Final Project Check Off

Group 1:

Amazon's DynamoDB

Toh Kai Feng	1004581
Amrish Dev Sandhu	1004241
Tan Xin Yi	1004499
Goh Shao Cong Shawn	1004116
Lim Jun Wei	1004379

Presentation Flow

1. Intro
2. System Overview
3. Demo Time: Frontend
4. Features of Frontend
5. Demo Time: Joining
6. Demo Time: Read Write
7. Demo Time: Front End Vector Clock
8. Hinted Handoff
9. Demo Time: Scalability Tests
10. Scalability
11. Consistency - Sequential
12. Consistency - Concurrent
13. Fault Tolerance

Overview

1. Introduction
2. System Overview
3. Features
4. Scalability
5. Consistency
6. Fault Tolerance
7. Conclusion

Introduction

Problem and our approach

Introduction

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

Need for a system that is:

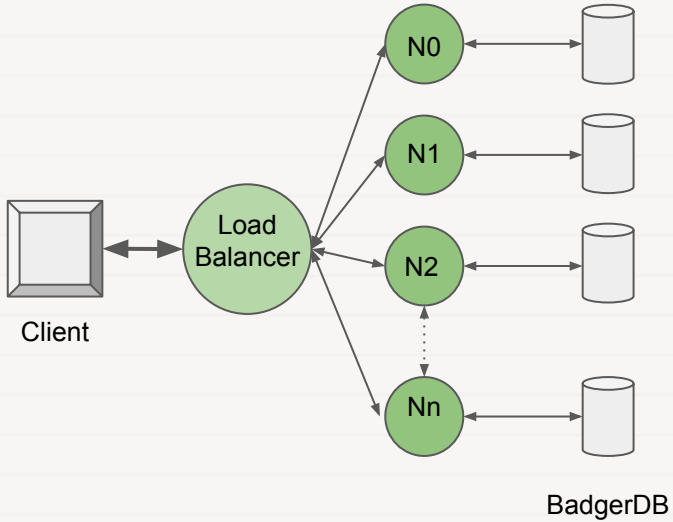
1. Scalable
2. Always Available
3. Reliable
4. Accurately reflects clients carts

02

System Overview

System Architecture,
Assumptions of the system and
Ring Structure.

2.1 Project Set-Up



System Architecture

- NGINX Load Balancer
- Use of BadgerDB for persistent data store
- 1 Ring containing 100 Positions for nodes
- Coordinator can be any node, will be contacted by the load balancer

2.2 Roles and Responsibilities

Client

- Read records
- Write records
- Merge conflicting records

Load Balancer

- Randomly distribute requests to nodes

Node (Coordinator)

- Handle requests from load balancer
- Identify responsible storage nodes
- Updates vector clock
- Send and retrieve hinted replicas

Node (Node)

- Read from storage
- Write to storage
- Append versions
- Store hinted replicas and attempt hand-off

