SUTD Spring 2022 - 50.041 Distributed Systems

Final Project Report

Group 1

Toh Kai Feng	1004581
Amrish Dev Sandhu	1004241
Tan Xin Yi	1004226
Goh Shao Cong Shawn	1004116
Lim Jun Wei	1004379

Prepared for: Professor Sudipta Chattopadhyay

1 Introduction

E-commerce is on the rise, fuelled by COVID-19 (*Global E-Commerce Jumps to $26.7 Trillion, Fuelled by COVID-19*, 2021).

As such, there is a rising need for a highly available data-store to serve the increasing number of online customers. Amazon as a leader in e-commerce has implemented distributed systems solutions that enable their services to be always available (millions of customers can always add an item into their shopping cart at any time even amidst network and server failure) to maintain customer trust. This means reliability on a large scale with highly decentralised, service oriented architecture consisting of hundreds of services.

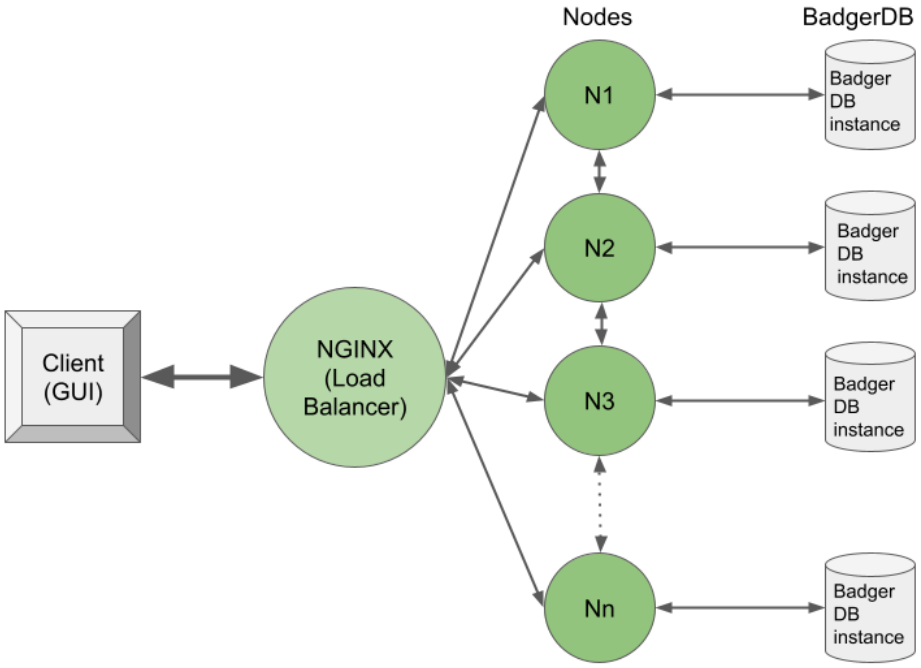
With e-commerce platforms becoming increasingly competitive and supply chain shortages occurring around the world, maintaining a system that is ever reliable that allows customers to accurately add and delete items to their cart and see these changes reflected accurately is important to the user experience. Companies have to ensure that their systems are reliable to avoid financial consequences and losing customer trust, and also have to be highly scalable to feed the increased web demand.

After our exploration of the various project topics, we are inspired to dive deep into the exploration of Amazon's implementation of the Dynamo solution in its e-commerce platform by designing our own simplified e-commerce cart system that would like to provide **availability (highly available writes), reliability (eventual consistency)** and **scalability (cheaper maintenance).**

2 System Overview

To implement our distributed system, we introduce 3 key components: nodes, a load balancer and databases for persistent storage. As shown in the following diagram, clients will contact the Nginx load balancer to send requests to our distributed database. The load balancer will select a random node to handle the request based on the peer-to-peer protocol we have implemented. Data will then be stored at or retrieved from the local BadgerDB databases at each node. Responses will be returned to the clients through the load balancer.

To demonstrate one possible use case for our system, a shopping cart GUI application was created with functionalities of adding a new user, viewing the user's cart, adding items to the user's cart and removing items from the user's cart. These functionalities enable us to show the capabilities of our distributed database.

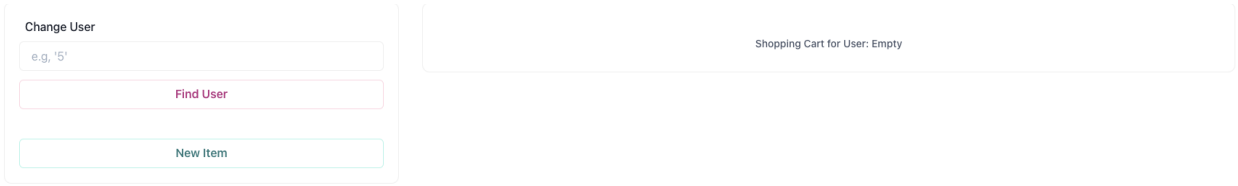


Overall System Architecture

2.1 Front-end

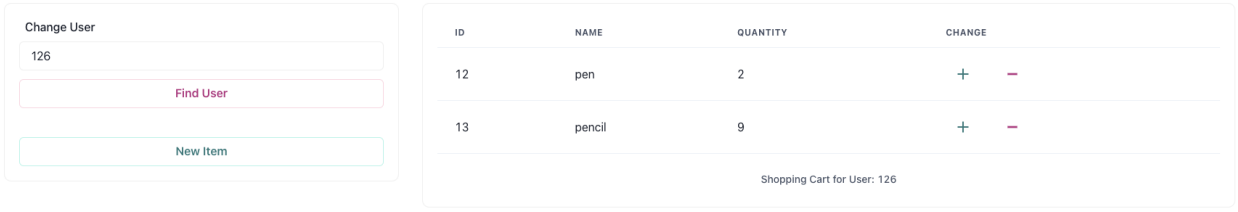
To demonstrate our distributed systems, we created a front-end GUI that mimicked an ecommerce cart application. We utilised the React Framework to build a cart GUI that allows a user to query for a particular user's cart before adding items to that cart and incrementing or decrementing the quantity of items within the cart. We also utilised the inbuilt Fetch API to perform GET and POST requests to the back-end. Below are some screenshots of our front-end illustrating the process of starting a cart session, adding an item to the cart and then incrementing, decrementing or deleting an item from the cart.

Step 1: When the user first enters the webpage, he is greeted with the landing page as shown below:



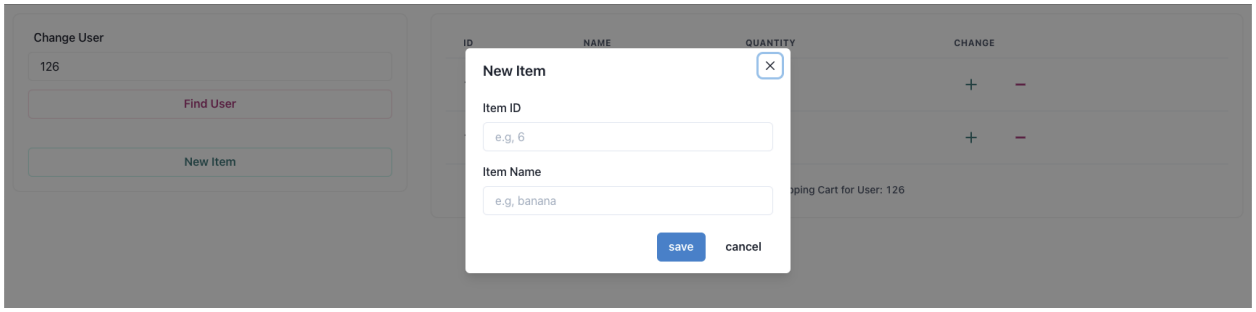
Landing Page of Cart Front-End

Step 2: To view his cart, he would have to type in his userID and then click the "Find User" button. When he does so, the load balancer will forward the read request to the responsible nodes which will perform a read operation of the items associated with the particular userID. This is displayed within the table on the right, showcasing the item ID, item name and item quantity associated with the userID.

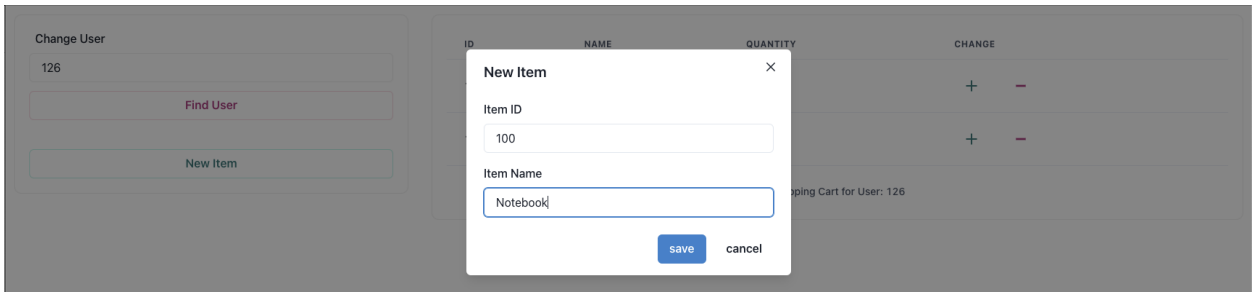


UserID read request to retrieve cart contents

Step 3: When the user wants to add an item to the cart, he would click on the "New Item" button and input an item ID (numerical digits only) and the item name. Selecting on save would execute a write request which the load balancer would forward to the responsible node to then perform a write operation of the new item associated with the userID.

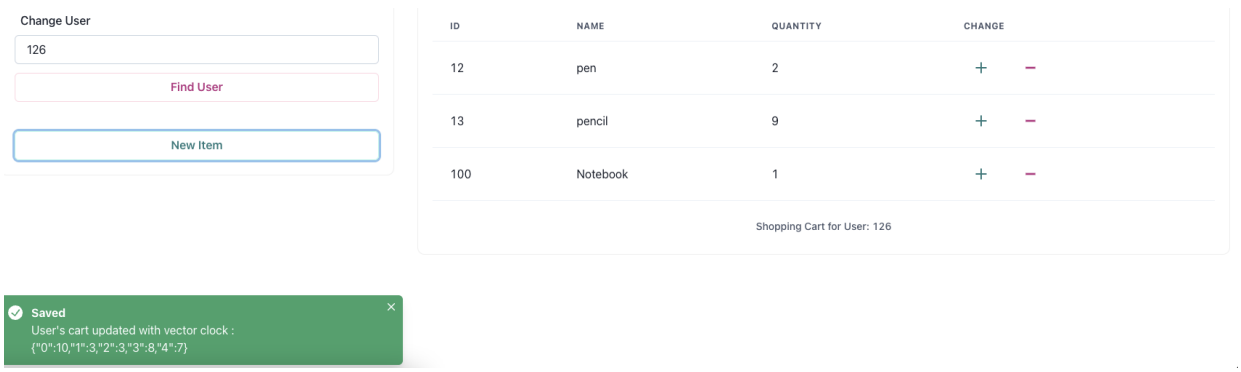


New Item addition to cart part 1



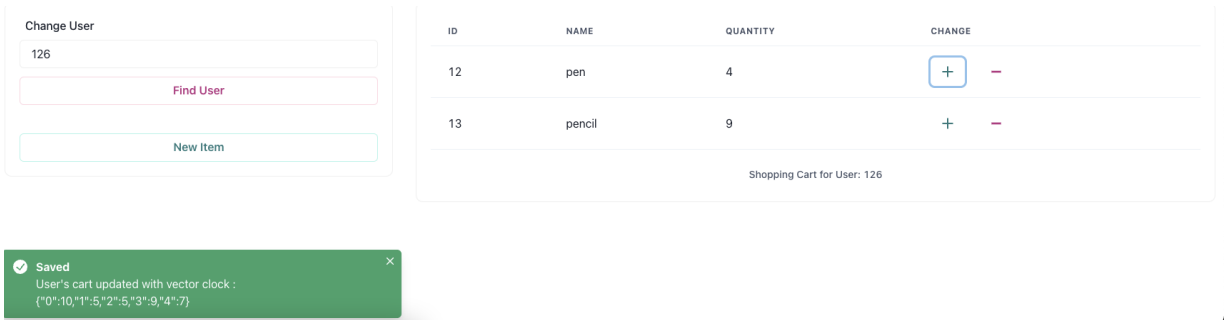
New Item addition to cart part 2

The database would then return a response message that indicates whether the write operation was successful or not with its corresponding vector clock and the item entry will be appended to the table displayed.



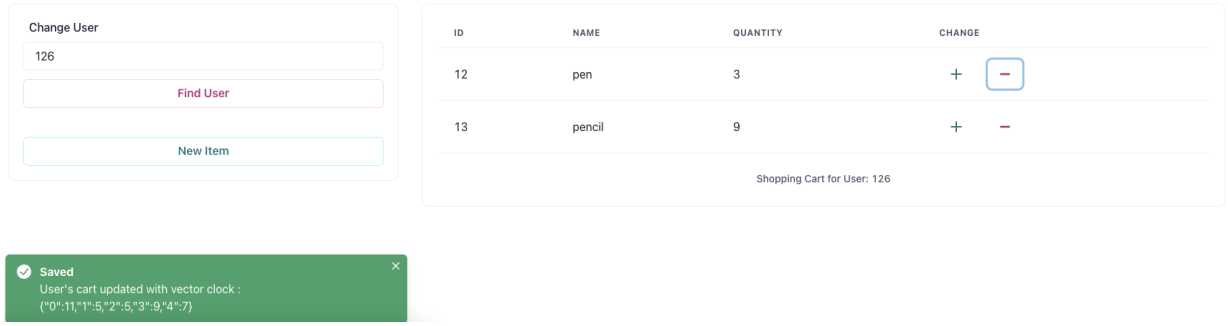
Successful addition of new item to cart

Step 4: When the user wants to increment the quantity of an item in the cart, he would click on the “+” button beside the item quantity under the change column. This would execute a write request which the load balancer would forward to the responsible node to then perform a write operation of incrementing the item quantity by the number of clicks associated with the userID.



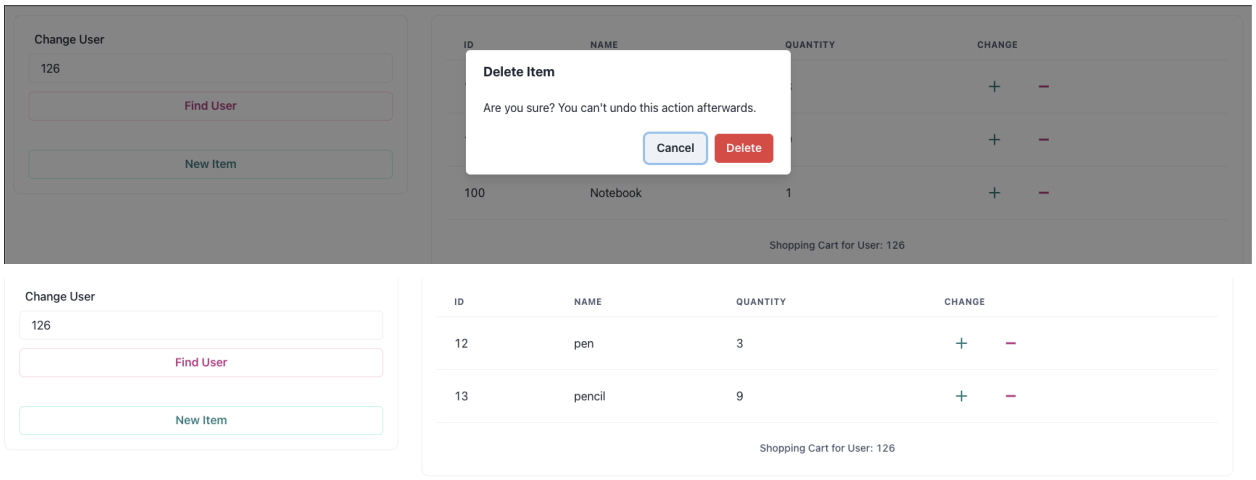
Successful increment of item quantity in cart

Step 5: When the user wants to decrease the quantity of an item in the cart, he would click on the "-" button beside the item quantity under the change column. This would execute a write request which the load balancer would forward to the responsible node to then perform a write operation of decrementing the item quantity by the number of clicks associated with the userID.



