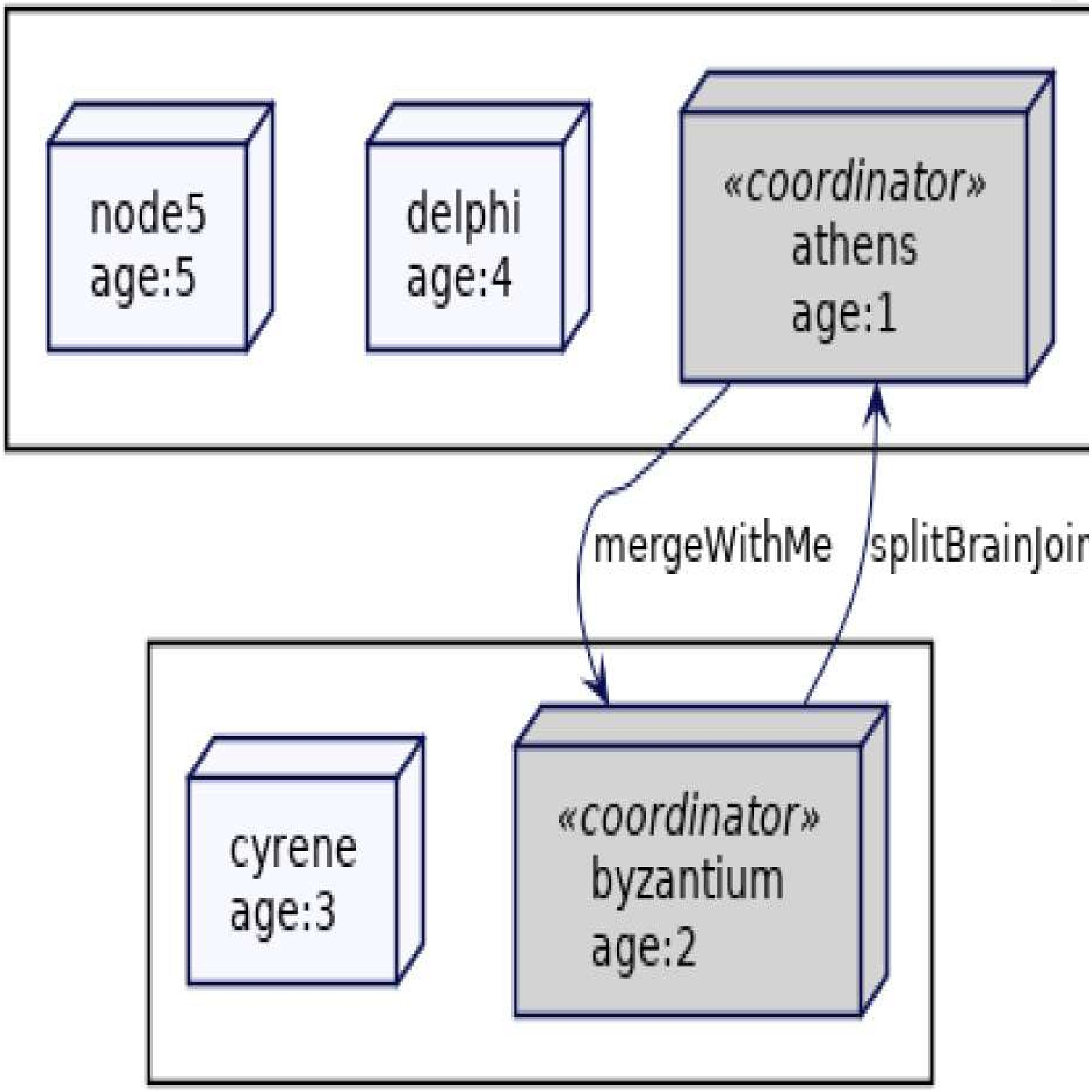
```
private void searchOtherClusterGroups() {
  if (membership.getFailedMembers().isEmpty()) {
     return;
  }
  List<Member> allMembers = new ArrayList<>();
  allMembers.addAll(membership.getLiveMembers());
  allMembers.addAll(membership.getFailedMembers());
    if (isCoordinator()) {
    for (Member member : membership.getFailedMembers()) {
      logger.info("Sending SplitBrainJoinRequest to " + member.addr
      network.send(member.address, new SplitBrainJoinRequest(messag
    }
  }
}
```

If the receiving node is the coordinator of the majority subgroup, it asks the sending coordinator node to merge with itself.

class MembershipService…

```
private void handleSplitBrainJoinMessage(SplitBrainJoinRequest spli
  logger.info(selfAddress + " Handling SplitBrainJoinRequest from "
  if (!membership.isFailed(splitBrainJoinRequest.from)) {
     return;
  }

  if (!isCoordinator()) {
     return;
  }

  if(splitBrainJoinRequest.getMemberCount() < membership.getLiveMem
     //requesting node should join this cluster.
     logger.info(selfAddress + " Requesting " + splitBrainJoinReque
     network.send(splitBrainJoinRequest.from, new SplitBrainMergeMe

  } else {
    //we need to join the other cluster
    mergeWithOtherCluster(splitBrainJoinRequest.from);
  }
}

private void mergeWithOtherCluster(InetAddressAndPort otherClusterC
  askAllLiveMembersToMergeWith(otherClusterCoordinator);
  handleMerge(new MergeMessage(messageId++, selfAddress, otherClust
}

private void askAllLiveMembersToMergeWith(InetAddressAndPort mergeT
  List<Member> liveMembers = membership.getLiveMembers();
  for (Member m : liveMembers) {
     network.send(m.address, new MergeMessage(messageId++, selfAddr
    }
}
```

In the example discussed in the above section, when athens can communicate with byzantium, it will ask byzantium to merge with itself.