2.3 Assumptions

Similar Capacities

No need to take care of heterogeneity in the infrastructure

Transient Node Failures

Node failures are transient, no permanent failure of nodes

>3 to operate

Requires 3 nodes due to replication factor

Broadcasting ability

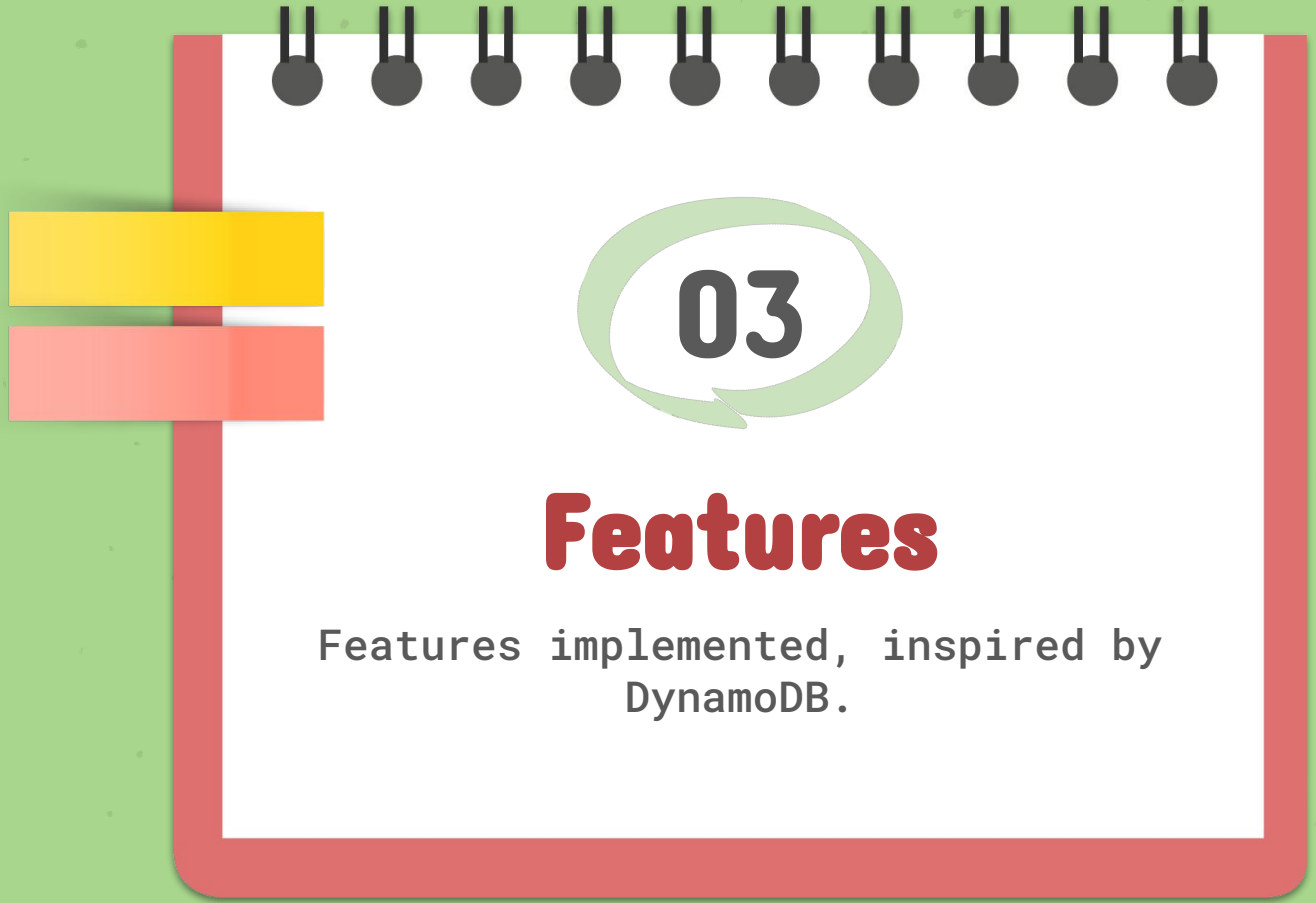
All nodes can broadcast to all other nodes

No Complete Data Loss

Hinted replicas are not lost therefore there is no complete loss of data

Used by internal services

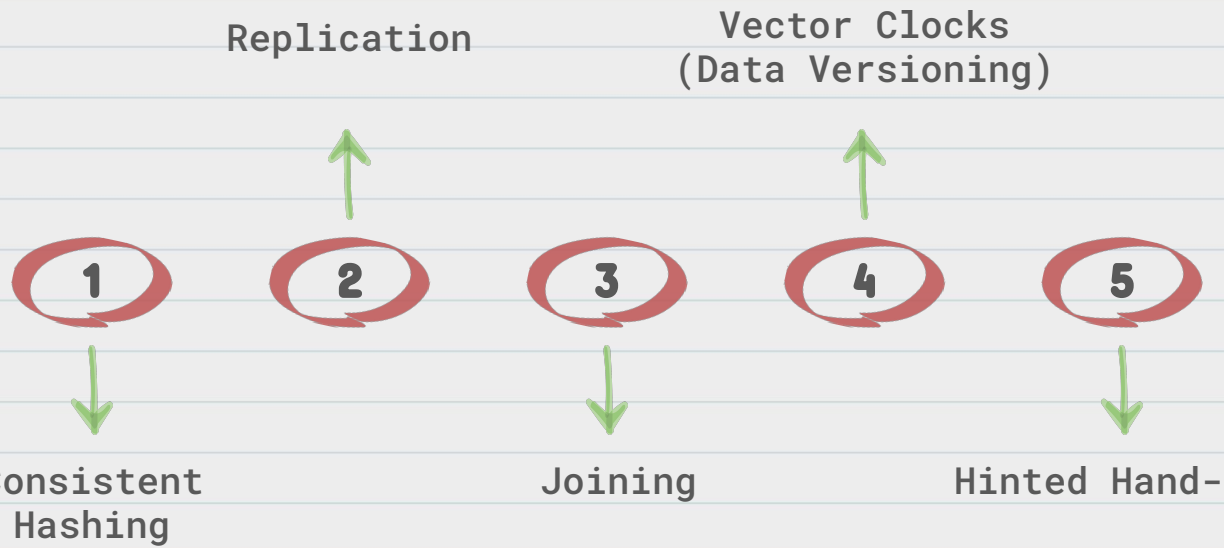
Operation environment assumed to be non-malicious.