Successful decrement of item quantity in cart

Step 6: When the user wants to delete an item in the cart, he would click on the "-" button beside the item quantity under the change column when the quantity displayed is "1". This would execute a write request which the load balancer would forward to the responsible node to then perform a write operation of decrementing the item quantity to 0, performing a deletion of that item associated with the userID.



Successful deletion of item in cart

2.2 Back-end

In our distributed system, we modelled our system structure after a logical ring with 100 positions. Nodes will be positioned along this ring and consistent hashing will be used to identify where each key will be stored. To ensure availability, our system replicates the data across multiple nodes (with a replication factor of 3 defined in our protocol), but does not require all nodes to complete their replication before responding that the operation was a success (with a minimum successful write count of 2 defined in our protocol)..

Local data persistence is provided at each node using the BadgerDB key-value store. These databases are initialised whenever a node joins the logical ring structure. Other functionalities will be explained in the sections that follow this.

2.3 System Assumptions

To simplify the implementation of our distributed system, our team made the following assumptions.

1. **Similar capacities of machines**: This assumption is necessary so that we do not need to consider heterogeneity of machines within our distributed system.
2. **Transient node failures**: Given that we would only be handling temporary failures instead of both temporary failure and permanent failures due to resource and time constraints, we assume that node failures are transient and that there are no permanent failures of nodes.
3. **Operation Requirement**: Our distributed system requires at least 3 nodes to operate due to the replication factor being 3.
4. **Broadcasting ability**: In our distributed system, every node can be selected as a coordinator by the load balancer and as such all nodes can broadcast to all other nodes.
5. **No complete data loss**: Due to the earlier assumption that there is no permanent failure, any replica that is handed hinted data would not fail permanently. This ensures that hinted replicas are not lost and nodes will eventually get back the data that was supposed to be originally written.
6. **Non-malicious environment**: A key assumption within DynamoDB is that their distributed system is utilised by internal services and that the operation environment is assumed to be non-malicious. With that assumption, we can also assume that all nodes in our distributed system are non-malicious too.

2.4 System Components

The following table describes the key function and responsibilities of each component of our system.

System Components	Function/Responsibilities
Front-End Service (ReactJS)	Acting as a non-malicious service that is using our implementation of DynamoDB, this front-end service models a typical e-commerce “cart” service that users can modify and edit. The service allows for the access of a user’s cart as well as the addition, modification and deletion of items from the user’s cart. The service will also perform conflict resolution in place of the client should there be cart versioning conflicts.
Load Balancer(Nginx)	Utilising Nginx, the load balancer is responsible for distributing read and write requests to a random node as a coordinator within the ring structure which would handle the read and write request before contacting the responsible node to perform the read and write request.
Nodes	Nodes play a vital role in ensuring the correctness, scalability and availability of our system. While all nodes are similar with regards to their capabilities and responsibilities, nodes can, at times, play different roles. **Coordinator** To prevent a single bottleneck from occurring, the random node that the load balancer selects to handle client requests acts as a coordinator node. Since our system uses replication to ensure availability, as a coordinator, the node must identify all responsible storage nodes for the given key and forward the client requests to these nodes. This node must also account for the responses from each of these storage nodes and handle failure cases when responses are not received. **Storage** These storage nodes are in charge of communicating with their local instances of BadgerDB to persist the received data or to retrieve data for a given key.
Persistent Store (BadgerDB)	We used the Badger library as a key-value store to handle read and write requests within each node. The Badger library would ensure that objects persist in the disk (instead of memory) so that data would not be completely lost when nodes crash.

2.5 Features

To ensure that our distributed system guarantees correctness, scalability and fault tolerance, our team has implemented the following features.

- Consistent Hashing
- Joining
- Replication
- Data Versioning using vector clocks
- Hinted Handoff

In the table below, we associate the implemented features with the principle with which we strived to implement in our distributed system.

Principle	**Rationale**	**Features**
Scalability	One single server would eventually have insufficient bandwidth and storage. Thus the architecture of our system should be designed to manage incrementally increasing load and storage given rising user traffic.	Consistent Hashing Joining
High Write Availability	We want users to view updated carts and items to be added/deleted from their carts even amidst server failures. This would present a site that is “always functioning” and present carts that contained the updated items from the previous session.	Consistent Hashing Joining Hinted Handoff Replication
Consistency / Eventual Consistency	It is important that the data stored in all servers reflect the same changes made by the user so that even viewed on different devices or clients, they are up to date. However, some relaxation to our system’s consistency (allowing for eventual consistency) can be made since higher priority is instead given to our system’s availability.	Data versioning using vector clocks
Fault Tolerance (Temporary Failure)	Assuming that temporary failure of machines can occur, we need to design a system that is tolerant of such failures. This improves the availability of the system, and reduces the likelihood of permanent data loss.	Replication Hinted handoff

The following subsections provide further explanations regarding the protocol design and implementation of each listed feature.

2.5.1 Consistent Hashing

Consistent hashing distributes the write requests such that the nodes within the system have an even share of “keys” they are responsible for when servicing read and write requests.

We have set an arbitrary but configurable number of 100 positions on the ring structure that can be occupied by any node. That means that we can have up to 100 nodes in our set up, but the value of “100” can be configured to a higher value for a system that requires more than 100 nodes due to higher traffic.

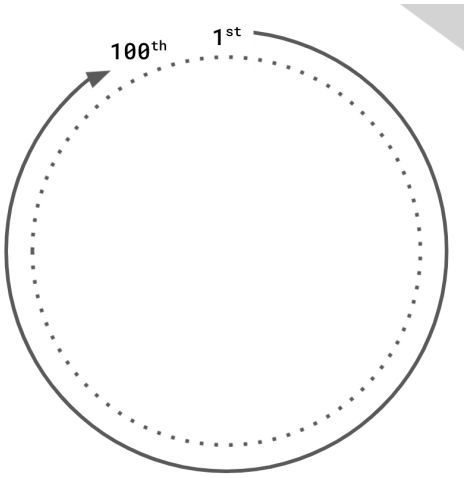


Illustration of the logical ring structure

When a read or write request is being made for a key-value pair, a random node is chosen to become the “Coordinator” to service the request. The coordinator thus uses an MD5 hashing algorithm on the key to generate an integer value, say X. We would then perform X modulo 100, to determine the position Y on the ring (where 0 <= Y < 100). As a replication factor of 3 is used, the 3 nodes clockwise of Y will be selected as the nodes responsible for this key, and thus the coordinator forwards the write request to those 3 nodes.

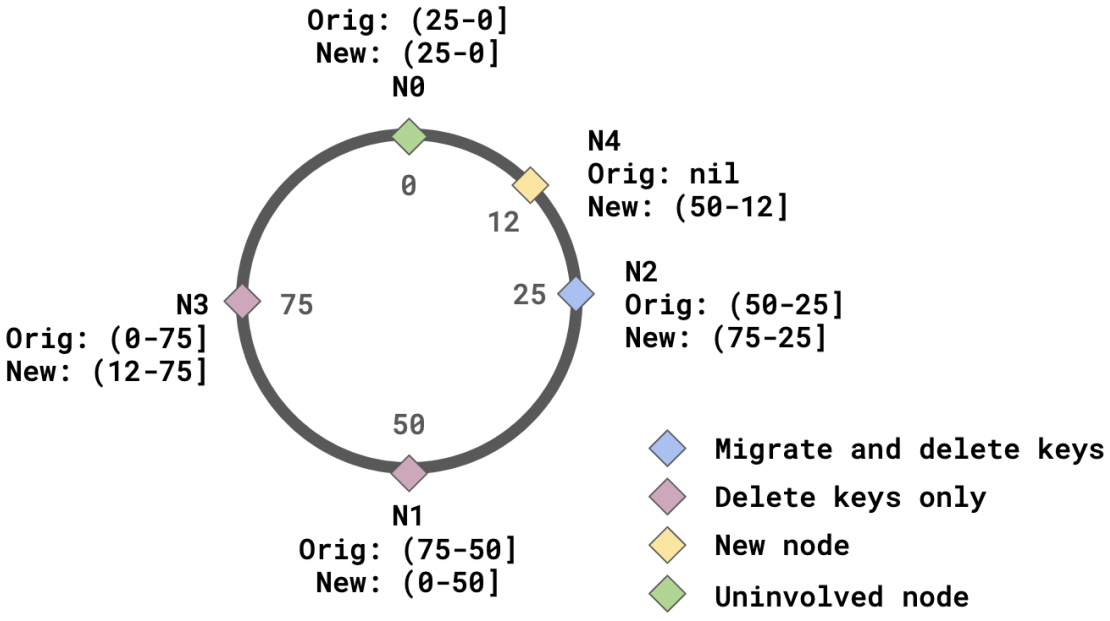
This mechanism requires at most 4 nodes (coordinator + 3 responsible nodes) to service any read or write request. If keys are evenly distributed around the ring, this will ensure that all nodes get an even distribution of requests.

2.5.2 Joining

To discuss the joining protocol, we assume that the system already contains the minimum number of nodes required, which is equivalent to the replication factor. This is because any lower number of nodes will not allow the system to function, thus implying that no data will be stored in the nodes.

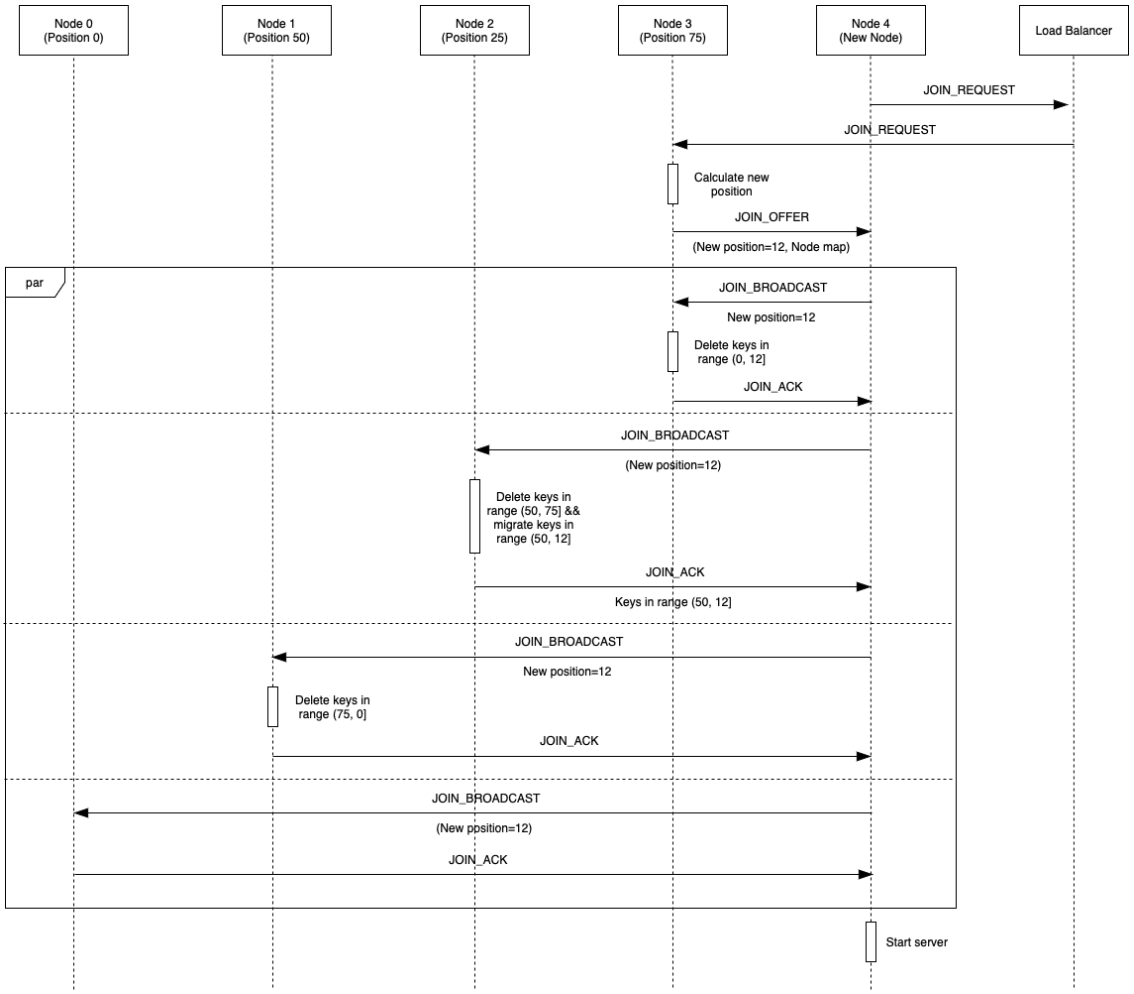
When a new node joins, it will send a JOIN_REQUEST message to the load balancer as it will not be aware of the existing nodes in the system. The load balancer will forward this message to a random node in the system. This node will calculate the position for the new node using its record of the current ring structure. This position will be chosen as the middle position of the largest gap between two nodes in the ring. Once the position is calculated, the position and a copy of the ring structure will be sent back to the new node.

Upon receiving the ring structure and its position, this new node will send a JOIN_BROADCAST message containing its new position directly to all existing nodes in the system. All nodes will update their records of the ring structure to include this new node and respond with an acknowledgement. Additionally, the immediate successor of the new node will identify the set of keys that this new node should be responsible for and send these keys together with the acknowledgement. The three immediate successors of the new node will also delete the keys that they no longer need to be responsible for. From the example in the following diagram, we see that when N4 joins at position 12, its immediate successor, N2, holds all the keys that N4 will be responsible for and can send them to N4. Additionally, we see that N2, N1 and N3 can delete the key ranges (50,75], (75,0] and (0,12] respectively.



Migration and deletion of key ranges when N4 joins the system

Once the new node receives an acknowledgement from all nodes, it will begin listening on its specified port and start servicing requests. The process described above is depicted in the following sequence diagram.



Sequence diagram of the joining protocol

2.5.3 Replication

Our system uses a replication factor of N=3, minimum successful reads of R=2 and minimum successful writes of W=2, which are standard values recommended by Dynamo. This means that for any given write request, the key-value pair would be stored on 3 responsible nodes using the consistent hashing method we have defined in 2.5.1. Furthermore, a read or write request is successful if 2 out of 3 nodes have successfully read from/written into their databases.

Replication provides fault tolerance since read and writes can continue to be serviced even when responsible nodes are down. For any given read request, the assigned “coordinator” would still successfully respond (without error) to the client if a node (storing 1 of the 3 replicas) is down. Conversely, for any given write request, the assigned coordinator would still successfully respond to the client as long as there are still 2 nodes alive in the entire system. This provides the high write availability that Dynamo aims to achieve. Additionally, by setting our minimum successful reads and writes to 2 nodes, we aim to reduce latencies even in the absence of node failures since requests can still be fulfilled when 1 of the 3 responsible nodes is slow to respond.

2.5.4 Data Versioning (utilising Vector Clocks)

Our system uses eventual consistency to ensure the correctness of the data across multiple replicas. To guarantee high write availability, writes can be executed in any order on the storage nodes depending on which requests were received first. This allows concurrent requests to be made but simultaneous attempts to write to an object could result in conflicting versions. To handle this problem, data versioning is implemented using vector clocks to detect conflicting versions. Upon detection of possible conflicts, instead of handling conflict resolution internally, conflicts are raised to the clients for them to select the version that they want to keep. Different (potentially conflicting) versions can be resolved when a client sends a read request to retrieve the different versions of an object created. Thereafter, by sending a write request to the same object using a vector clock strictly greater than all versions of the object retrieved, nodes containing multiple versions of the object would keep only the latest version written by the client.

To detect conflicts, each key contains its own vector clock which is initialised when the key is created. The number of elements within the clock corresponds to the number of nodes in the system. When a coordinator handles a write request for the key, the vector clock element corresponding to the id of the coordinator will be incremented. When data is stored in the storage nodes, the vector clock of the new object will be compared with all existing versions in the database. A conflict is detected if the vector clock of the incoming object is not strictly greater than the current version(s) stored within a node.