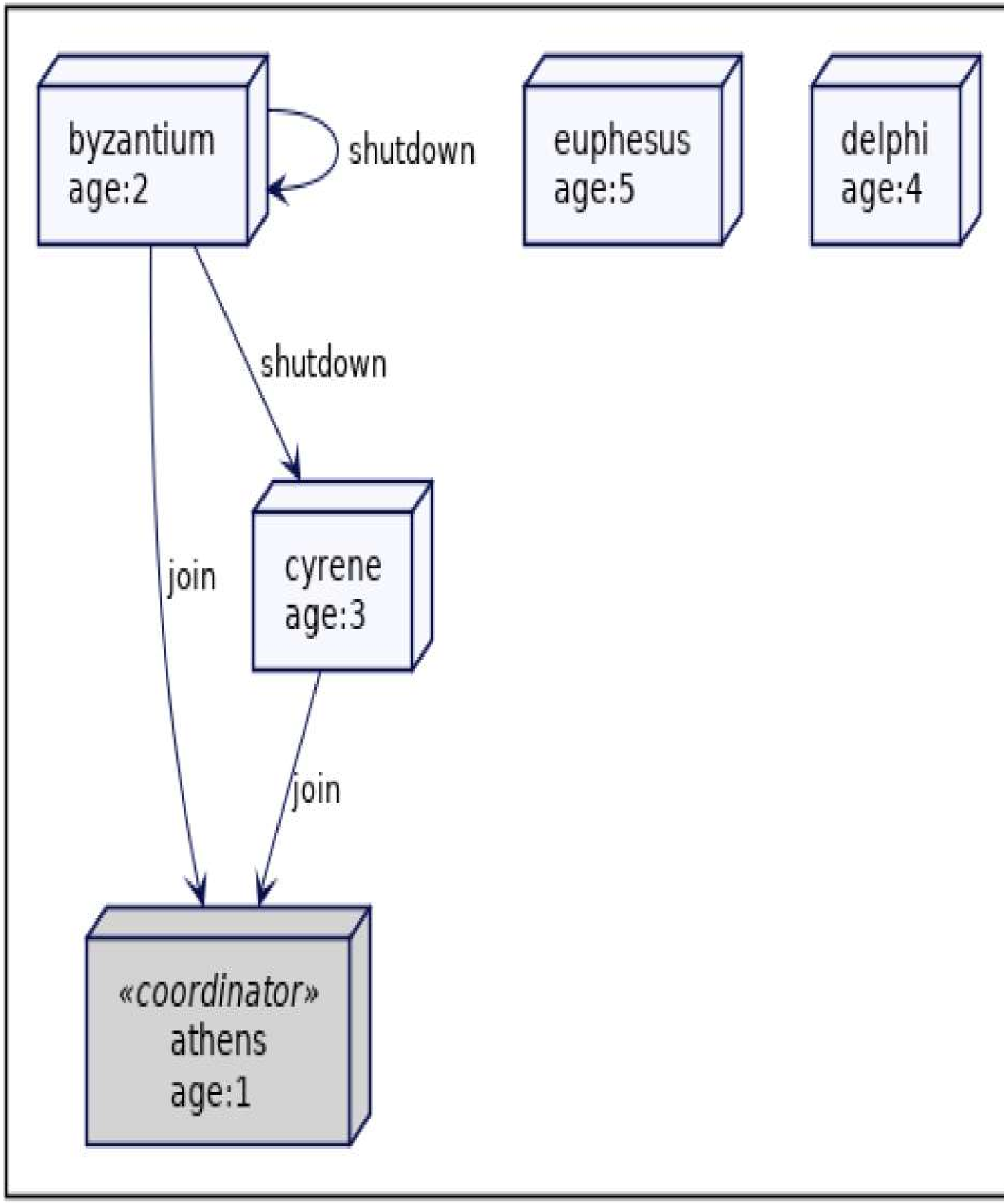


The coordinator of the smaller subgroup, then asks all the cluster nodes inside its group to trigger a merge. The merge operation shuts down and rejoins the cluster nodes to the coordinator of the larger group.

class MembershipService…

```
private void handleMerge(MergeMessage mergeMessage) {
  logger.info(selfAddress + " Merging with " + mergeMessage.getMerg
  shutdown();
  //join the cluster again through the other cluster’s coordinator
  taskScheduler.execute(()-> {
     join(mergeMessage.getMergeToAddress());
  });
}
```

In the example above, byzantium and cyrene shutdown and rejoin athens to form a full cluster again.



Comparison with *Leader and Followers*

It's useful to compare this pattern with that of Leader and Followers. The leader-follower setup, as used by patterns like *Consistent Core*, does not function unless the leader is selected by running an election. This guarantees that the *Quorum* of cluster nodes have an agreement about who the leader is. In the worst case scenario, if an agreement isn't reached, the system will be unavailable to process any requests. In other words, it prefers consistency over availability.

The emergent leader, on the other hand will always have some cluster node acting as a leader for processing client requests. In this case, availability is preferred over consistency.

Examples

In JGroups [bib-jgroups] the oldest member is the coordinator and decides membership changes. In Akka [bib-akka] the oldest member of the cluster runs actor singletons like shard coordinator which decide the placement of *Fixed Partitions* across cluster nodes. In-memory data grids like Hazelcast [bib-hazelcast] and Ignite [bib-ignite] have the oldest member as the cluster coordinator.

Part VI: Patterns of communication between nodes

Chapter 30. Single Socket Channel

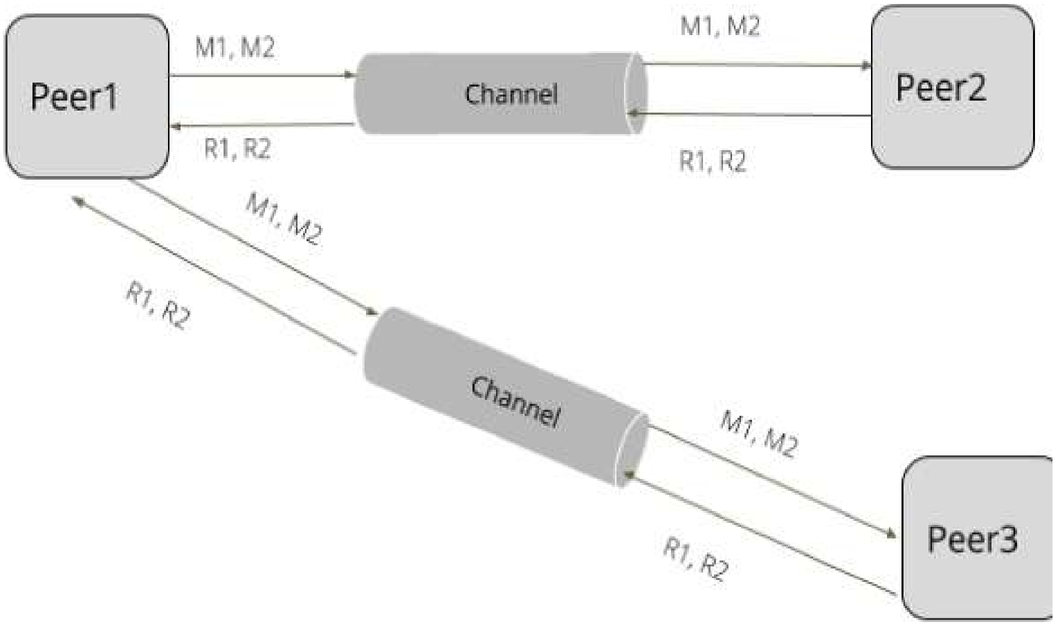
Maintain order of the requests sent to a server by using a single TCP connection.