Features

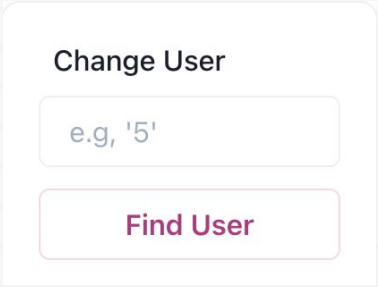
Features implemented, inspired by DynamoDB.

3. System's Features

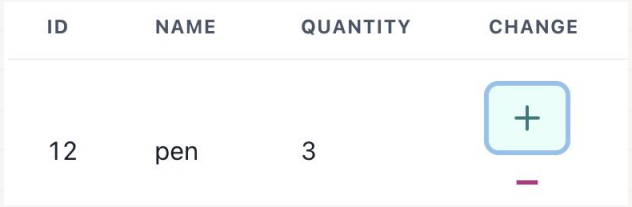


3.1 Our Front-End

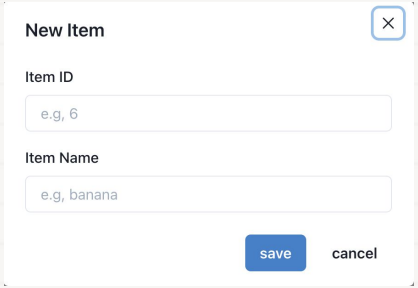
Find User



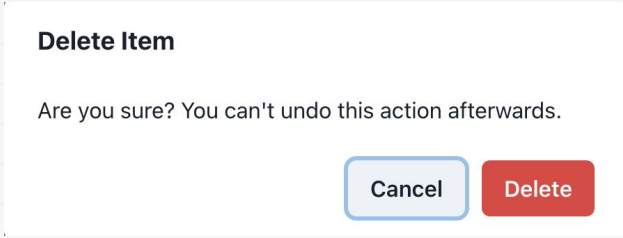
Increment/Decrement



New Item



Delete




Demo Time:
Front End
Cart
Interface!

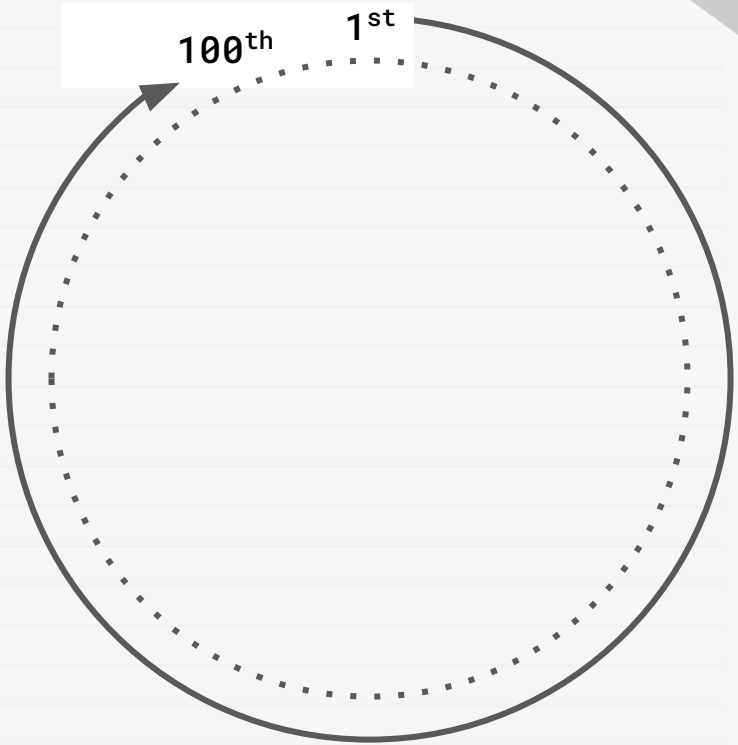
3.2 Consistent Hashing

Given that we have set the number of positions on the ring to 100.