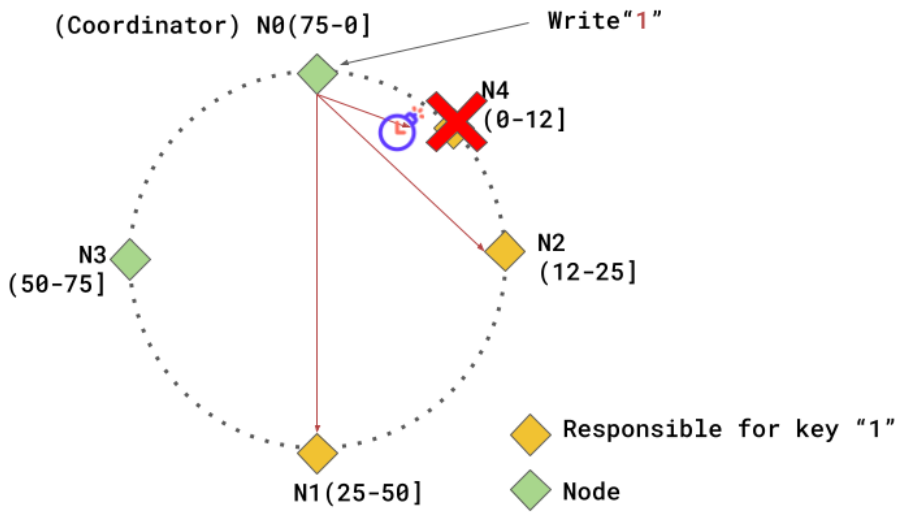
Further details regarding the implementation and test cases for data versioning will be covered in Section 3.2.1.

2.5.5 Hinted Handoff

Hinted Handoff is a technique that we have implemented in our distributed system to ensure that read and write operations are not failed due to temporary node failures in the system.

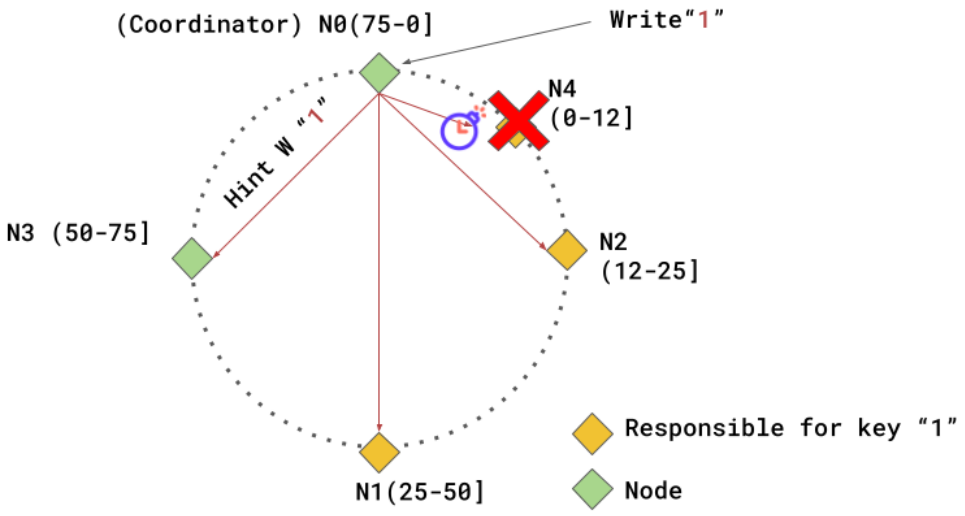
In the following case, we illustrate how we have implemented hinted handoff within our system. Further cases are illustrated under Section 3.3.

Assuming that Node 0 is the coordinator and is handling an incoming write request for key “1”, it would contact three nodes to be responsible, which are in this case Node 4, Node 2 and Node 1. However, given that Node 4 is temporarily down, Node 0 will not receive a response from Node 4 and will timeout, indicating that Node 4 is unavailable.



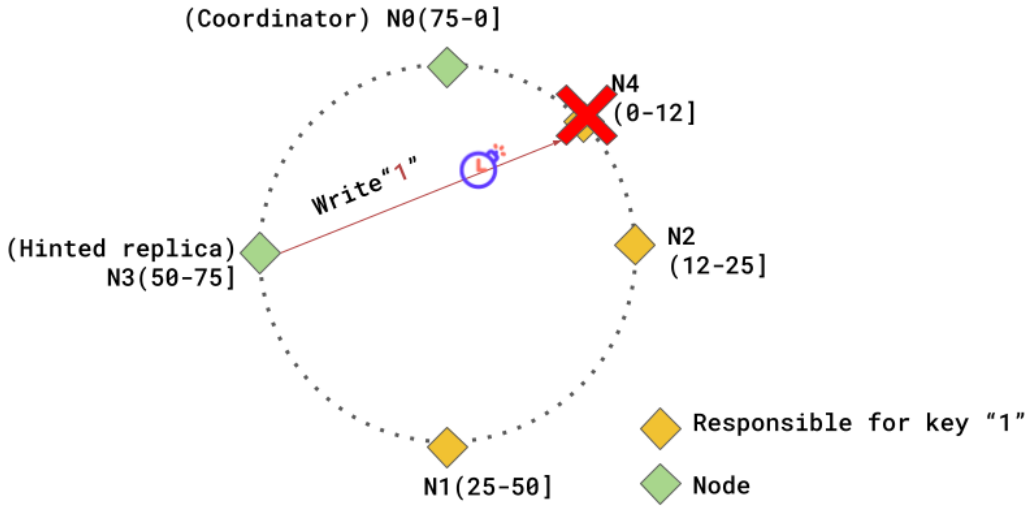
Regular write request sent to N4, N2 and N1, with N4 being unavailable

Noting a timeout, Node 0 would then send a hinted write request to Node 3 which is the successor node in the ring structure. The hinted write request would contain the key and data that was to initially be written to Node 4 and a hint that Node 4 was the intended recipient.



Hinted replica sent to N3

Node 3 would then attempt to hand over the data to Node 4 by periodically pinging Node 4 with the data. This handover attempt will continue until Node 4 revives and responses with a success status code, at which, Node 3 will end the hinted handoff process and delete the hinted replica from its storage.



Periodic attempts to return hinted replica to N4 by N3

3 System Design

3.1 Scalability

3.1.1 Considerations

The scalability of our system is enabled by (1) the ability to add new nodes in the system, and (2) the ability to evenly distribute read and write requests to the nodes in the system.

3.1.1.1 Joining Protocol

The joining protocol described in Section 2.5.2 enables more nodes to be added to our system to handle traffic spikes and support high loads. By including more nodes in the system, keys will be more spread out, thus reducing the number of requests that each node will have to deal with.

Additionally, the joining protocol is designed in such a way as to encourage a more even spread of keys among the nodes. This is done by identifying the biggest gap between two nodes in the existing ring structure and adding the new node at the middle of this gap. Doing so ensures that no single node will have to be responsible for an unproportionally large region on the ring. Instead, nodes will be kept evenly spaced, thus making sure that requests are distributed fairly across all nodes.

3.1.1.2 Load Balancing

In our system, all nodes can act as coordinators to handle requests sent from clients. The use of a load balancer to distribute these requests helps to ensure a more even distribution of requests to each node such that no single node becomes a bottleneck. This also means that when there are more nodes in the system, more client requests can be serviced at any single time.

3.1.2 Experiments

3.1.2.1 Experiment Setup

Load tests were conducted using a script written with the Vegeta library. Requests were sent at a steady rate for 1min from a single attacking host. The Vegeta attacker, Nginx load balancer and Dynamo nodes were all run on the same machine. For each test, the attacking host will send a random sequence of keys and ClientCart objects to the load balancer and the requests will be forwarded to our nodes.

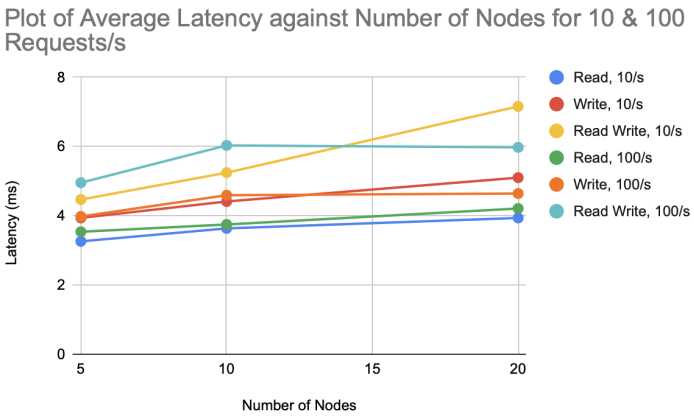
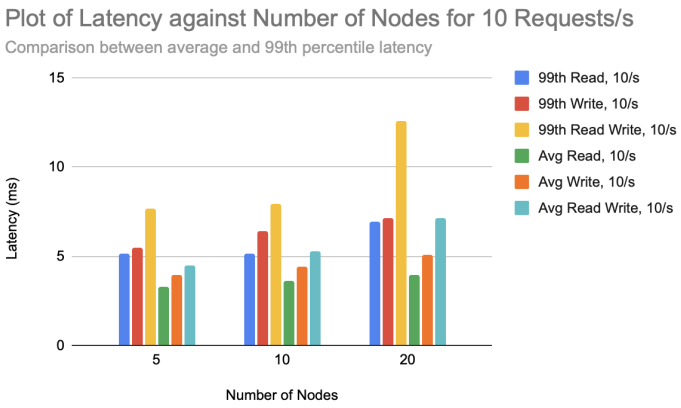
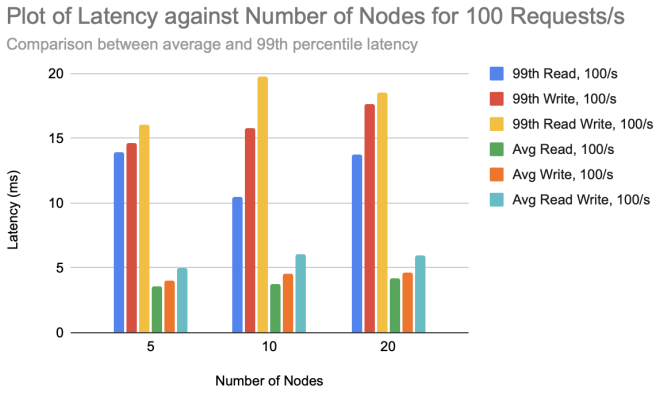
For this experiment, we varied the number of nodes and the rate of requests. For each combination of node number and request rate, we also experimented with sending only read requests, only write requests and an equal mix of concurrent read and write requests. For all the tests, 3 metrics were recorded: success rate, average latency and the 99th percentile latency. Each test case was repeated 3 times and the average of the 3 latencies and success rates were taken.

The number of nodes used were 5, 10 and 20. While we attempted to run tests with more nodes, resource limitations of running on a single machine made it infeasible. Hence, only these 3 cases were used.

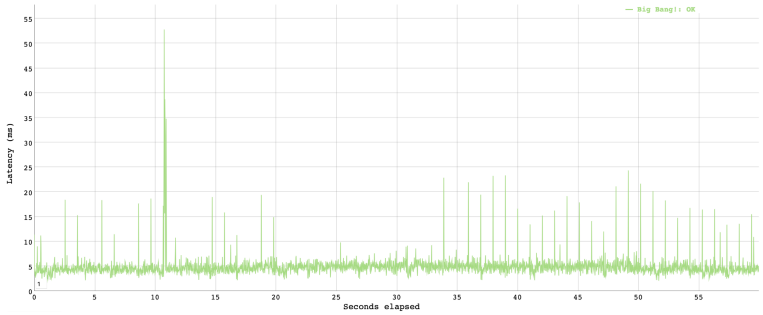
The request rates used were 10, 100 and 500 requests per second, meaning that a total of 600, 6000 and 30000 requests were sent to our nodes during each respective test. Similarly, while we attempted to increase the request rate further, we were limited by the resources of a single machine.

3.1.2.2 Experiment Results

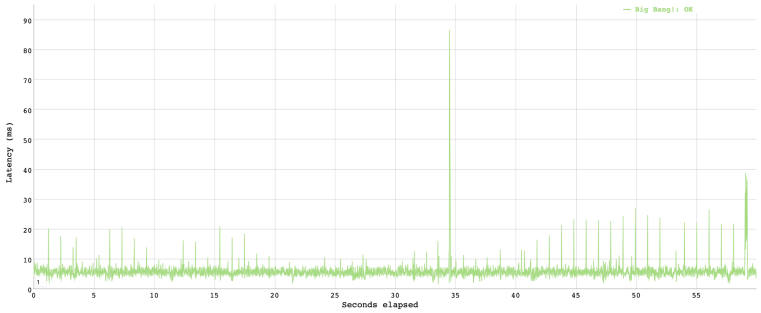
The following graphs show the results of the tests. The success rate is not plotted in the graphs as all tests completed with 100% success rates. A table breaking down the exact latencies for each test case can be found in Table A1 of Appendix A.

Plots	Explanations
	At 10 and 100 requests/s, the average latency of requests remained around the same and even increased a little when the number of nodes increased. This is expected as the average latency was approximately 3-6ms, meaning that at 10-100 requests/s, the system was not being stressed. It becomes more likely that the randomness of the queries sent and the fact that there may be resource limits when running all nodes on a single machine affected the recorded latencies more.
 	At 10 requests/s, the average latency was extremely close to the 99th percentile latencies of requests for all tests, with most differences being less than 5ms. At 100 requests/s, there was a more prominent difference in the average and 99th percentile latencies of requests, though still not very significant as the differences were generally around 10-15ms.

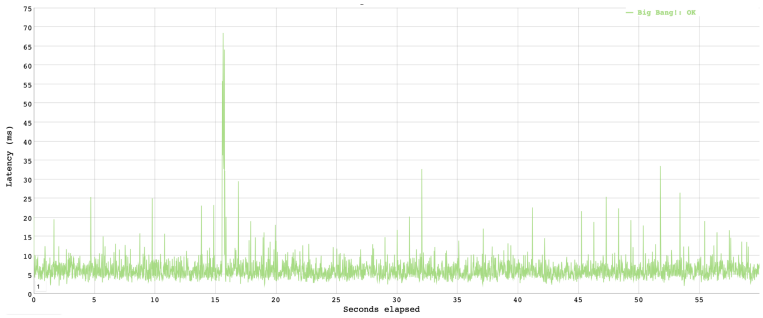
Distribution of request latencies in 1 test with 5 nodes and 100 requests/s



Distribution of request latencies in 1 test with 10 nodes and 100 requests/s

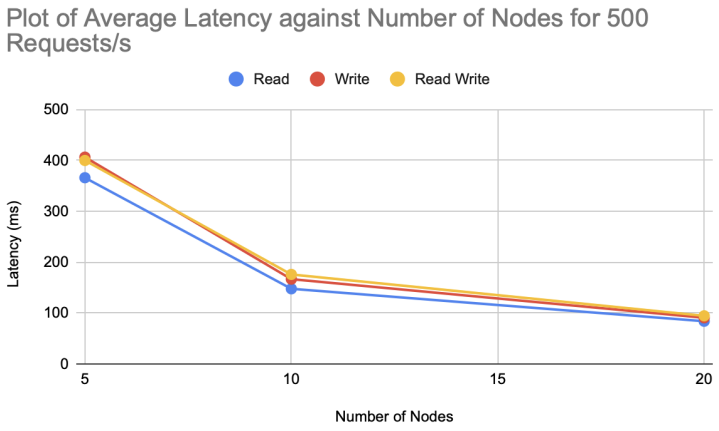


Distribution of request latencies in 1 test with 20 nodes and 100 requests/s

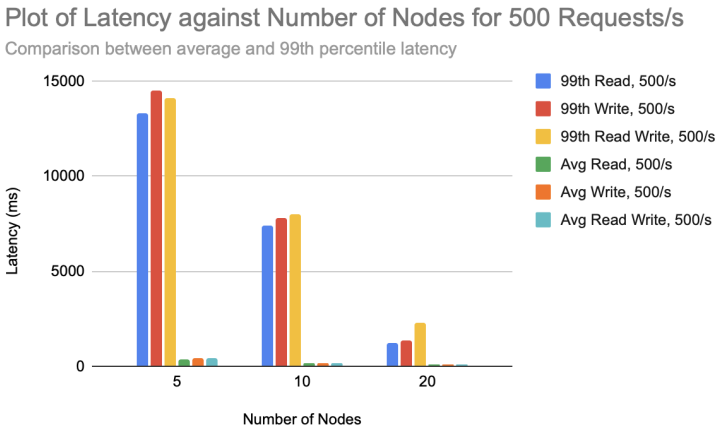


The above results can be explained when we look at these plots that show the distribution of request latencies throughout the entire test duration. These plots compare the distribution across different numbers of nodes at a request rate of 100 requests/s. We see that most requests get fulfilled within 5-10ms, with some taller spikes appearing occasionally, which likely account for the 1% of requests with the highest latencies.