Problem

When we are using *Leader and Followers*, we need to ensure that messages between the leader and each follower are kept in order, with a retry mechanism for any lost messages. We need to do this while keeping the cost of new connections low, so that opening new connections doesn’t increase the system’s latency.

Solution

Fortunately, the long-used and widely available TCP [bib-tcp] mechanism provides all these necessary characteristics. Thus we can get the communication we need by ensuring all communication between a follower and its leader goes through a single socket channel. The follower then serializes the updates from leader using a *Singular Update Queue*



© 2019 ThoughtWorks

Nodes never close the connection once it is open and continuously read it for new requests. Nodes use a dedicated thread per connection to read and write requests. A thread per connection isn't needed if non blocking io [bib-java-nio] is used.

A simple-thread based implementation will be like the following:

class SocketHandlerThread…

```
@Override
public void run() {
  isRunning = true;
  try {
     //Continues to read/write to the socket connection till it is
     while (isRunning) {
       handleRequest();
    }
  } catch (Exception e) {
    getLogger().debug(e);
    closeClient(this);
  }
}

private void handleRequest() {
  RequestOrResponse request = clientConnection.readRequest();
  RequestId requestId = RequestId.valueOf(request.getRequestId());
  server.accept(new Message<>(request, requestId, clientConnection)
}

public void closeConnection() {
  clientConnection.close();
}
```

The node reads requests and submits them to a Singular Update Queue for processing. Once the node has processed the request it writes the response back to the socket.

Whenever a node establishes a communication it opens a single socket connection that's used for all requests with the other party.

class SingleSocketChannel…

```
public class SingleSocketChannel implements Closeable {
  final InetAddressAndPort address;
  final int heartbeatIntervalMs;
  private Socket clientSocket;
  private final OutputStream socketOutputStream;
  private final InputStream inputStream;

  public SingleSocketChannel(InetAddressAndPort address, int heartb
    this.address = address;
    this.heartbeatIntervalMs = heartbeatIntervalMs;
    clientSocket = new Socket();
    clientSocket.connect(new InetSocketAddress(address.getAddress()
    clientSocket.setSoTimeout(heartbeatIntervalMs * 10); //set sock
    socketOutputStream = clientSocket.getOutputStream();
    inputStream = clientSocket.getInputStream();
}
public synchronized RequestOrResponse blockingSend(RequestOrRespons
  writeRequest(request);
  byte[] responseBytes = readResponse();
  return deserialize(responseBytes);
}

private void writeRequest(RequestOrResponse request) throws IOExcep
  var dataStream = new DataOutputStream(socketOutputStream);
  byte[] messageBytes = serialize(request);
  dataStream.writeInt(messageBytes.length);
  dataStream.write(messageBytes);
}
```

It’s important to keep a timeout on the connection so it doesn’t block indefinitely in case of errors. We use *HeartBeat* to send requests periodically over the socket channel to keep it alive. This timeout is generally kept as a multiple of the HeartBeat interval, to allow for network round trip time and some possible network delays. It’s reasonable to keep the connection timeout as say 10 times that of the HeartBeat interval.

class SocketListener…

```
private void setReadTimeout(Socket clientSocket) throws SocketExcep
  clientSocket.setSoTimeout(config.getHeartBeatIntervalMs() * 10);
}
```

Sending requests over a single channel can create a problem with head of line blocking [bib-head-of-line-blocking] issues. To avoid these, we can use a *Request Pipeline*.

Examples

- Zookeeper [bib-zookeeper-internals] uses a single socket channel and a thread per follower to do all the communication.

- Kafka [bib-kafka-replication] uses a single socket channel between follower and leader partitions to replicate messages.
- Reference implementation of the Raft [bib-raft] consensus algorithm, LogCabin [bib-logcabin-raft] uses single socket channel to communicate between leader and followers

Chapter 31. Request Batch

Combine multiple requests to optimally utilise the network.

Problem

When requests are sent to cluster nodes, if a lot of requests are sent with a small amount of data, network latency and the request processing time (including serialization, deserialization of the request on the server side) can add significant overhead.

For example, if a network's capacity is 1gbps and its latency and request processing time is, say, 100 microseconds, if the client is sending hundreds of requests at the same time — each one just a few bytes — it will significantly limit the overall throughput if each request needs 100 microseconds to complete.

Solution

Combine multiple requests together into a single request batch. The batch of the request will be sent to the cluster node for processing. with each request processed in exactly the same manner as an individual request. It will then respond with the batch of the responses.

As an example, consider a distributed key-value store, where the client sends requests to store multiple key-values on the server. When the client receives a call to send the request, it does not immediately send it over the network; instead, it keeps a queue of requests to be sent.

class Client…

```
LinkedBlockingQueue<RequestEntry> requests = new LinkedBlockingQue

public CompletableFuture send(SetValueRequest setValueRequest) {
  int requestId = enqueueRequest(setValueRequest);
  CompletableFuture responseFuture = trackPendingRequest(requestId
  return responseFuture;
}

private int enqueueRequest(SetValueRequest setValueRequest) {
  int requestId = nextRequestId();
  byte[] requestBytes = serialize(setValueRequest, requestId);
  requests.add(new RequestEntry(requestBytes, clock.nanoTime()));
  return requestId;
}
private int nextRequestId() {
  return requestNumber++;
}
```

The time at which the request is enqued is tracked; this is later used to decide if the request can be sent as part of the batch.

class RequestEntry…

```
class RequestEntry {
  byte[] serializedRequest;
  long createdTime;

  public RequestEntry(byte[] serializedRequest, long createdTime)
    this.serializedRequest = serializedRequest;
    this.createdTime = createdTime;
}
```

It then tracks the pending requests to be completed when a response is received. Each request will be assigned a unique request number which can be used to map the response and complete the requests.

class Client…

```
Map<Integer, CompletableFuture> pendingRequests = new ConcurrentHa

private CompletableFuture trackPendingRequest(Integer correlationI
  CompletableFuture responseFuture = new CompletableFuture();
  pendingRequests.put(correlationId, responseFuture);
  return responseFuture;
}
```

The client starts a separate task which continuously tracks the queued requests.