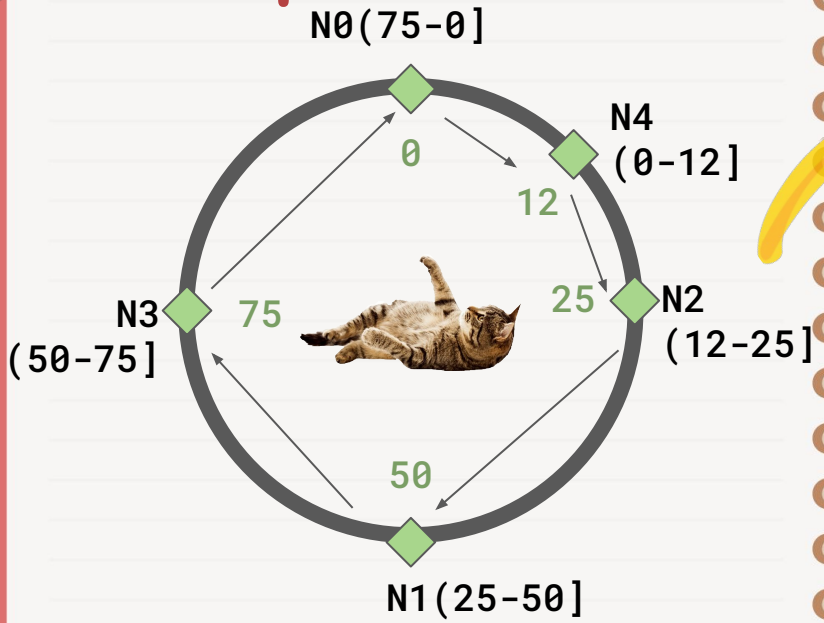
Position on the node = Md5(UserId.toBytes) mod 100

UserID —MD5→ Position*

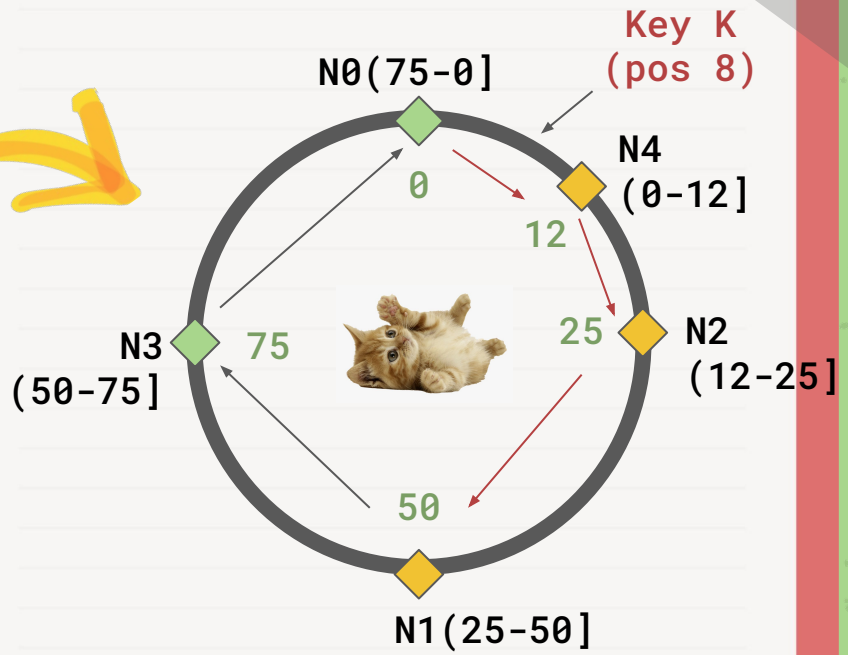
*Where 0 <= Position < 100



3.3 Replication



Each key is stored in the next 3 nodes along the ring

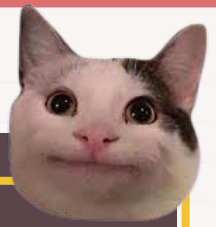


Key at position 8 is stored in nodes 4, 2 and 1.