We also see that at 100 requests/s, the case with 5 nodes already does considerably well and that increasing the number of nodes does not change the pattern of the latency distribution, thus explaining why there is little difference between the average latencies across different numbers of nodes recorded in the earlier graph.

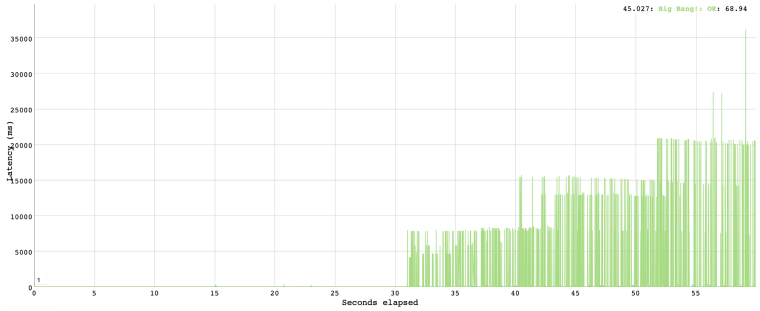


At 500 requests/s, we see the expected trend of the average latency of requests decreasing as the number of nodes increases, from 400ms with 5 nodes to 100ms with 20 nodes. This trend makes sense as the rate of 500 requests/s is sufficient to stress the system. In such a scenario, having more nodes can distribute both the keys and the requests more evenly such that each node handles fewer requests.

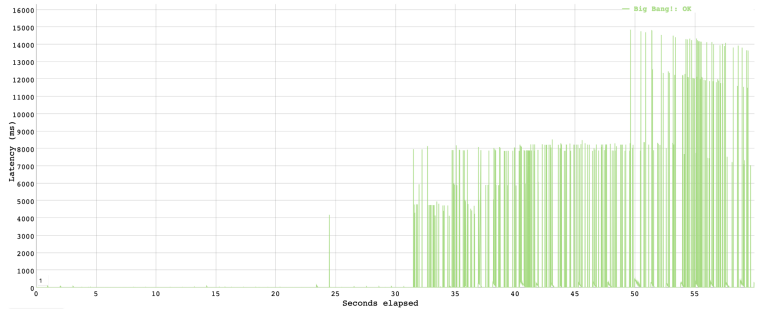


Unlike the lower request rates, at 500 requests/s, there is a significant difference in the average and 99th percentile latencies for all test cases, though this difference is much larger in the cases with fewer nodes. With 5 nodes, the difference is more than 10s while with 20 nodes, it is approximately 1-3s.

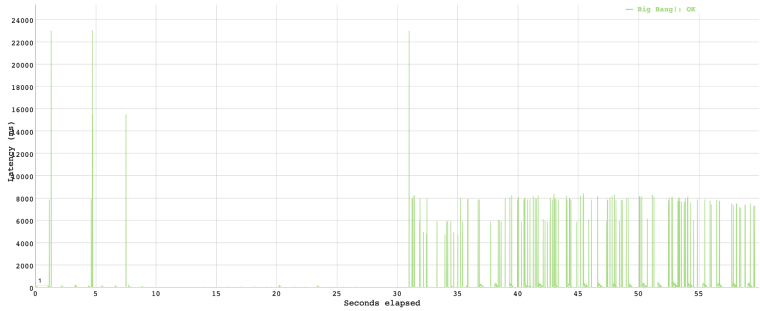
Distribution of request latencies in 1 test with 5 nodes and 500 requests/s



Distribution of request latencies in 1 test with 10 nodes and 500 requests/s



Distribution of request latencies in 1 test with 20 nodes and 500 requests/s



The latency distribution graphs at 500 requests/s provide greater insight to the earlier 2 graphs.

With 5 nodes, we see a few levels of increase in the latencies of requests, with most requests being fulfilled within a few ms before the 30s mark, and many requests requiring about 7s, 13s and 20s to complete thereafter.

With 10 nodes, the graph is less dense where there is an increase in the latency. Additionally, the levels at which latencies increased to are lower at about 8s and 13s.

With 20 nodes, significantly fewer requests suffer from the increased latency and latency generally only increased to about 8s as well.

These results demonstrate why the average latency of requests decreased significantly as the number of nodes increased, and provided evidence for the difference in the average and 99th percentile latencies.

While there were initially concerns that the single Nginx load balancer could act as a bottleneck during the load tests, these results indicate that any overhead from Nginx is likely negligible since increasing the number of nodes still managed to reduce the request latencies. This pattern indicates that requests were likely being backlogged due to the nodes inability to process the large influx of requests fast enough instead.

3.2 Consistency/Correctness

3.2.1 Considerations

3.2.1.1 Data Structure of ClientCart Objects

```
type BadgerObject struct {
    UserID   string
    Versions []ClientCart
    Conflict bool
}
```

BadgerObject schema

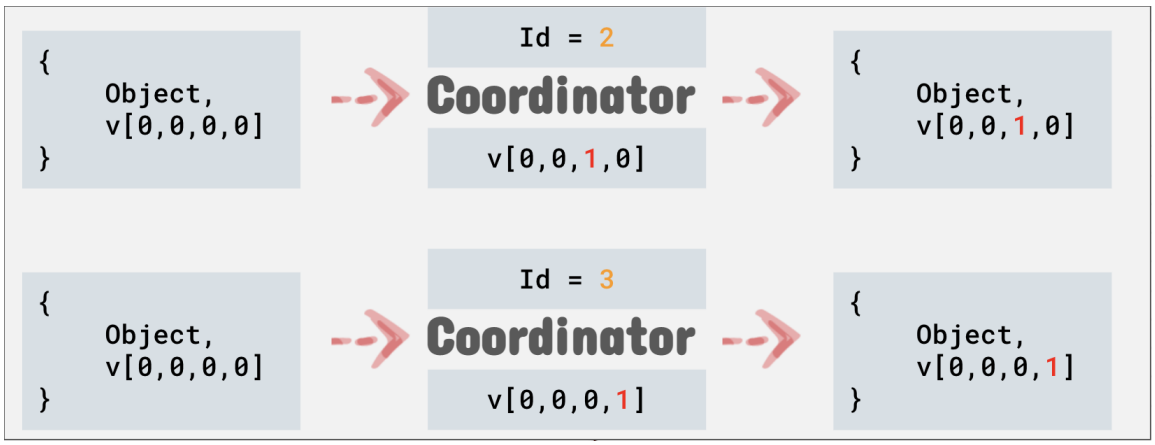
BadgerObject is the data structure for the “value” in our Key-Value store.

```
type ClientCart struct {
    UserID      string
    Item        map[int]ItemObject
    VectorClock map[int]int // {coordinatorId: verstion_number}
    ClientId    string
    Timestamp   int
}
```

ClientCart schema

ClientCart is the data structure of one Version of a user’s shopping cart

3.2.1.2 Vector clock processing during write requests by coordinator



Coordinator Vector Clock Increment

Upon receiving a write request (HTTP Post), the load balancer will select a node to act as the coordinator to handle the write request. The selected coordinator would assess the incoming ClientCart object and attempt to retrieve a vector clock.

Should there be an absence of a vector clock for the said object, the coordinator will create a vector clock for the object. Thereafter, based on its ID, the coordinator will increment the corresponding position in the vector clock before forwarding the object and the write request to the nodes responsible for the object's ID.

Otherwise, the coordinator will simply increment its corresponding position in the object's existing vector clock before carrying out the object forwarding to the responsible nodes.

Using this vector clock, the responsible nodes can then differentiate the versions of the incoming ClientCart objects as later versions would have higher vector clocks. This will be covered in greater detail in the next section (Section 3.2.1.3).

3.2.1.3 Vector clock processing during write requests by responsible nodes

In general, a node can contain multiple versions of a ClientCart object with varying vector clocks that have ambiguous ordering. We have thus designed the “value” in our key-value store to keep an array of objects with multiple versions.

When a node receives an incoming write request for a WriteObject, it will compare the vector clock of this incoming request with the existing versions in its persistent storage and decide on the actions to perform for the incoming request. To do this, the node will first create a temporary array before iterating through the different versions that may exist in its persistent storage. At each iteration, it will compare the 2 vector clocks(incoming object and existing object). Should the incoming vector clock be strictly larger than the current iteration, the node will continue to the next iteration. Otherwise, the node will append the current iteration into the temporary array before moving on to the next iteration. At the end of the iteration, the node will then append the incoming write request to the temporary array before committing this array to its persistent storage. Of which, there are 3 possible outcomes depending on the vector clocks of the existing versions in stored in the node’s persistent storage:

Case 1: Incoming vector clock is strictly larger than all existing stored versions

If the incoming vector clock has a strictly larger value than **all** existing versions, all stored versions will be deleted and only the incoming version will be kept.

One possible way this can happen is after a version merge was done on the (client) front-end service, leading to a merging of all versions (including a “max” operation on all vector clocks).

Case 2: Incoming vector clock is only strictly larger than some of the existing stored versions

If the incoming vector clock has a strictly larger value than **some** existing versions, only stored versions with strictly smaller vector clocks than the vector clock of the incoming object would be deleted, resulting in several versions waiting to be merged by the client. This would mean that a client is carrying out operations on one of the existing versions before any conflict resolution has occurred.

In this case, a boolean attribute called “Conflict” would also be set to true and passed to the client to raise a warning.

Case 3: Incoming vector clock is not strictly larger than any existing stored versions

If the incoming vector clock is not strictly larger than **any** existing versions, no existing version would be deleted. The incoming versions would also be appended to the existing versions. This would mean that the client is operating on a stale version of the client cart before any conflict resolution has occurred.

In this case, a boolean attribute called “Conflict” would also be set to true and passed to the client to raise a warning.

3.2.1.4 Conflict Resolution: Merge during read resolves conflicts by client

When a conflict is flagged to the client after a successful write request, the client would raise a warning to encourage a “read” to happen before further “writes”.

We have designed our system to allow the merging semantics of multiple object versions to be defined by the client.

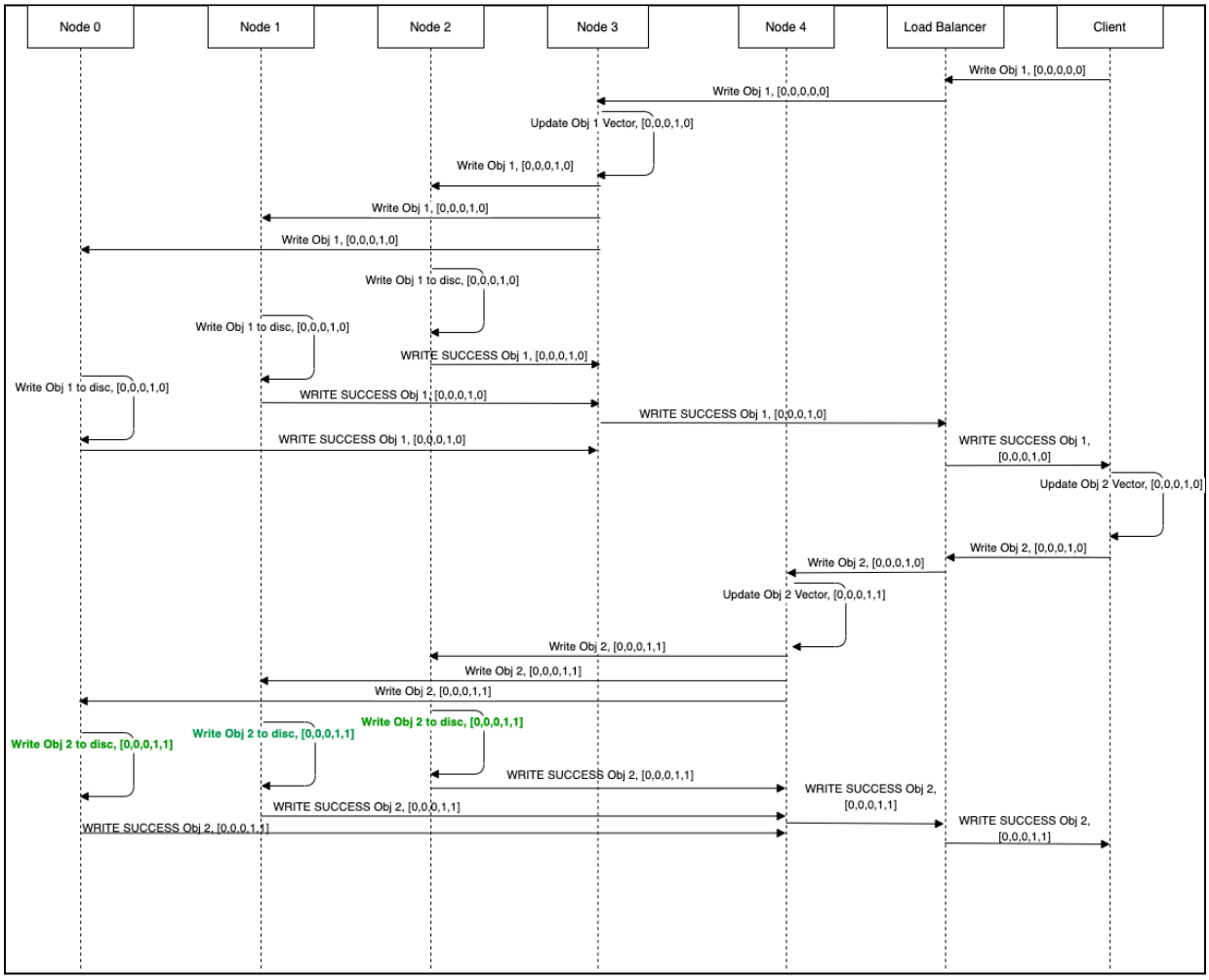
Our defined merging semantics

Our merging semantics involve:

1. Separating the versions and grouping them by client ID
2. Taking the latest version of each client ID
3. Taking union (or max) values from each group.

3.2.1.5 Possible scenarios considered for this system:

Scenario 1: Seq diagram for 2 sequential requests (Normal use case)



Sequence diagram for 2 sequential write requests

Scenario 2: Seq diagram for 2 concurrent requests (2 writes before response)

Since the client (1) sends the vector clock of the object when writing, and (2) only updates the object's vector clock when the write operation completes, there lies an opportunity to for a client to make a request using the same vector clock before the first write completes to allow the client to update the object's vector clock.