class Client…

```
public Client(Config config, InetAddressAndPort serverAddress, Sys
  this.clock = clock;
  this.sender = new Sender(config, serverAddress, clock);
  this.sender.start();
}
```

class Sender…

```
@Override
public void run() {
  while (isRunning) {
    boolean maxWaitTimeElapsed = requestsWaitedFor(config.getMaxBa
    boolean maxBatchSizeReached = maxBatchSizeReached(requests);
    if (maxWaitTimeElapsed || maxBatchSizeReached) {
       RequestBatch batch = createBatch(requests);
       try {
          BatchResponse batchResponse = sendBatchRequest(batch, ad
          handleResponse(batchResponse);

       } catch (IOException e) {
         batch.getPackedRequests().stream().forEach(r -> {
         pendingRequests.get(r.getCorrelationId()).completeExcepti
       });
```

```
     }
   }
 }
}

private RequestBatch createBatch(LinkedBlockingQueue<RequestEntry>
  RequestBatch batch = new RequestBatch(MAX_BATCH_SIZE_BYTES);
  RequestEntry entry = requests.peek();
  while (entry != null && batch.hasSpaceFor(entry.getRequest())) {
    batch.add(entry.getRequest());
    requests.remove(entry);
    entry = requests.peek();
  }
  return batch;
}
```

class RequestBatch…

```
public boolean hasSpaceFor(byte[] requestBytes) {
  return batchSize() + requestBytes.length <= maxSize;
}
private int batchSize() {
  return requests.stream().map(r->r.length).reduce(0, Integer::sum
}
```

There are two checks which are generally done.

class Sender…

```
private boolean maxBatchSizeReached(Queue<RequestEntry> requests)
  return accumulatedRequestSize(requests) > MAX_BATCH_SIZE_BYTES;
}

private int accumulatedRequestSize(Queue<RequestEntry> requests) {
  return requests.stream().map(re -> re.size()).reduce((r1, r2) ->
}
```

class Sender…

```
private boolean requestsWaitedFor(long batchingWindowInMs) {
  RequestEntry oldestPendingRequest = requests.peek();
  if (oldestPendingRequest == null) {
     return false;
  }
  long oldestEntryWaitTime = clock.nanoTime() - oldestPendingReque
  return oldestEntryWaitTime > batchingWindowInMs;
}
```

- If enough requests have accumulated to fill the batch to the maximum configured size.
- Because we cannot wait forever for the batch to be filled in, we can configure a small amount of wait time. The sender task waits and then checks if the request has been added before the maximum wait time.

Once any of these conditions has been fulfilled the batch request can then be sent to the server. The server unpacks the batch request, and processes each of the individual requests.

class Server…

```
private void handleBatchRequest(RequestOrResponse batchRequest, Cl
  RequestBatch batch = JsonSerDes.deserialize(batchRequest.getMess
  List<RequestOrResponse> requests = batch.getPackedRequests();
  List<RequestOrResponse> responses = new ArrayList<>();
  for (RequestOrResponse request : requests) {
    RequestOrResponse response = handleSetValueRequest(request);
    responses.add(response);
  }
  sendResponse(batchRequest, clientConnection, new BatchResponse(r
}

private RequestOrResponse handleSetValueRequest(RequestOrResponse
  SetValueRequest setValueRequest = JsonSerDes.deserialize(request
  kv.put(setValueRequest.getKey(), setValueRequest.getValue());
  RequestOrResponse response = new RequestOrResponse(RequestId.Set
  return response;
}
```

The client receives the batch response and completes all the pending requests.

class Sender…

```
private void handleResponse(BatchResponse batchResponse) {
  List<RequestOrResponse> responseList = batchResponse.getResponse
  logger.debug("Completing requests from " + responseList.get(0).g
  responseList.stream().forEach(r -> {
    CompletableFuture completableFuture = pendingRequests.remove(r
     if (completableFuture != null) {
        completableFuture.complete(r);
     } else {
         logger.error("no pending request for " + r.getCorrelationId
     }
  });
}
```

Technical Considerations

The batch size should be chosen based on the size of individual messages and available network bandwidth as well as the observed latency and throughput improvements based on the real life load. These are configured to some sensible defaults assuming smaller message sizes and the optimal batch size for server side processing. For example, Kafka [bib-kafka] has a default batch size of 16Kb. It also has a configuration parameter called "linger.ms" with the default value of 0. However if the size of the messages are bigger a higher batch size might work better.

Having too large a batch size will likely only offer diminishing returns. For example having a batch size in MBs can add further overheads in terms of processing. This is why the batch size parameter is typically tuned according to observations made through performance testing.

A request batch is generally used along with *Request Pipeline* to improve overall throughput and latency.

When the retry-backoff policy is used to send requests to cluster nodes, the entire batch request will be retried. The cluster node might have processed part of the batch already; so to ensure the retry works without any issues, you should implement *Idempotent Receiver*.

Examples

Kafka [bib-kafka] supports the batch of the producer requests.

Batching is also used when saving data to disk. For example [bookkeeper] [bib-bookkeeper] implements the batching in a similar way to flush the log to the disk.

Nagel's Algorithm [bib-tcp-nagel] is used in TCP to batch multiple smaller packets together to improve overall network throughput.

Chapter 32. Request Pipeline

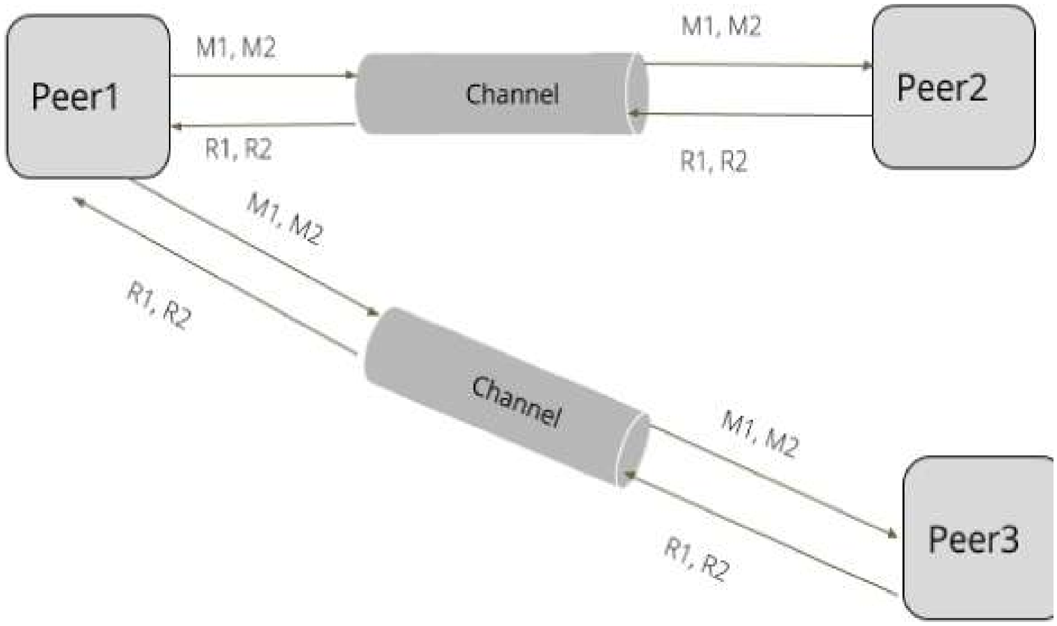
Improve latency by sending multiple requests on the connection without waiting for the response of the previous requests.