3.3 Replication

```
2022/03/20 21:41:05 Input: 126
2022/03/20 21:41:05 Key position: 44
2022/03/20 21:41:05 First node position: 50
2022/03/20 21:41:05 Responsible nodes: [{Id:1 Ip:http://127.
0.0.1:8001 Port:0 Position:50} {Id:3 Ip:http://127.0.0.1:800
3 Port:0 Position:75} {Id:0 Ip:http://127.0.0.1:8000 Port:0
Position:0}]
2022/03/20 21:41:05 Successful operation for request type 0
2022/03/20 21:41:05 Successful operation for request type 0
2022/03/20 21:41:05 Successful operation for request type 0
```

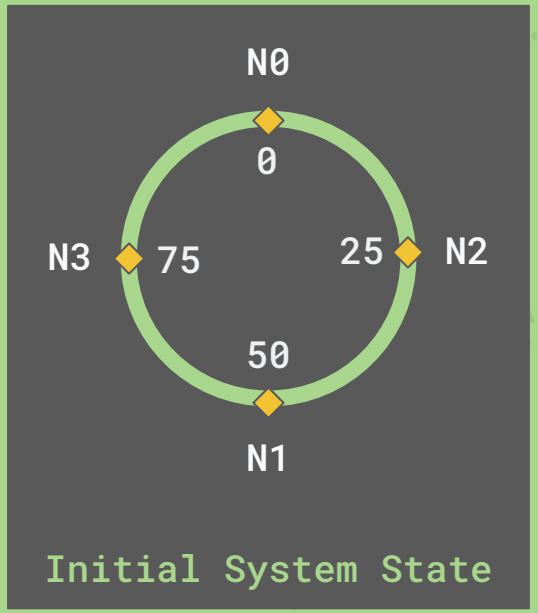
- Input “126” assigned position 44 on the ring
- Nodes 1, 3 and 0 are responsible for it

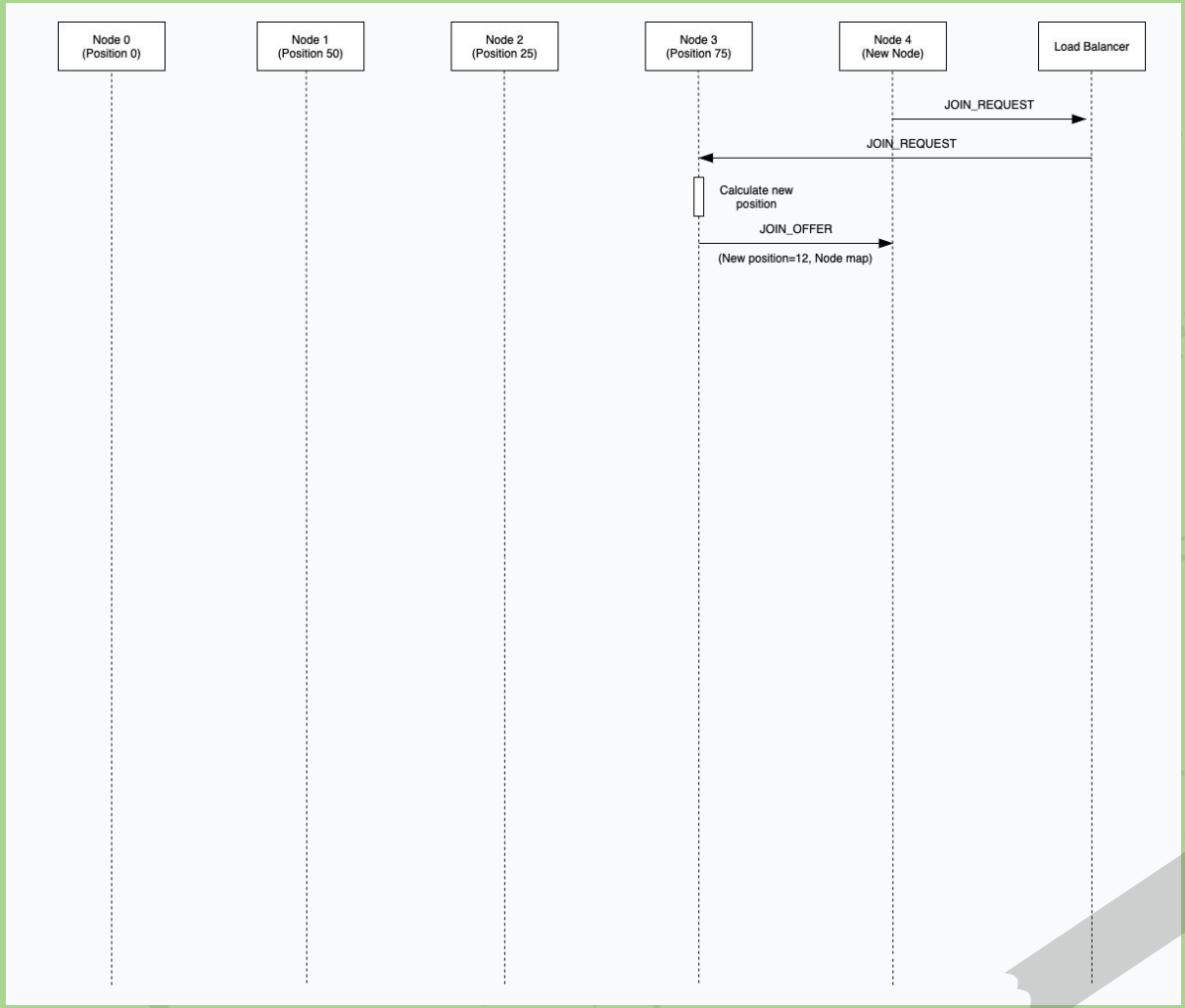


```
2022/03/20 21:17:33 Node 2 started
Press Enter to end
2022/03/20 21:18:59 Input: 123
2022/03/20 21:18:59 Key position: 8
2022/03/20 21:18:59 First node position: 12
2022/03/20 21:18:59 Responsible nodes: [{Id:4 Ip:http://127.
0.0.1:8004 Port:0 Position:12} {Id:2 Ip:http://127.0.0.1:800
2 Port:0 Position:25} {Id:1 Ip:http://127.0.0.1:8001 Port:0
Position:50}]
2022/03/20 21:18:59 Read request completed for {123 map[12:{
11 pencil case 3} 13:{14 fountain pen 3}] [0 0 1 0 0]}
2022/03/20 21:18:59 Successful operation for request type 0
2022/03/20 21:18:59 Successful operation for request type 0
2022/03/20 21:18:59 Successful operation for request type 0
```