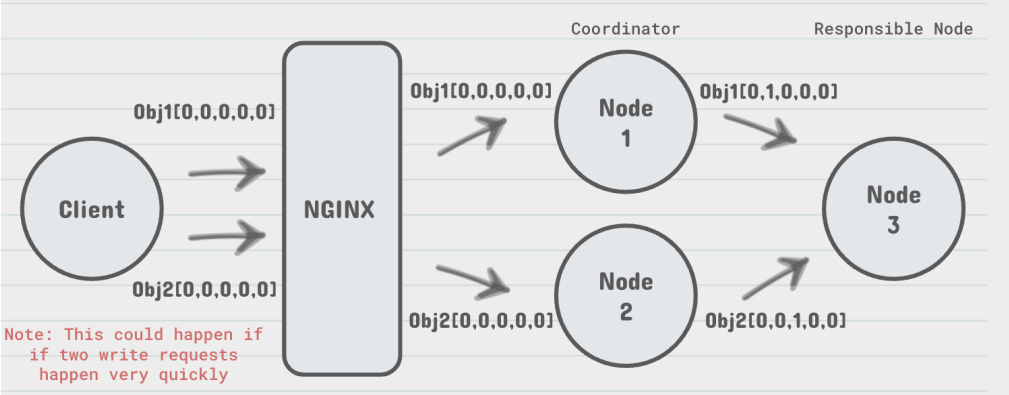
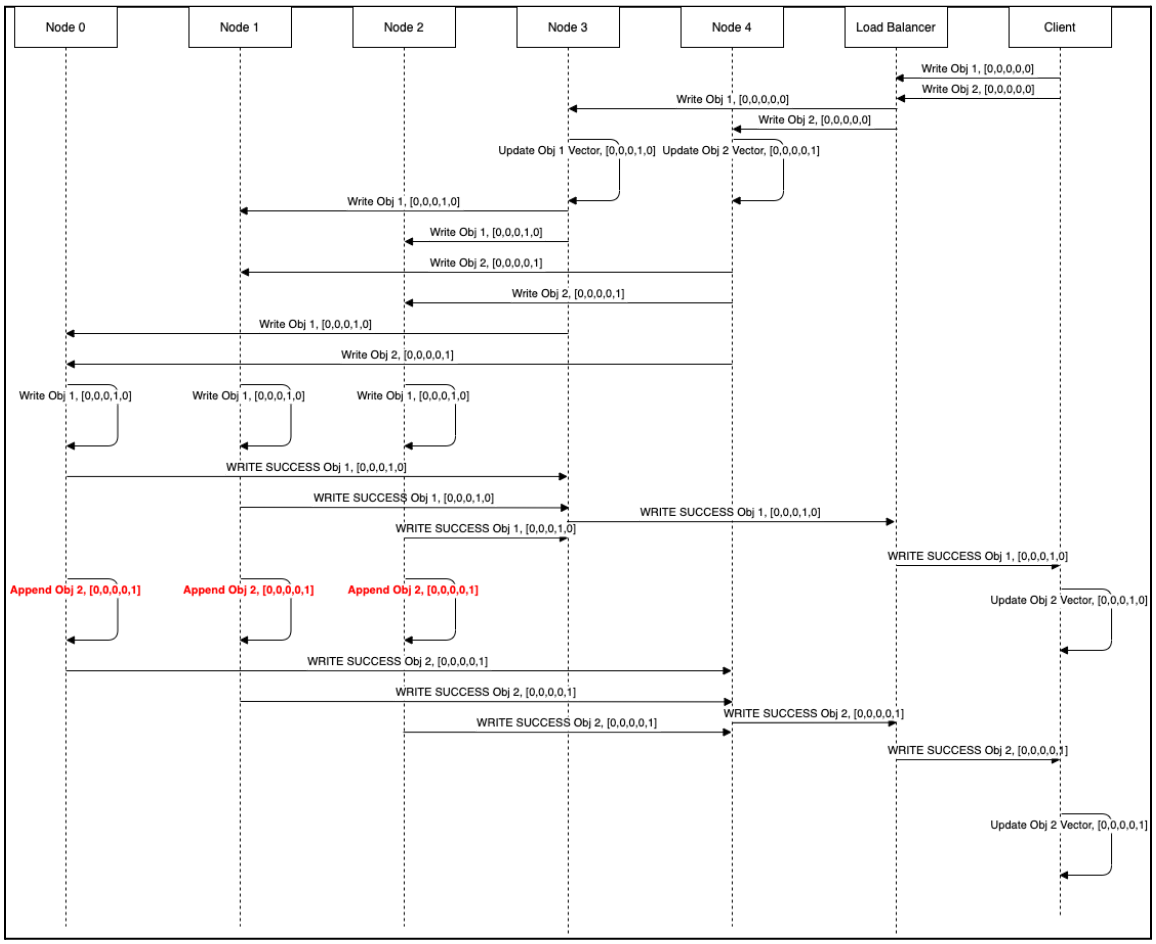
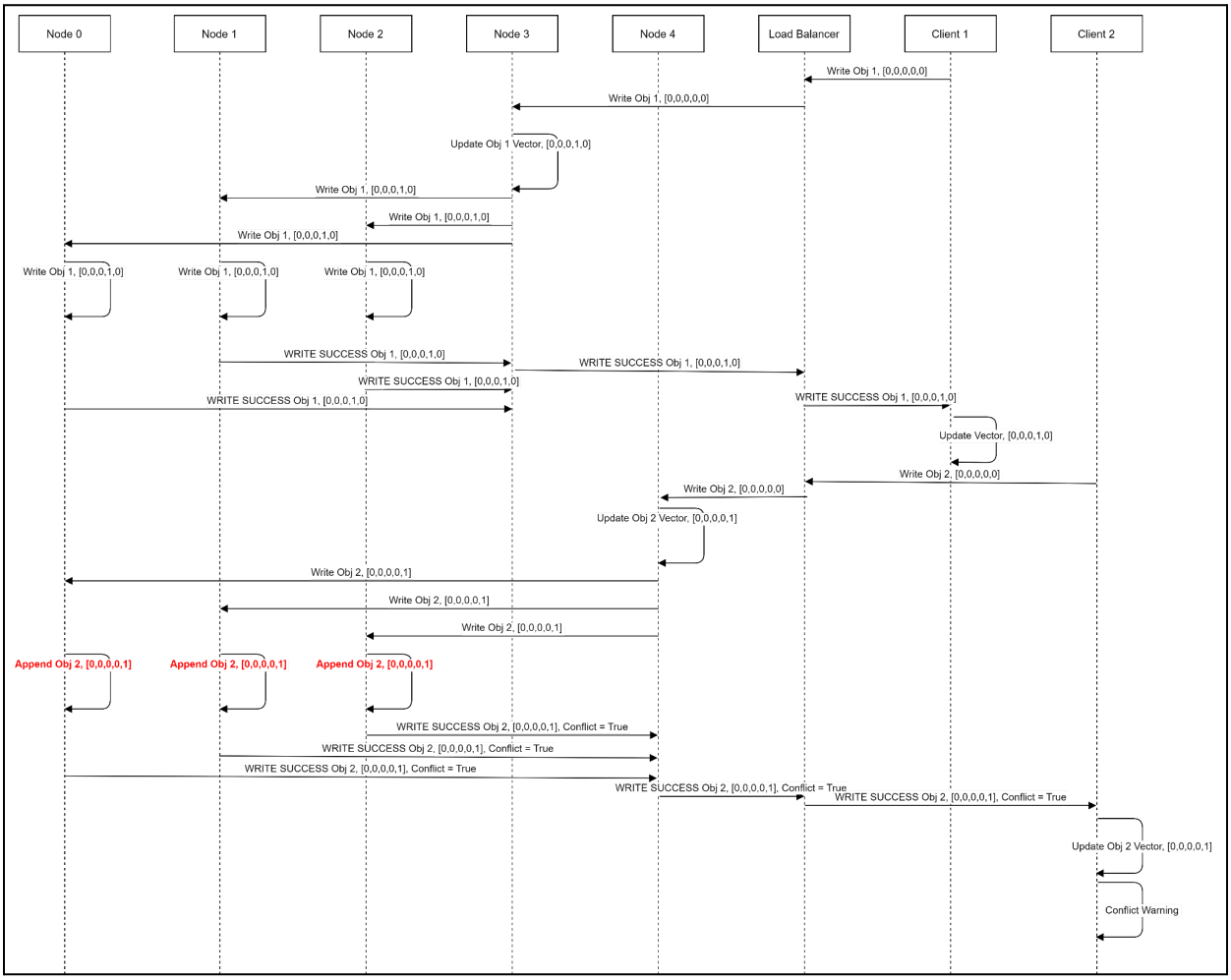


Illustration of how concurrent vector clocks can get created



Sequence diagram for 2 concurrent requests from the same client

Scenario 3: Seq diagram for 2 concurrent requests (2 clients and stale writes)



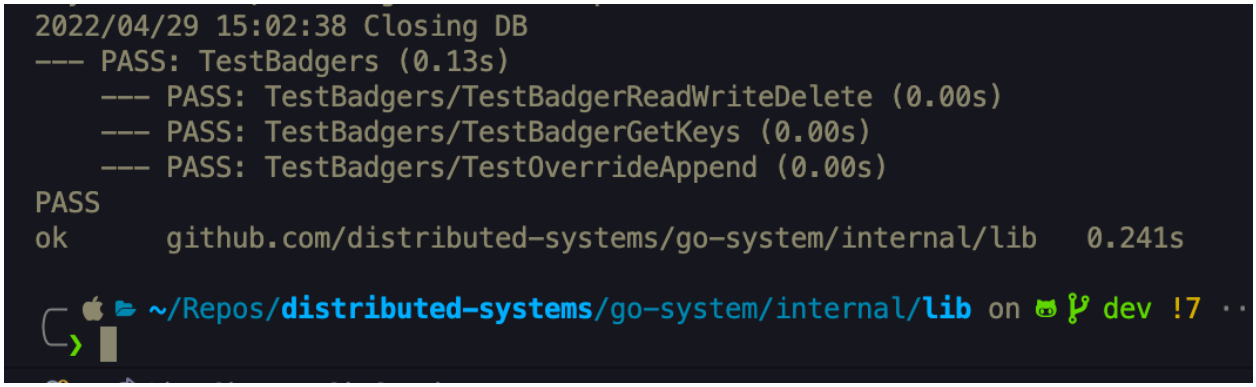
Sequence diagram for 2 concurrent requests from different clients

3.2.2 Experiments

3.2.2.1 Unit Tests for Appending New Versions

Unit tests were conducted using golang's internal testing tool to test the appending of new versions upon encountering a conflicting ambiguous vector clock at the node level. By first initialising a testnode and seeding it with an arbitrary entry and a predetermined vector clock, we can then introduce a write operation to that same key store with a comparatively ambiguous vector clock to simulate the environment that would call for the appending of this new version instead of overwriting the previous version.

Additionally, we also conducted unit tests for the verification of the other possibility whereby an incoming value's vector clock is strictly larger than the existing value stored in the database which would prompt the overwriting of the previous version. This was conducted using the unit test shown below:



Unit test results

3.2.2.2 Manual Testing of Concurrent Writes

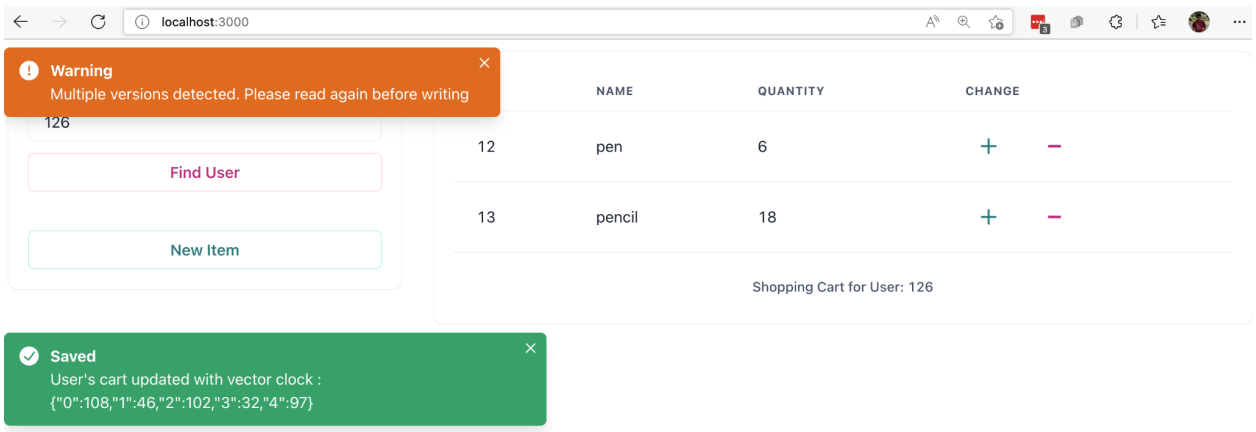
Manual testing was done to simulate the concurrent write operations using the front-end application that we implemented. We first started the entire distributed system through the conventional joining procedure we implemented (see Section 2.5.2). Thereafter, the NGINX load balancer was started before we conducted our testing.

Overall, we simulated 2 of the possible scenarios that we covered previously (see Section 3.1.2.5).

Firstly, to simulate the normal use case that would result in an overwrite and a single value stored in the database throughout the entire duration of the test, we first initiated a write operation from the front-end application by incrementing or decrementing any item quantity through the "+" or "-" buttons. This could also be done by adding a new item to the cart. Thereafter, we would wait for the write success toast message before we sent a second write operation from the same front-end. We verified that there was only one version for the same client through the console output of both operations in the "versions" entry of the return json

from both operations and the update of the vector clock wherein the 2nd write operation would be strictly larger than the 1st write operation.

Secondly, we tested the submission of stale values by 2 client copies through the use of different instances of the same front-end application. Opening both tabs, we first opened the cart of the same client on both instances of the application. Thereafter, we issued a write request from one of the instances and await the success notification. The other instance of the front end application would now have stale values of the same client cart with a stale vector clock. With this stale cart, we then issued a write request. This resulted in an orange warning of stale values and conflicting versions existing in the database (shown below), which was the expected behaviour of our implementation of the DynamoDB clone whereby merge conflicts are handled by the middleware or front-end applications using the key-value store.



Warning displayed on front end application when conflicts are detected

These tests were also done using HTTP curl requests from the command line.

3.3 Fault Tolerance

3.3.1 Considerations

The key points to we had to take into consideration when designing our protocols for fault tolerance were:

1. The system only handles temporary failures.
2. When a temporarily failed node revives, it will rejoin the system, be assigned the same position and continue operations from there.
3. When a node fails during writes, a hinted replica will be sent to the next live successor.
4. When the coordinator encounters node failures during reads, the coordinator will query the successors for any hinted replicas.

3.3.1.1 Hinted Handoff

When the coordinator sends write requests to the responsible storage nodes, it will maintain a timer for detecting unannounced node failures. For each request sent, if the coordinator does not receive a response before the timeout occurs, the coordinator will recognise it as a node failure and proceed to conduct a hinted handoff. The coordinator will find the next live successor of the failed node that is not already storing the replica and send the write request to that node, along with a hint of the intended recipient for that key.

The node storing the hinted replica will periodically send write requests to the intended recipient to hand off the data. Once the failed node revives, it will receive the write request, update its data store to reflect the latest changes and its successor will delete the hinted replica and stop the hinted handoff operation.

During this process, if read requests are sent and the coordinator is unable to contact the responsible nodes, the coordinator will identify the successors of the nodes and query them for any hinted replicas. If hinted replicas are found, they will be returned to the client in the read response, thus ensuring further fault tolerance.

It is important to note that the number of nodes alive must be at least the minimum successful write count (W=2) defined for hinted handoff to work, thus the requirement of minimum 2 working nodes. Additionally, the client will receive a successful response as long as 2 replicas of the data are stored, regardless whether they are hinted replicas or normal replicas.