Problem

Communicating between servers within a cluster using *Single Socket Channel* can cause performance issues if requests need to wait for responses for previous requests to be returned. To achieve better throughput and latency, the request queue on the server should be filled enough to make sure server capacity is fully utilized. For example, when *Singular Update Queue* is used within a server, it can always accept more requests until the queue fills up, while it’s processing a request. If only one request is sent at a time, most of the server capacity is unnecessarily wasted.

Solution

Nodes send requests to other nodes without waiting for responses from previous requests. This is achieved by creating two separate threads, one for sending requests over a network channel and one for receiving responses from the network channel.



© 2019 ThoughtWorks

The sender node sends the requests over the socket channel, without waiting for response.

class SingleSocketChannel…

```
public void sendOneWay(RequestOrResponse request) throws IOExceptio
  var dataStream = new DataOutputStream(socketOutputStream);
  byte[] messageBytes = serialize(request);
  dataStream.writeInt(messageBytes.length);
  dataStream.write(messageBytes);
}
```

A separate thread is started to read responses.

class ResponseThread…

```
class ResponseThread extends Thread implements Logging {
  private volatile boolean isRunning = false;
  private SingleSocketChannel socketChannel;
```

```
 public ResponseThread(SingleSocketChannel socketChannel) {
  this.socketChannel = socketChannel;
 }

 @Override
 public void run() {
  try {
   isRunning = true;
   logger.info("Starting responder thread = " + isRunning);
   while (isRunning) {
    doWork();
   }

 } catch (IOException e) {
  e.printStackTrace();
  getLogger().error(e); //thread exits if stopped or there is IO
 }
}

public void doWork() throws IOException {
 RequestOrResponse response = socketChannel.read();
 logger.info("Read Response = " + response);
 processResponse(response);
}
```

The response handler can immediately process the response or submits it to a Singular Update Queue

There are two issues with the request pipeline which need to be handled.

If requests are continuously sent without waiting for the response, the node accepting the request can be overwhelmed. For this reason, there is an upper limit on how many requests can be kept inflight at a time. Any node can send up to the maximum number of requests to other nodes. Once the maximum inflight requests are sent without receiving the response, no more requests are accepted and the sender is blocked. A very simple strategy to limit maximum inflight requests is to keep a blocking queue to keep track of requests. The queue is initialized with the number of requests which can be in flight. Once the response is received for a request, it’s removed from the queue to create room for more requests. As shown in the below code, the maximum of five inflight requests are accepted per socket connection.

class RequestLimitingPipelinedConnection…

```
private final Map<InetAddressAndPort, ArrayBlockingQueue<RequestOrR
private int maxInflightRequests = 5;
public void send(InetAddressAndPort to, RequestOrResponse request)
  ArrayBlockingQueue<RequestOrResponse> requestsForAddress = inflig
  if (requestsForAddress == null) {
     requestsForAddress = new ArrayBlockingQueue<>(maxInflightReque
     inflightRequests.put(to, requestsForAddress);
  }
  requestsForAddress.put(request);
```

The request is removed from the inflight request queue once the response is received.

class RequestLimitingPipelinedConnection…

```
private void consume(SocketRequestOrResponse response) {
  Integer correlationId = response.getRequest().getCorrelationId();
  Queue<RequestOrResponse> requestsForAddress = inflightRequests.ge
  RequestOrResponse first = requestsForAddress.peek();
  if (correlationId != first.getCorrelationId()) {
     throw new RuntimeException("First response should be for the f
  }
  requestsForAddress.remove(first);
  responseConsumer.accept(response.getRequest());
}
```

Handling failures and also maintaining ordering guarantees becomes tricky to implement. Let’s say there are two requests in flight. The first request failed and retried, the server might have processed the second request before the retried first request reaches the server. Servers need some mechanism to make sure out of order requests are rejected. Otherwise, there’s always a risk of messages getting re-ordered in case of failures and retries. For example, Raft [bib-raft] always sends the previous log index that is expected with every log entry. If the previous log index does not match, the server rejects the request. Kafka can allow max.in.flight.requests.per.connection to be more than one, with the idempotent producer implementation [bib-kafka-idempotent-producer], which assigns a unique identifier to each message batch that is sent to the broker. The broker can then check the sequence number of the incoming request and reject the request if the requests are out of order.

Examples

All consensus algorithm such as Zab [bib-zab] and Raft [bib-raft] allow request pipeline support.

Kafka [bib-kafka-protocol] encourages clients to use request pipelining to improve throughput.

Bibliography

[bib-lamport] http://lamport.org

[bib-lamport-paxos-simple] https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

[bib-lamport-paxos-original] http://lamport.azurewebsites.net/pubs/pubs.xhtml#lamport-paxos

[bib-single-writer] https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.xhtml

[bib-seda] https://dl.acm.org/doi/10.1145/502034.502057

[bib-xpe] http://www.amazon.com/exec/obidos/ASIN/0321278658

[bib-lmax] https://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf

[bib-gossip] https://en.wikipedia.org/wiki/Gossip_protocol

[bib-local-pause] https://issues.apache.org/jira/browse/CASSANDRA-9183

[bib-zab] https://zookeeper.apache.org/doc/r3.4.13/zookeeperInternals.xhtml#sc_atomicBroadcast

[bib-zookeeper] https://zookeeper.apache.org/

[bib-zookeeper-hunt-paper] https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf [bib-zookeeper-internals] https://zookeeper.apache.org/doc/r3.4.13/zookeeperInternals.xhtml

[bib-raft] https://raft.github.io/

[bib-raft-phd] https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf

[bib-logcabin-raft] https://github.com/logcabin/logcabin

[bib-logcabin-raft-clustertime] https://ongardie.net/blog/logcabin-2015-02-27/

[bib-Linearizable] https://jepsen.io/consistency/models/linearizable

[bib-etcd] https://etcd.io/

[bib-consul] https://www.consul.io/

[bib-kafka] https://kafka.apache.org/

[bib-kafka-raft] https://cwiki.apache.org/confluence/display/KAFKA/KIP-595%3A+A+Raft+Protocol+for+the+Metadata+Quorum

[bib-kafka-replication-protocol] https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/

[bib-kafka-protocol] https://kafka.apache.org/protocol

[bib-kafka-replication] https://kafka.apache.org/protocol

[bib-kafka-follower-fetch] https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica

[bib-kafka-leader-epoch] https://cwiki.apache.org/confluence/display/KAFKA/KIP-101+-

+Alter+Replication+Protocol+to+use+Leader+Epoch+rather+than+High+Watermark+for+Truncation

[bib-kafka-controller] https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Controller+Internals

[bib-future] https://en.wikipedia.org/wiki/Futures_and_promises

[bib-tcp] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[bib-java-nio] https://en.wikipedia.org/wiki/Non-blocking_I/O_(Java)

[bib-head-of-line-blocking] https://en.wikipedia.org/wiki/Head-of-line_blocking

[bib-grpc] https://grpc.io/

[bib-birman] http://www.amazon.com/exec/obidos/ASIN/1447158423

[bib-applying-usl-to-dist-sys] https://speakerdeck.com/drqz/applying-the-universal-scalability-law-to-distributed-systems?slide=68

[bib-zookeeper-wait-free] https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf

[bib-etcd-parformance] https://coreos.com/blog/performance-of-etcd.xhtml

[bib-paxos] https://en.wikipedia.org/wiki/Paxos_(computer_science)

[bib-multi-paxos] https://www.youtube.com/watch?v=JEpsBg0AO6o&t=1920s

[bib-cassandra] http://cassandra.apache.org/

[bib-cassandra-hinted-handoff] http://cassandra.apache.org/doc/latest/operating/hints.xhtml [bib-cassandra-heartbeat] https://issues.apache.org/jira/browse/CASSANDRA-597

[bib-cassandra-generation] http://cassandra.apache.org/doc/latest/operating/hints.xhtml

[bib-http2] https://tools.ietf.org/html/rfc7540

[bib-fallacies-of-distributed-computing] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[bib-consensus] https://en.wikipedia.org/wiki/Consensus_(computer_science)

[bib-view-stamp-replication] http://pmg.csail.mit.edu/papers/vr-revisited.pdf

[bib-virtual-synchrony] https://www.cs.cornell.edu/ken/History.pdf

[bib-chubby] https://research.google/pubs/pub27897/

[bib-patterns] articles/writingPatterns.xhtml

[bib-alexander] https://en.wikipedia.org/wiki/Christopher_Alexander

[bib-gc] https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

[bib-network-partition] https://en.wikipedia.org/wiki/Network_partition

[bib-acid] https://en.wikipedia.org/wiki/Network_partition

[bib-usl-to-dist-sys] https://speakerdeck.com/drqz/applying-the-universal-scalability-law-to-distributed-systems?slide=68

[bib-usl] http://www.perfdynamics.com/Manifesto/USLscalability.xhtml

[bib-kafka-shutdownable-thread] https://github.com/a0x8o/kafka/blob/master/connect/runtime/src/main/java/org/apache/kafka/connect/util/ShutdownableThread.java

[bib-zookeeper-processor-thread] https://github.com/apache/zookeeper/blob/master/zookeeper-server/src/main/java/org/apache/zookeeper/server/quorum/CommitProcessor.java

[bib-cassandra-local-pause-detection] https://issues.apache.org/jira/browse/CASSANDRA-9183

[bib-cassandra-generation-as-timestamp] https://issues.apache.org/jira/browse/CASSANDRA-515

[bib-log-cabin] https://github.com/logcabin/logcabin

[bib-akka] https://akka.io/

[bib-java-concurrency-in-practice] http://www.amazon.com/exec/obidos/ASIN/0321349601

[bib-bookkeeper] https://bookkeeper.apache.org/

[bib-bookkeeper-protocol] https://bookkeeper.apache.org/archives/docs/r4.4.0/bookkeeperProtocol.xhtml

[bib-designing-data-intensive-applications] http://www.amazon.com/exec/obidos/ASIN/1449373321

[bib-lamport-timestamp] https://en.wikipedia.org/wiki/Lamport_timestamps

[bib-logical-clock] https://en.wikipedia.org/wiki/Logical_clock

[bib-aws-outage] https://aws.amazon.com/message/41926/

[bib-github-outage] https://github.blog/2018-10-30-oct21-post-incident-analysis/

[bib-google-outage] https://status.cloud.google.com/incident/cloud-networking/19009

[bib-correlation-identifier] https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.xhtml

[bib-akka-2400-node-cluster] https://www.lightbend.com/blog/running-a-2400-akka-nodes-cluster-on-google-compute-engine

[bib-rapid] https://www.usenix.org/conference/atc18/presentation/suresh

[bib-stat-machine-replication] https://en.wikipedia.org/wiki/State_machine_replication

[bib-kubernetes] https://kubernetes.io/

[bib-hdfs] https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.xhtml

[bib-flink] https://ci.apache.org/projects/flink/flink-docs-release-1.11/ops/jobmanager_high_availability.xhtml

[bib-spark] http://spark.apache.org/docs/latest/spark-standalone.xhtml#standby-masters-with-zookeeper

[bib-numbers-every-programmer-should-know] http://highscalability.com/numbers-everyone-should-know

[bib-kip-500] https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum

[bib-kip-631] https://cwiki.apache.org/confluence/display/KAFKA/KIP-631%3A+The+Quorum-based+Kafka+Controller [bib-jepsen] https://github.com/jepsen-io/jepsen

[bib-strict-serializability] http://jepsen.io/consistency/models/strict-serializable

[bib-database-consistency] https://jepsen.io/consistency

[bib-serializability] https://jepsen.io/consistency/models/serializable

[bib-ntp] https://en.wikipedia.org/wiki/Network_Time_Protocol

[bib-dhcp] https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol

[bib-dhcp-failover] https://tools.ietf.org/html/draft-ietf-dhc-failover-12

[bib-zookeeper-wal] https://github.com/apache/zookeeper/blob/master/zookeeper-server/src/main/java/org/apache/zookeeper/server/persistence/FileTxnLog.java

[bib-etcd-wal] https://github.com/etcd-io/etcd/blob/master/server/wal/wal.go

[bib-cassandra-wal] https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/db/commitlog/CommitLog.java

[bib-kafka-log] https://github.com/axbaretto/kafka/blob/master/core/src/main/scala/kafka/log/Log.scala

[bib-etcd-watch-channel-issue] https://github.com/etcd-io/etcd/issues/11906

[bib-reactive-streams] https://www.reactive-streams.org/

[bib-rsocket] https://rsocket.io/

[bib-kubernetes-api] https://kubernetes.io/docs/reference/using-api/api-concepts/

[bib-etcd3] https://coreos.com/blog/etcd3-a-new-etcd.xhtml

[bib-etcd3-doc] https://etcd.io/docs/v3.4.0/dev-guide/interacting_v3/

[bib-etcd-readindex-impl] https://github.com/etcd-io/etcd/pull/6212/commits/e3e39930229830b2991ec917ec5d2ba625febd3f

[bib-kafka-idempotent-producer] https://cwiki.apache.org/confluence/display/KAFKA/Idempotent+Producer

[bib-hbase-recoverable-zookeeper] https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.4.0/bk_hbase_java_api/org/apache/hadoop/hbase/zookeeper/RecoverableZooKeeper.xhtml