3.3.1.2 Maximum Node Failures

Given N nodes and F failures,

$$N - F \geq \max(W, R)$$

where W and R are the minimum read and write success required respectively
Hence, our system has a fault tolerance of F where

$$F \leq N - \max(W, R)$$

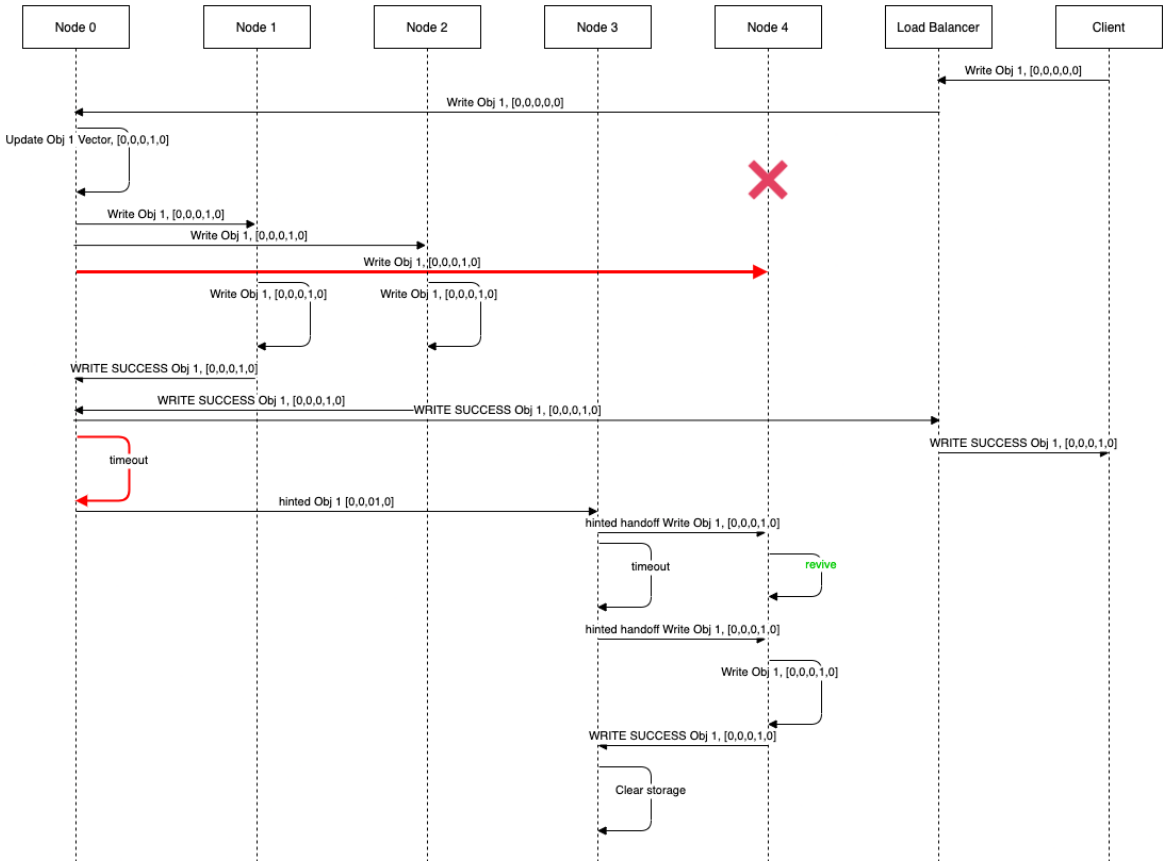
e.g. for N = 5, we can only allow a maximum of 3 node failures.

3.3.2 Fault Tolerance Cases with Experiments

The experiments were conducted by temporarily failing specific nodes during various write operations. A demonstration of the fault tolerance cases can be found in Appendix B.

Case 1: Write Operation with single node failure

A write request for Object 1 is sent to the coordinator (Node 0), with the responsible nodes of Node 1, 2 and 4 to handle the data store. However Node 4 is down, so the write operation to Node 4 is unresponsive. Since the coordinator receives 2 successful responses, it will first return a success response to the client. After the coordinator triggers a time out, the coordinator will get Node 4's successor, Node 3, to store the hinted replica for Node 4 and periodically attempt to hand off the replica to Node 4. When Node 4 revives, Object 1 will be successfully written into Node 4's data store and Node 3 can delete the replica.

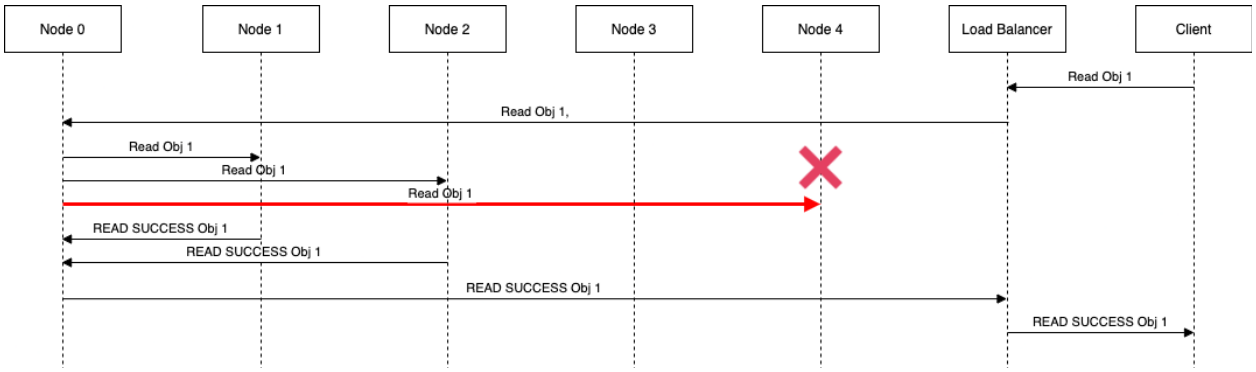


Sequence diagram for write operation with a single responsible node down

For screenshots of the actual output and steps to recreate fault refer to Section B.1 in the appendix.

Case 2: Read Operation with single node failure

A read request for Object 1 is sent to the coordinator (Node 0), with the responsible nodes of Node 1, 2 and 4 to provide the data. Because the replication factor of the system is 3 and the minimum read success needed is 2, even if Node 4 is down Nodes 1 and 2 are able to provide the data for Object 1. Thus, the read operation is successful.



Sequence diagram for read operation with a single responsible node down

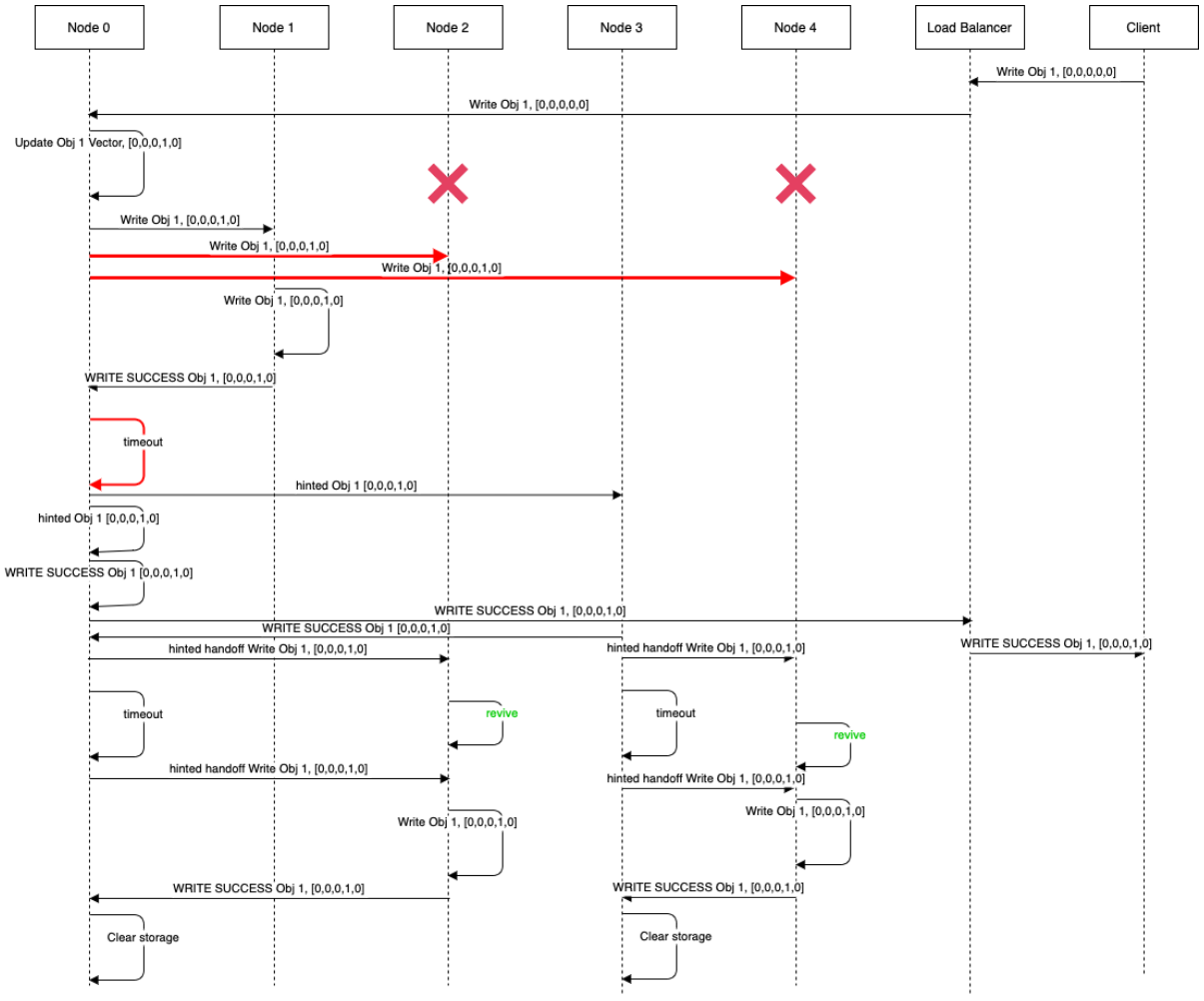
For screenshots of actual output and steps to recreate fault refer to Section B.2 in the appendix.

Case 3a: Write Operation with multiple node failures (responsible nodes failed)

A write request for Object 1 is sent to the coordinator (Node 0), with the responsible nodes of Node 1, 2 and 4 to handle the data store. However, in this case, since Node 2 and 4 are down, the successful write is only received by the coordinator from Node 1. As this is less than the minimum successful writes factor of 2, the write success message will be delayed until one of the hinted write handoffs is complete.

After the coordinator triggers a time out, the coordinator will get the next successors (Node 3 and Node 0) to handle the hinted handoff. The write will be considered successful once one of the nodes returns a successful response. When that happens, the coordinator will send the success message back to the client.

When Node 2 and 4 revive, Object 1 will be successfully written into their data store and Node 0 and 3 can stop the hinted handoff.



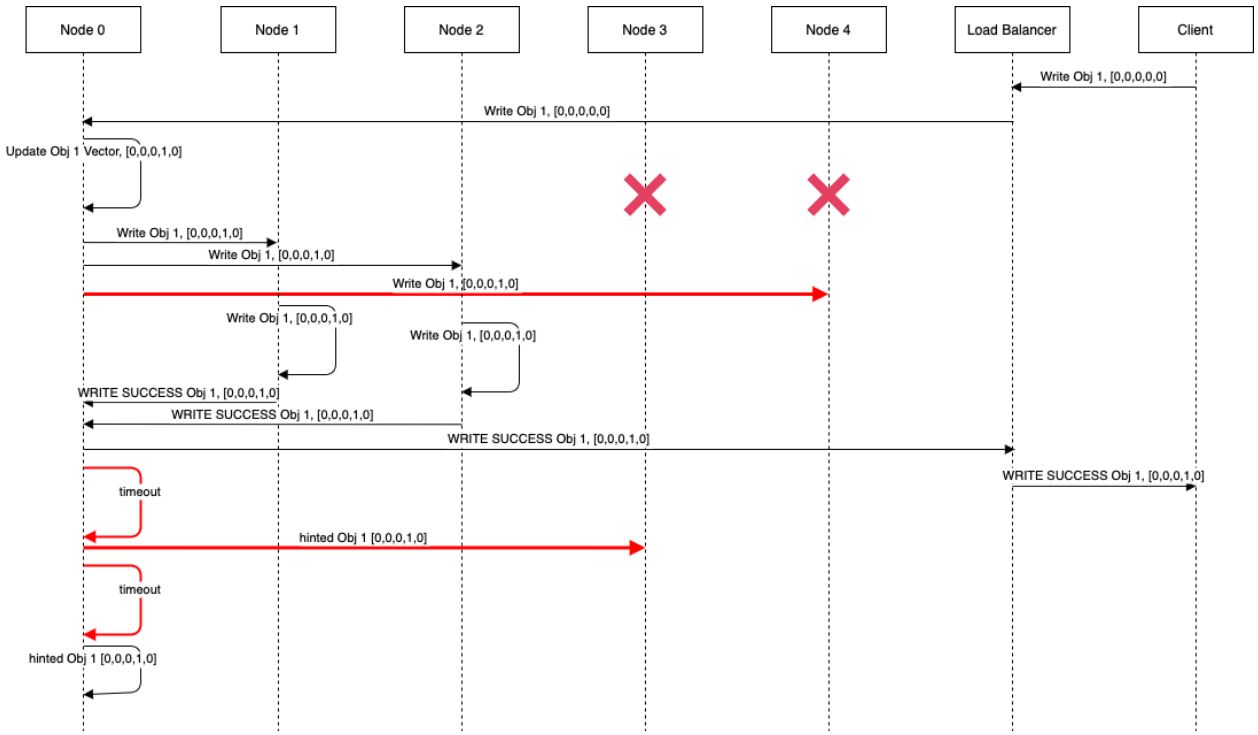
Sequence diagram for write operation with 2 responsible nodes down

For screenshots of actual output and steps to recreate fault refer to Section B.3 in the appendix.

Case 3b: Write Operation with multiple node failures (responsible nodes + successor node failed)

A write request for Object 1 is sent to the coordinator (Node 0), with the responsible nodes of Node 1, 2 and 4 to handle the data store. However, in this case since Node 4 and its successor, Node 3, are down, successful write responses are only received by the coordinator from Node 1 and 2. Since the coordinator has received 2 successful responses, it will first return a success response to the client.

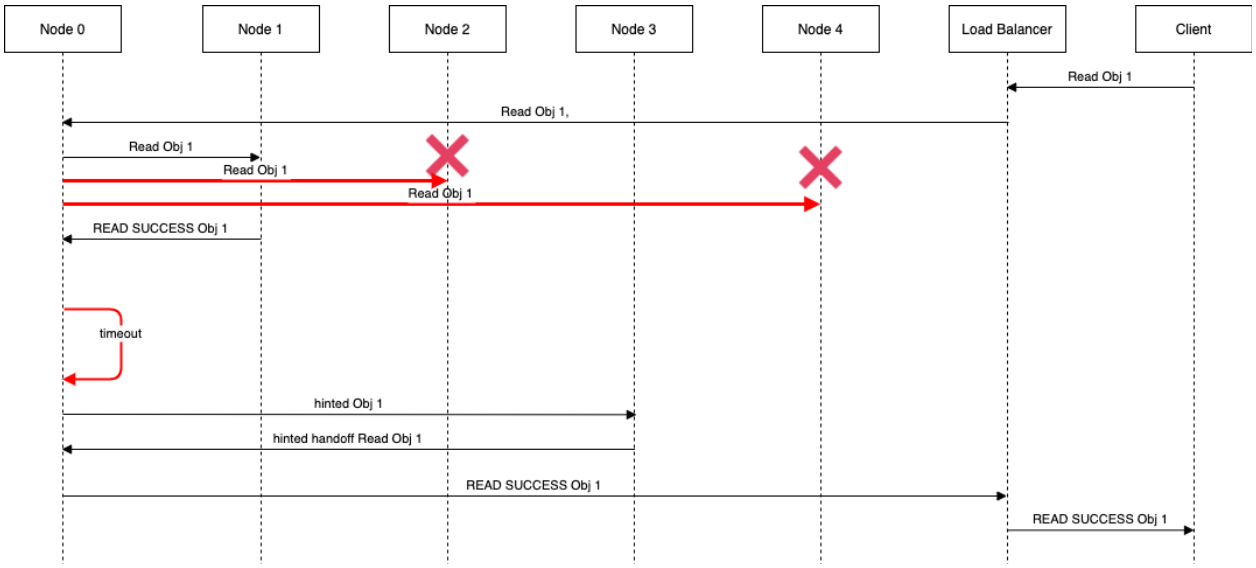
After the coordinator triggers a time out, the coordinator will get the next successor, Node 3, to handle the hinted handoff. However, unlike Case 3a, Node 3 is down as well, so the coordinator will have to wait for another timeout before sending the hinted handoff to the successor's successor, Node 0. Hinted handoff will proceed as described in Case 1.



Sequence diagram for write operation with a single responsible node and the immediate successor node down

Case 4: Read Operation with multiple node failures (with nodes containing hinted replicas)

Similar to Case 2, when Node 2 and 4 are down, Node 0 and 3 handle the hinted handoff. If a client tries to read Object 1 before Node 2 and 4 revive, out of the 3 responsible nodes, the coordinator will only be able to get a successful read response from Node 1. Hence, the coordinator will identify the 2 immediate successors of Node 2 and 4, which are Node 0 and 3, and send read requests to check if those nodes are holding hinted replicas. If they are, successful read responses will be sent to the coordinator. Since the minimum successful read count is set to 2, once one of the nodes returns a successful read response, the coordinator will return a read success message to the client.