[bib-zookeeper-error-handling] https://cwiki.apache.org/confluence/display/ZOOKEEPER/ErrorHandling

[bib-cloudflare-outage] https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/

[bib-kafka-enhanced-raft] https://cwiki.apache.org/confluence/display/KAFKA/KIP-650%3A+Enhance+Kafkaesque+Raft+semantics#KIP650:EnhanceKafkaesqueRaftsemantics-Non-leaderLinearizableRead

[bib-serf-convergence-simulator] https://www.serf.io/docs/internals/simulator.xhtml

[bib-gossip-analysis] https://www.coursera.org/lecture/cloud-computing/1-3-gossip-analysis-jjieX

[bib-cockroachdb] https://www.cockroachlabs.com/docs/stable/

[bib-cockroachdb-hybridclock] https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer.xhtml

[bib-cockroachdb-design] https://github.com/cockroachdb/cockroach/blob/master/docs/design.md

[bib-kip-631-configurations] https://cwiki.apache.org/confluence/display/KAFKA/KIP-631%3A+The+Quorum-based+Kafka+Controller#KIP631:TheQuorumbasedKafkaController-Configurations

[bib-swim-gossip] https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

[bib-wan-gossip] https://dl.acm.org/doi/10.1109/TPDS.2006.85

[bib-epidemiology] https://en.wikipedia.org/wiki/Epidemiology

[bib-hyperledger-fabric] https://github.com/hyperledger/fabric

[bib-hyperledger-fabric-gossip] https://hyperledger-fabric.readthedocs.io/en/release-2.2/gossip.xhtml

[bib-mongodb-cluster-logical-clock] https://dl.acm.org/doi/pdf/10.1145/3299869.3314049

[bib-vector-clock] https://en.wikipedia.org/wiki/Vector_clock

[bib-version-vector] https://en.wikipedia.org/wiki/Version_vector

[bib-dvv] https://riak.com/posts/technical/vector-clocks-revisited-part-2-dotted-version-vectors/index.xhtml

[bib-version-vectors-are-not-vector-clocks] https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/

[bib-riak-vector-clock] https://riak.com/posts/technical/vector-clocks-revisited/index.xhtml?p=9545.xhtml

[bib-sibling-explosion] https://docs.riak.com/riak/kv/2.2.3/learn/concepts/causal-context/index.xhtml#sibling-explosion

[bib-riak-conflict-resolver] https://docs.riak.com/riak/kv/latest/developing/usage/conflict-resolution/java/index.xhtml

[bib-mvcc] https://en.wikipedia.org/wiki/Multiversion_concurrency_control

[bib-lamport-clock] https://en.wikipedia.org/wiki/Lamport_timestamp

[bib-partial-order] https://en.wikipedia.org/wiki/Partially_ordered_set

[bib-aws-strong-consistency] https://aws.amazon.com/about-aws/whats-new/2020/12/amazon-s3-now-delivers-strong-read-after-write-consistency-automatically-for-all-applications/

[bib-hybrid-clock] https://cse.buffalo.edu/tech-reports/2014-04.pdf

[bib-dynamo] https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[bib-last-write-wins] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/globaltables_HowItWorks.xhtml

[bib-mongodb-hybridclock] https://www.mongodb.com/blog/post/transactions-background-part-4-the-global-logical-clock

[bib-mongodb] https://www.mongodb.com/

[bib-mongodb-causal-consistency] https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns/

[bib-voldemort] https://www.project-voldemort.com/voldemort/

[bib-riak] https://riak.com/posts/technical/vector-clocks-revisited/index.xhtml?p=9545.xhtml

[bib-causal-consistency] https://jepsen.io/consistency/models/causal

[bib-cockroachdb-timestamp-cache] https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer.xhtml#timestamp-cache

[bib-cockroachdb-follower-read] https://www.cockroachlabs.com/docs/v20.2/follower-reads.xhtml

[bib-snapshot-isolation] https://jepsen.io/consistency/models/snapshot-isolation

[bib-transaction-isolation] https://en.wikipedia.org/wiki/Isolation_(database_systems)

[bib-rocksdb] https://rocksdb.org/docs/getting-started.xhtml

[bib-pebble] https://github.com/cockroachdb/pebble

[bib-boltdb] https://github.com/etcd-io/bbolt#using-keyvalue-pairs

[bib-two-phase-commit] https://en.wikipedia.org/wiki/Two-phase_commit_protocol

[bib-state-machine-replication] https://en.wikipedia.org/wiki/State_machine_replication

[bib-laslie-lamport] https://en.wikipedia.org/wiki/Leslie_Lamport [bib-time-clocks-ordering] https://lamport.azurewebsites.net/pubs/time-clocks.pdf

[bib-spanner] https://cloud.google.com/spanner

[bib-neo4j] https://neo4j.com/docs/operations-manual/current/clustering/

[bib-neo4j-causal-cluster] https://neo4j.com/docs/operations-manual/current/clustering-advanced/lifecycle/#causal-clustering-lifecycle

[bib-moving-average] https://en.wikipedia.org/wiki/Moving_average

[bib-mongodb-max-staleness] https://docs.mongodb.com/manual/core/read-preference-staleness/#std-label-replica-set-read-preference-max-staleness

[bib-flp-impossibility] https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf

[bib-gryadka-comparison] http://rystsov.info/2017/02/15/simple-consensus.xhtml

[bib-gryadka] https://github.com/gryadka/js

[bib-cosmosdb] https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

[bib-byzantine-fault] https://en.wikipedia.org/wiki/Byzantine_fault

[bib-pbft] http://pmg.csail.mit.edu/papers/osdi99.pdf

[bib-blockchain] https://en.wikipedia.org/wiki/Blockchain

[bib-crash-fault] https://en.wikipedia.org/wiki/Crash_(computing)

[bib-epaxos] https://www.cs.cmu.edu/~dga/papers/epaxos-sosp2013.pdf

[bib-event-sourcing] https://martinfowler.com/eaaDev/EventSourcing.xhtml [bib-two-phase-locking] https://en.wikipedia.org/wiki/Two-phase_locking

[bib-wound-wait] http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/deadlock-woundwait.xhtml

[bib-wait-die] http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/deadlock-waitdie.xhtml

[bib-comparing-wait-die-and-wound-wait] http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/8-recv+serial/deadlock-compare.xhtml

[bib-external-consistency] https://cloud.google.com/spanner/docs/true-time-external-consistency

[bib-percolator] https://research.google/pubs/pub36726/

[bib-tikv] https://tikv.org/

[bib-multi-raft] https://www.cockroachlabs.com/blog/scaling-raft/

[bib-spanner-concurrency] https://dahliamalkhi.github.io/files/SpannerExplained-SIGACT2013b.pdf