Sequence diagram for read operation with 2 responsible nodes down

For screenshots of actual output and steps to recreate fault refer to Section B.3 in the appendix.

4 Future Work

To create a more robust system, the following features were considered for further development:

1. Implementation of leaving function of nodes
2. Fault tolerance during joining
3. Merkle trees for efficient synchronisation between diverged nodes
4. Gossip based membership protocol and failure detection

4.1 Leaving Function

During traffic spikes, nodes will have to be added to the system to support the increased load. However, once the spikes drop back to a normal level for daily operation, these additional nodes should be able to be safely removed without impacting the correctness of the system. Hence, a suitable leaving protocol can be designed and implemented as a future extension.

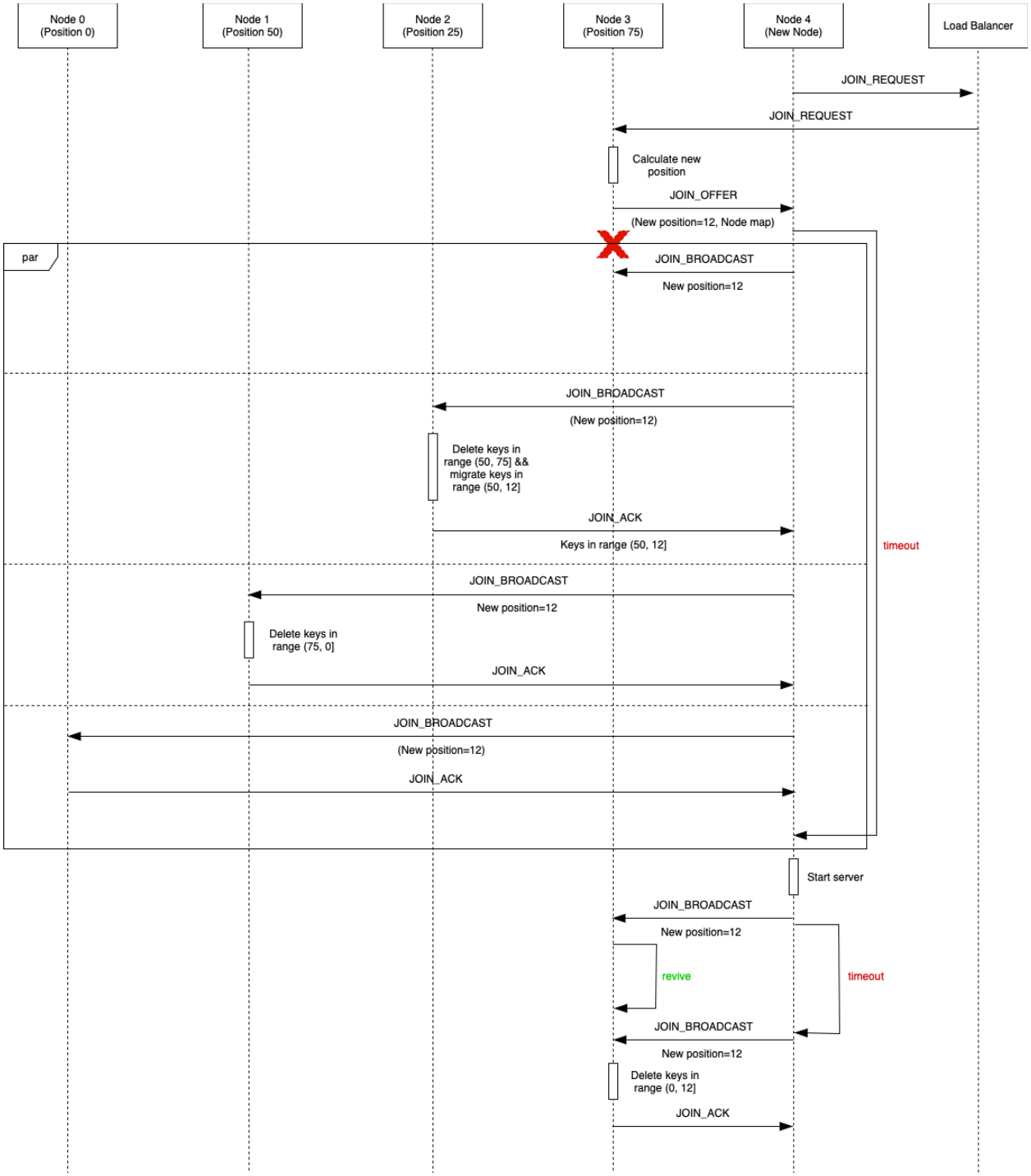
4.2 Fault Tolerance during Joining

The current joining protocol assumes that all nodes will be alive when a new node wishes to join the system since the new node must wait for an acknowledgement from all nodes before starting its API server. However, this is unlikely to be the case in reality as new nodes will have to be added when traffic spikes, during which some nodes may have already failed.

While we were unable to implement a fault tolerant joining protocol within the time and scope of this project, considerations have been made in designing such a protocol, as shown in the following sequence diagram.

The key change to the original joining protocol is that the new node will no longer wait for all replies to the JOIN_BROADCAST messages before starting the server. Instead, it will wait until a timeout occurs. All nodes that have yet to reply will be marked as failed. The new node will continue to retry the JOIN_BROADCAST message to these failed nodes after starting its server, similar to the idea presented in the hinted handoff. Only when the failed nodes revive and send a reply will this periodic retry stop.

Such a protocol means that if the failed node is the node responsible for migrating the keys to the new node, the new node will be starting without the data it should be storing. Writes to the keys that this new node should store will continue to work as the vector clocks specified by the clients will get updated by the coordinators, thus allowing this new node to store newer versions of the objects. However, reads will not work until the keys are migrated. Additionally, if writes have been made to existing keys when the data is migrated, vector clocks between the migrated keys and the newly written keys will have to be compared to ensure that all concurrent versions are stored correctly.



Sequence diagram for joining protocol with fault tolerance

4.3 Merkle Trees

In our system, transient failures are handled using hinted handoff. However, it may be possible that nodes storing the hinted replicas fail before being able to pass the replicas back to the original nodes, leading to the loss of data. To ensure that eventual consistency is maintained between all nodes and to handle such permanent faults, Merkle trees can be used to effectively compare and synchronise divergent replicas in the background.

4.4 Gossip-based Membership Protocol and Fault Detection

Given that in an actual system there are very frequent membership changes, using a broadcasting protocol for the entry and exit of nodes might be taxing. Thus Dynamo has implemented a gossip-based membership protocol that propagates membership information periodically where each node would contact another node at random to reconcile any membership differences. This mechanism thus facilitates the updating of membership history between nodes.

5 Access to Code

The code can be accessed from Github at this link.

6 Conclusion

Inspired by DynamoDB, we managed to prototype a highly available and scalable data store with the front facing cart application that is used for storing key-value records. Our database is also incrementally scalable, allowing us to add more nodes and more users to meet the required system demands. We have gained glimpses into how DynamoDB is utilised by Amazon to provide a system that is reliable and highly available for internal services that are a part of their e-commerce platform.

While the project was tough in that it required careful planning and alignment of the various protocols utilised by the various components of the system, it was also fulfilling when each feature was implemented successfully. The project provided great insight into how distributed systems can be implemented and the numerous considerations that are needed to be taken in the design of such systems.

We are grateful to Professor Sudipta for his guidance and advice through the various checkoffs and consultations. Your contributions have helped guide us and have shed light on how protocols ought to be considered in unison in distributed systems.

7 References

1. UN News. (2021, May 3). Global e-commerce jumps to $26.7 trillion, fuelled by COVID-19. UN News. https://news.un.org/en/story/2021/05/1091182
2. Dgraph. (n.d.). *BadgerDB Documentation - Dgraph.io*. Retrieved from https://dgraph.io/docs/badger/
3. Giuseppe DeCandia, D. H. (n.d.). *Dynamo: Amazon's Highly Available Key-value Store.* Amazon.com.
4. Nginx, Inc. (n.d.). *NGINX Documentation*. Retrieved from http://nginx.org/en/docs/
5. Senart, T. (n.d.). *Vegeta Documentation - Github.com*. Retrieved from https://github.com/tsenart/vegeta

Appendix

Appendix A: Load Test Results

# of Nodes	Req type	Req rate (#/s)	99th Percentile Latency (ms)				Average latency (ms)			
			Test 1	Test 2	Test 3	Avg	Test 1	Test 2	Test 3	Avg
5	Read	10	5.35	5.29	4.87	**5.17**	3.48	3.05	3.25	**3.26**
		100	16.07	15.17	10.48	**13.91**	3.71	3.57	3.33	**3.54**
		500	12819.81	12436.53	14633.24	**13296.52**	342.00	343.79	412.24	**366.01**
	Write	10	5.37	4.84	6.11	**5.44**	3.84	3.86	4.12	**3.94**
		100	14.16	14.21	15.61	**14.66**	3.87	4.00	4.08	**3.98**
		500	14902.52	14633.74	14040.51	**14525.59**	404.57	424.07	390.75	**406.47**
	Read Write	10	5.99	9.96	7.10	**7.68**	4.04	4.93	4.43	**4.47**
		100	16.26	15.56	16.43	**16.08**	4.91	5.01	4.95	**4.96**
		500	14694.29	13645.59	13950.75	**14096.88**	415.37	390.72	393.89	**399.99**
10	Read	10	5.06	5.15	5.25	**5.16**	3.63	3.71	3.56	**3.64**
		100	8.07	11.59	11.76	**10.47**	3.71	3.76	3.78	**3.75**
		500	6652.10	7816.61	7646.46	**7371.73**	132.50	150.42	160.24	**147.72**
	Write	10	6.27	7.43	5.46	**6.39**	4.55	4.47	4.22	**4.41**
		100	16.76	15.85	14.87	**15.83**	4.54	4.58	4.66	**4.60**
		500	7407.51	7887.23	8108.34	**7801.03**	152.16	166.72	181.01	**166.63**
	Read Write	10	7.75	8.48	7.47	**7.90**	4.99	5.48	5.26	**5.25**
		100	20.23	19.45	19.74	**19.81**	6.08	5.94	6.07	**6.03**
		500	7947.97	8125.07	7864.28	**7979.11**	171.12	191.81	164.63	**175.85**
20	Read	10	7.26	6.19	7.30	**6.92**	4.37	3.89	3.56	**3.94**
		100	11.23	15.08	14.97	**13.76**	4.15	4.27	4.22	**4.21**
		500	2273.28	765.48	538.71	**1192.49**	86.24	93.10	72.44	**83.93**

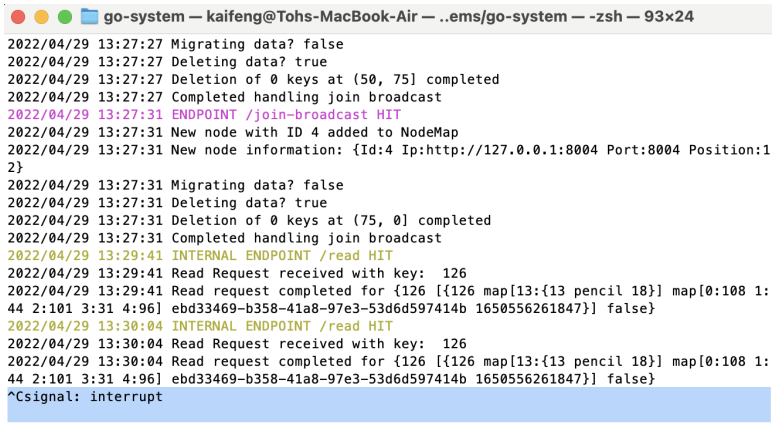
	Write	10	8.27	6.50	6.62	**7.13**	6.06	4.65	4.60	**5.10**
		100	16.91	15.64	20.37	**17.64**	4.61	4.62	4.71	**4.64**
		500	1644.57	912.14	1462.58	**1339.77**	98.19	74.72	99.17	**90.70**
	Read Write	10	12.44	12.70	12.65	**12.59**	6.66	8.61	6.21	**7.16**
		100	18.86	18.34	18.49	**18.56**	6.08	5.92	5.94	**5.98**
		500	1925.49	3800.85	1143.30	**2289.88**	91.99	109.83	81.65	**94.49**

Results of load test from Section 3.1.2

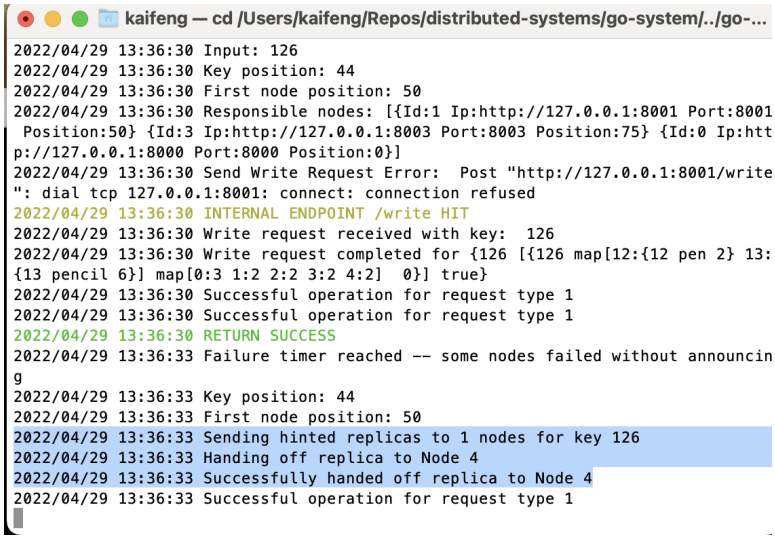
Appendix B: Experiments for Fault Tolerance Cases

B.1 Case 1: Write Operation with single node failure

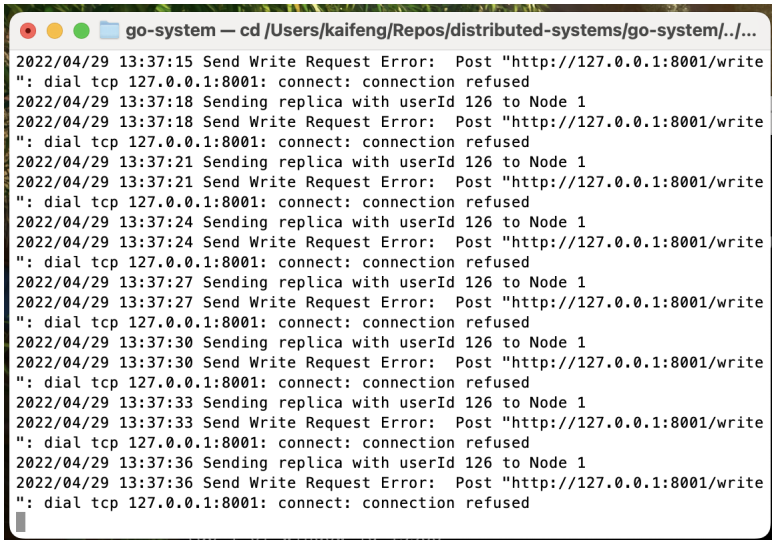
Step1 : Killing Node 1 using interrupt before trying to send a write request.