[bib-XA] https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

[bib-transactional-outbox] https://microservices.io/patterns/data/transactional-outbox.xhtml

[bib-activemq-slow-restart] https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/recover-xa-transactions.xhtml

[bib-flexible-paxos] https://arxiv.org/abs/1608.06696

[bib-akka-cluster-splitbrain] https://github.com/akka/akka/issues/18554 [bib-akka-gossip-convergence] https://doc.akka.io/docs/akka/current/typed/cluster-concepts.xhtml#gossip-convergence

[bib-aws-time-sync-service] https://aws.amazon.com/about-aws/whats-new/2021/11/amazon-time-sync-service-generate-compare-timestamps/

[bib-ydb-causal-reverse] https://docs.yugabyte.com/latest/benchmark/resilience/jepsen-testing-ysql/#rare-occurrence-of-causal-reversal

[bib-cdb-causal-reverse] https://www.cockroachlabs.com/blog/consistency-model/

[bib-aws-clock-accuracy] https://aws.amazon.com/blogs/mt/manage-amazon-ec2-instance-clock-accuracy-using-amazon-time-sync-service-and-amazon-cloudwatch-part-1/

[bib-yugabyte] https://www.yugabyte.com/

[bib-clock-bound] https://github.com/aws/clock-bound

[bib-yugabyte-leader-lease] https://blog.yugabyte.com/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/

[bib-hazelcast] https://hazelcast.com/

[bib-jgroups] http://www.jgroups.org/

[bib-ignite] https://ignite.apache.org/docs/latest/

[bib-ignite-partitioning] https://ignite.apache.org/docs/latest/data-modeling/data-partitioning/

[bib-hazelcast-partitioning] https://docs.hazelcast.com/imdg/4.2/overview/data-partitioning [bib-akka-shard-allocation] https://doc.akka.io/docs/akka/current/typed/cluster-sharding.xhtml#shard-allocation

[bib-kafka-metadata-issue] https://issues.apache.org/jira/browse/KAFKA-901

[bib-yb-metadata-issue] https://gist.github.com/jrudolph/be4e04a776414ce07de6019ccb0d3e42

[bib-yb-rocksdb-enhancements] https://blog.yugabyte.com/enhancing-rocksdb-for-speed-scale/

[bib-yb-range-vs-tablets] https://blog.yugabyte.com/yugabytedb-vs-cockroachdb-bringing-truth-to-performance-benchmark-claims-part-2/#ranges_vs_tablets

[bib-yb-tablet-splitting-issue] https://github.com/yugabyte/yugabyte-db/issues/1004

[bib-hbase] https://hbase.apache.org/

[bib-yb] https://www.yugabyte.com/

[bib-yb-automatic-table-splitting] https://github.com/yugabyte/yugabyte-db/blob/master/architecture/design/docdb-automatic-tablet-splitting.md

[bib-cockroach-load-splitting] https://www.cockroachlabs.com/docs/stable/load-based-splitting.xhtml

[bib-yb-load-splitting] https://github.com/yugabyte/yugabyte-db/issues/1463

[bib-gossip-convergence] https://doc.akka.io/docs/akka/current/typed/cluster-concepts.xhtml#gossip-convergence [bib-akka-auto-downing] https://doc.akka.io/docs/akka/2.5/cluster-usage.xhtml#auto-downing-do-not-use-

[bib-akka-split-brain-resolver] https://doc.akka.io/docs/akka-enhancements/current/split-brain-resolver.xhtml

[bib-hazelcast-split-brain-time-window] https://docs.hazelcast.com/imdg/4.2/network-partitioning/split-brain-protection#time-window-for-split-brain-protection

[bib-tcp-nagel] https://en.wikipedia.org/wiki/Nagle%27s_algorithm

[bib-correlation-id] https://www.enterpriseintegrationpatterns.com/CorrelationIdentifier.xhtml

[bib-kafka-purgatory] https://www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels/

[bib-etcd-wait] https://github.com/etcd-io/etcd/blob/main/pkg/wait/wait.go

[bib-jgroups-discovery-protocols] https://docs.jboss.org/jbossas/docs/Clustering_Guide/beta422/html/jbosscache-jgroups-discovery.xhtml

[bib-akka-discovery-protocols] https://doc.akka.io/docs/akka-management/current/bootstrap/index.xhtml

[bib-jeaf-dean-google-talk] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf

[bib-cassandra-cep-21] https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-21%3A+Transactional+Cluster+Metadata

[bib-distrib-algorithms-nancy-lynch] https://www.elsevier.com/books/distributed-algorithms/lynch/978-1-55860-348-6 [bib-intensive-data-book] https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[bib-microservices] https://martinfowler.com/articles/microservices.xhtml

[bib-ddd] https://martinfowler.com/bliki/DomainDrivenDesign.xhtml

[bib-sticky-sessions] https://docs.aws.amazon.com/elasticloadbalancing/latest/application/sticky-sessions.xhtml

[bib-nosql] https://martinfowler.com/books/nosql.xhtml

[bib-gang_of_four] https://martinfowler.com/bliki/GangOfFour.xhtml

[bib-pulsar] https://pulsar.apache.org/

[bib-rocksdb-sequence-number] https://github.com/cockroachdb/pebble/blob/master/docs/rocksdb.md#internal-keys

[bib-go-lang] https://go.dev/

[bib-2-hard-dist] https://twitter.com/mathiasverraes/status/632260618599403520

[bib-etcd-read-issue] https://github.com/etcd-io/etcd/issues/741

[bib-jepsen-etcd-consul] https://aphyr.com/posts/316-jepsen-etcd-and-consul

[bib-leader-lease] https://web.stanford.edu/class/cs240/readings/89-leases.pdf

[bib-yugabytedb-leader-lease] https://www.yugabyte.com/blog/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/

[bib-consul-leader-lease-issue] https://github.com/hashicorp/raft/issues/108 [bib-paxos-quorum-lease] https://www.cs.cmu.edu/~dga/papers/leases-socc2014.pdf

[bib-paxos-quorum-read] https://www.usenix.org/system/files/hotstorage19-paper-charapko.pdf

[bib-consul-leader-lease] https://gist.github.com/armon/11059431

[bib-linux-clock-gettime] https://linux.die.net/man/2/clock_gettime

[bib-clock-drift] https://en.wikipedia.org/wiki/Clock_drift

[bib-consistent-hashing] https://en.wikipedia.org/wiki/Consistent_hashing

[bib-cassandra-vnode] https://www.datastax.com/blog/virtual-nodes-cassandra-12

[bib-rendezvous_hashing] https://en.wikipedia.org/wiki/Rendezvous_hashing

[bib-cassandra-improve-vnode-allocation] https://issues.apache.org/jira/browse/CASSANDRA-7032