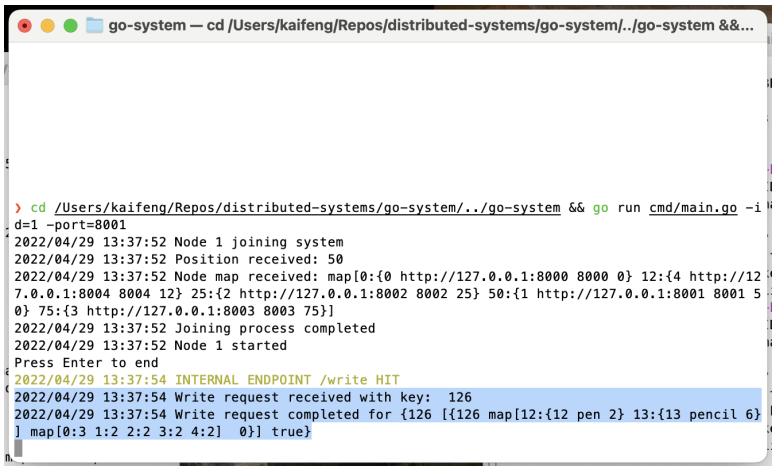
Step 2: Coordinator sending Hinted Replica to nodes



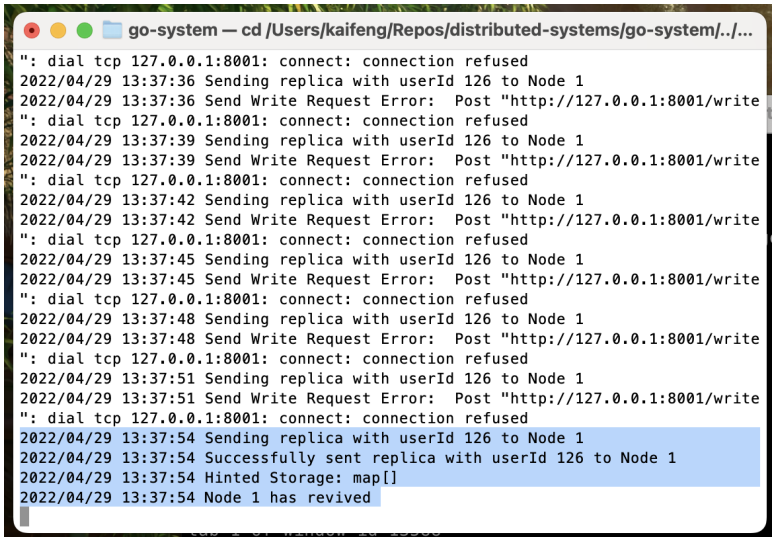
Step 3: Node with hinted replica repeated tries to send replica over to failed node.



Step 4: When failed node (node 1) revives, a write request from node with the hinted replica is received.

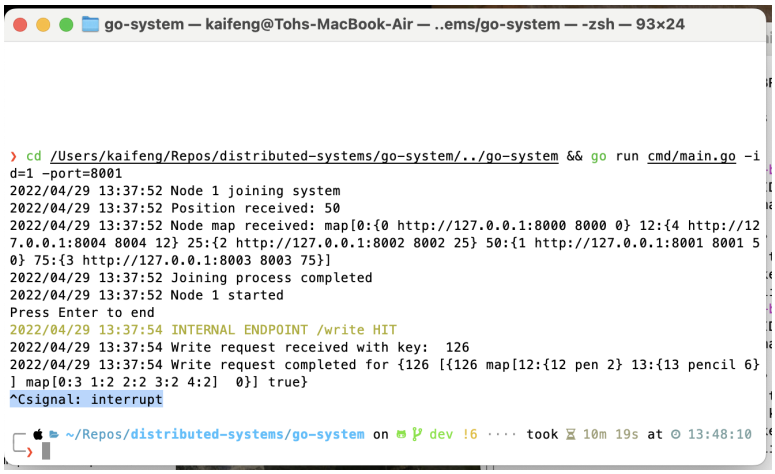


Step 5: Node with hinted replica stops repeatedly sending hinted write request, and deletes hinted storage.

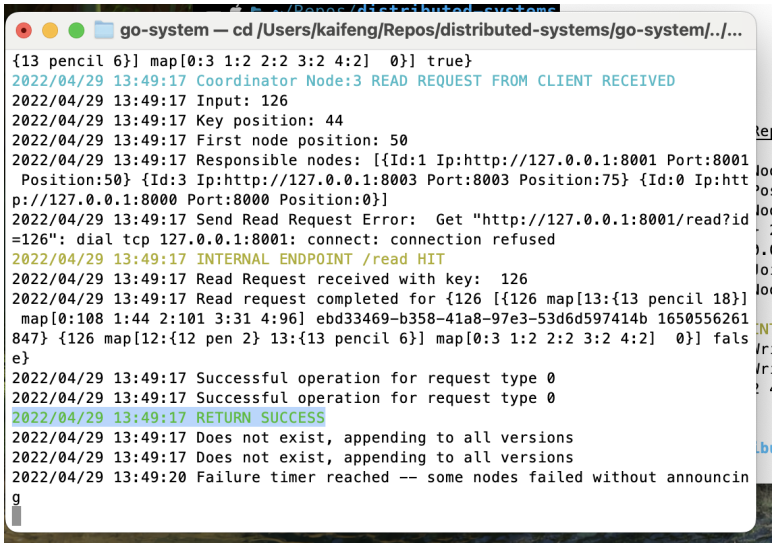


B.2 Case 2: Read Operation with single node failure

Step 1: Node 1 storing key 126 fails

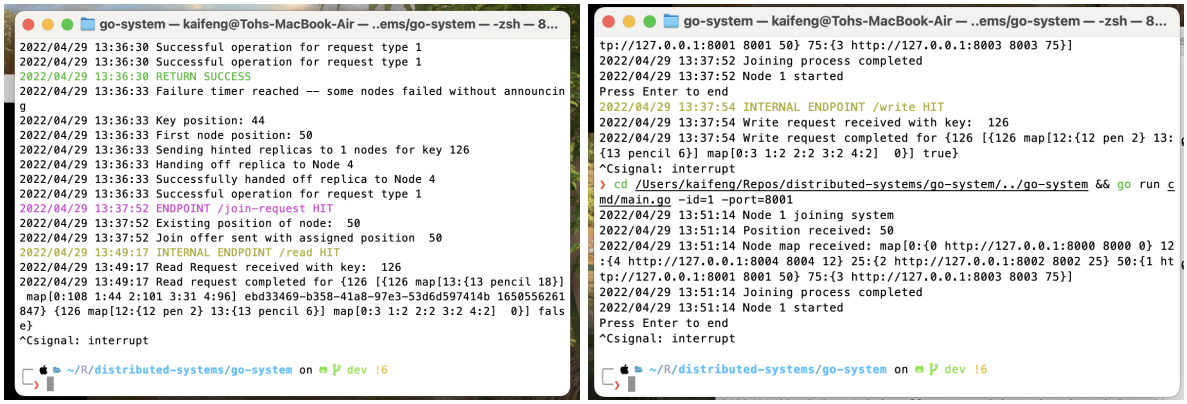


Step 2: Coordinator still returns a successful read to the client since 2 responses were received which is greater than the minimum successful read count we have configured (R=2).

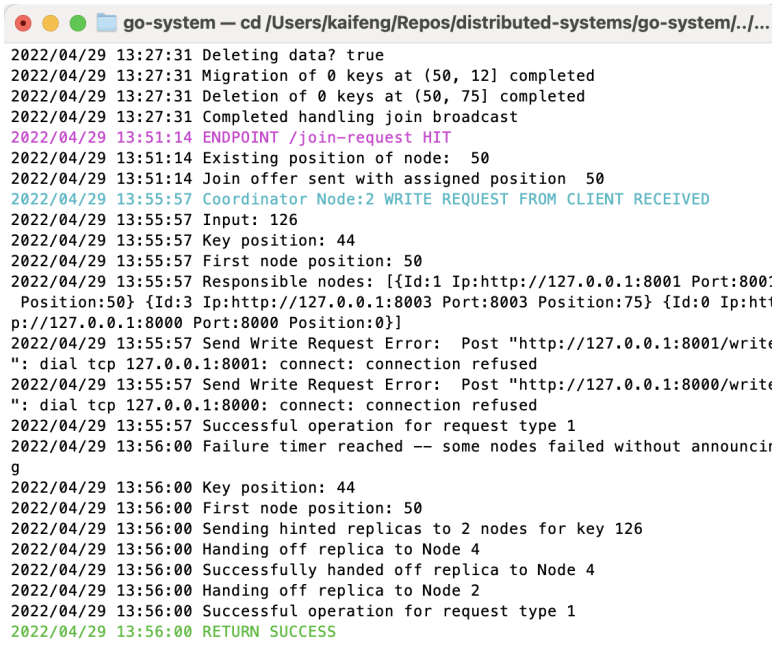


B.3 Case 3a & 4: Write Operation with multiple node failures + Read Operation with multiple node failures (with nodes containing hinted replicas)

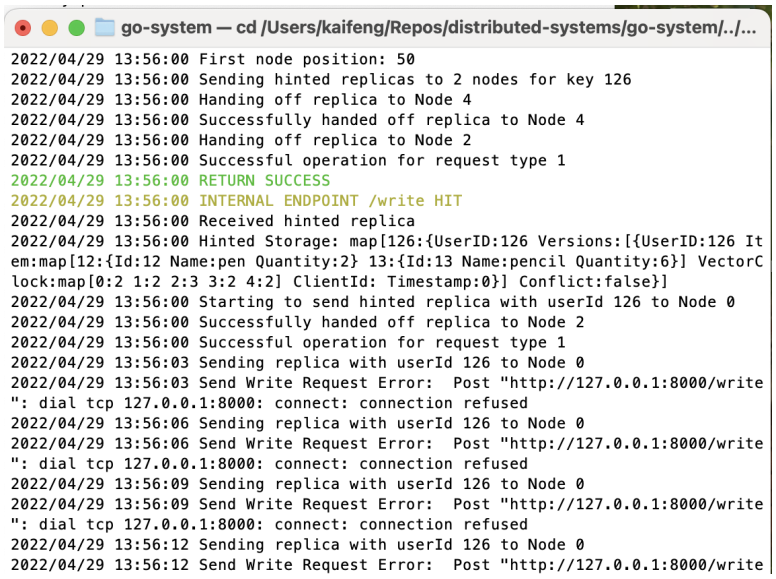
Step 1: Failing Nodes 0 and 1 since they are responsible for the storage of key 126.



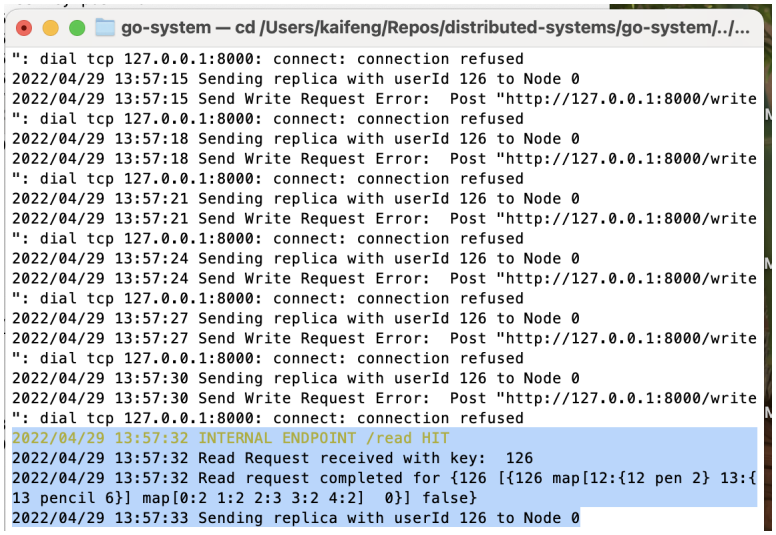
Step 2: Performing write request is still successful as long as an available successor node can store the hinted replica temporarily.



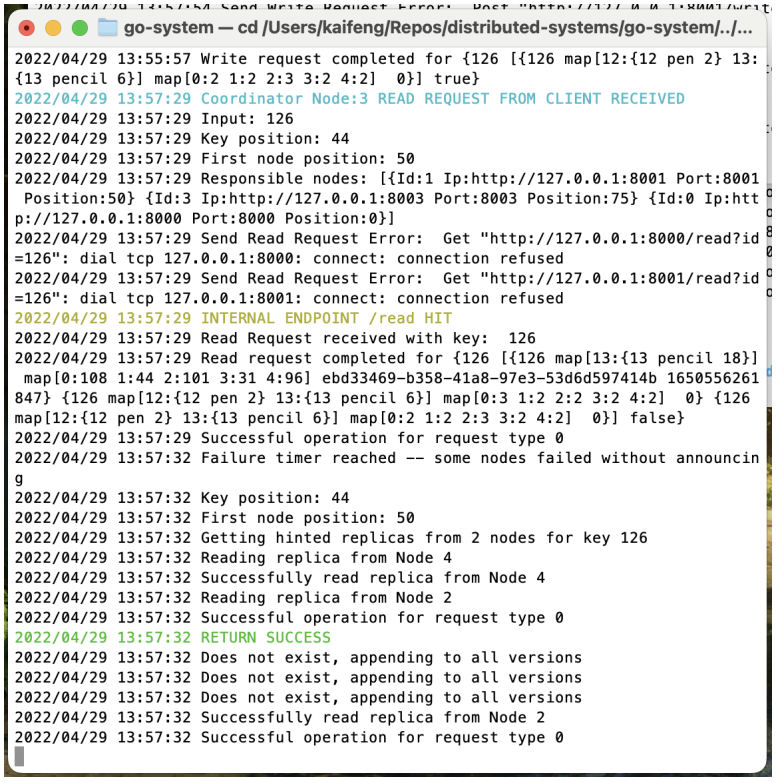
Step 3: Nodes with hinted replicas thus try to repeatedly send hinted write requests to failed nodes.



Step 3: If a read request is made while the failed nodes are still down, the coordinator would check whether there are any nodes containing hinted replicas. Nodes containing hinted replicas would thus respond to the coordinator if a hinted replica exists.

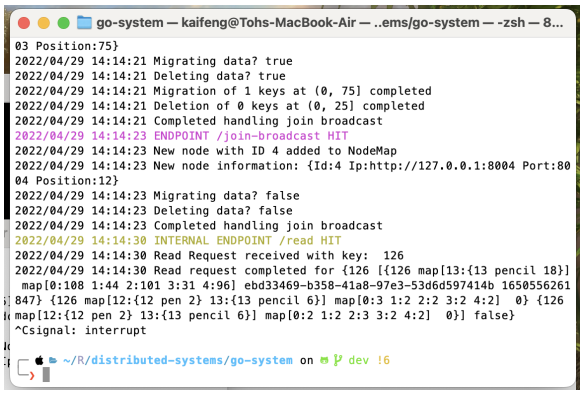


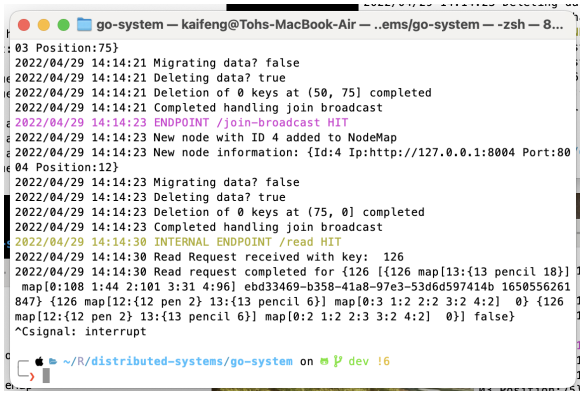
Step 4: Performing read operations while the failed nodes (nodes 0 and 1) are still down would still result in a successful read request where the coordinator can successfully return the data stored in the hinted replicas.

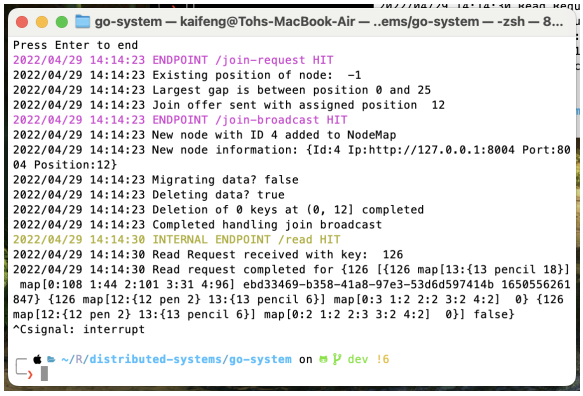


B.4 Case 3b: All originally responsible nodes fail

Step 1: Failing All 3 responsible nodes (Nodes 0,1,3)







Step 2: Coordinator handling the write request would have to wait for 2 nodes to store a hinted replica before returning a success message.