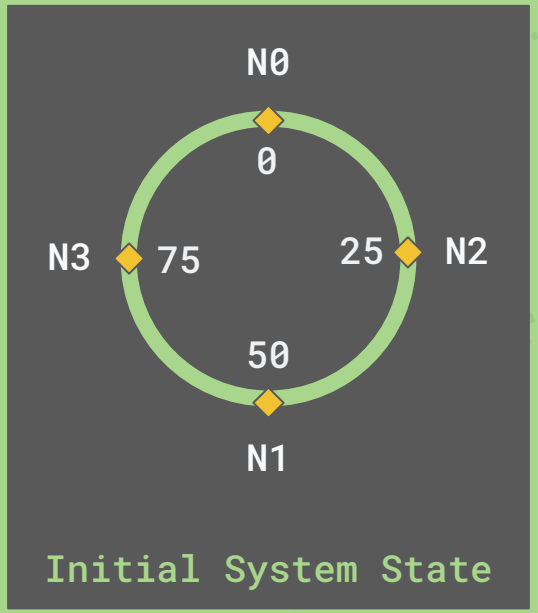
- Input “123” assigned position 8 on the ring
- Nodes 4, 2 and 1 are responsible for it

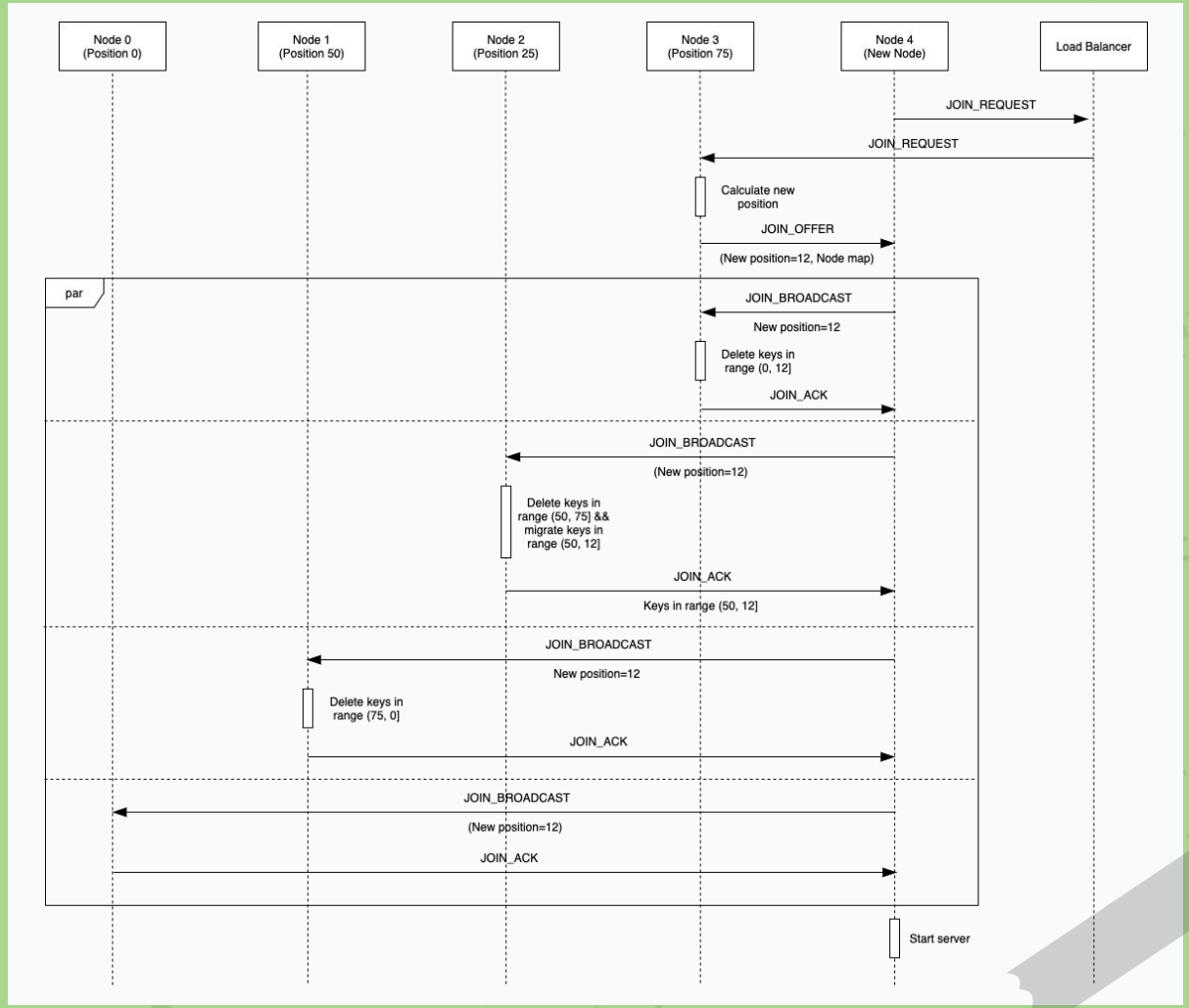
3.4 Joining


N0
0
N3
75
25
N2
50
N1
Initial System State

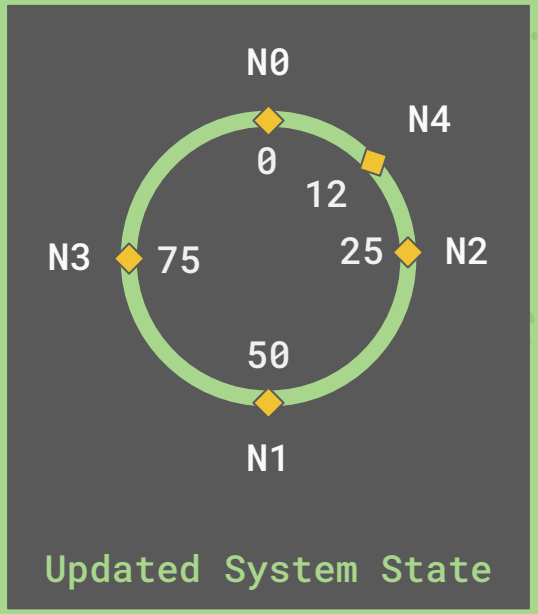

Node 0
(Position 0)
Node 1
(Position 50)
Node 2
(Position 25)
Node 3
(Position 75)
Node 4
(New Node)
Load Balancer
JOIN_REQUEST
JOIN_REQUEST
Calculate new position
JOIN_OFFER
(New position=12, Node map)

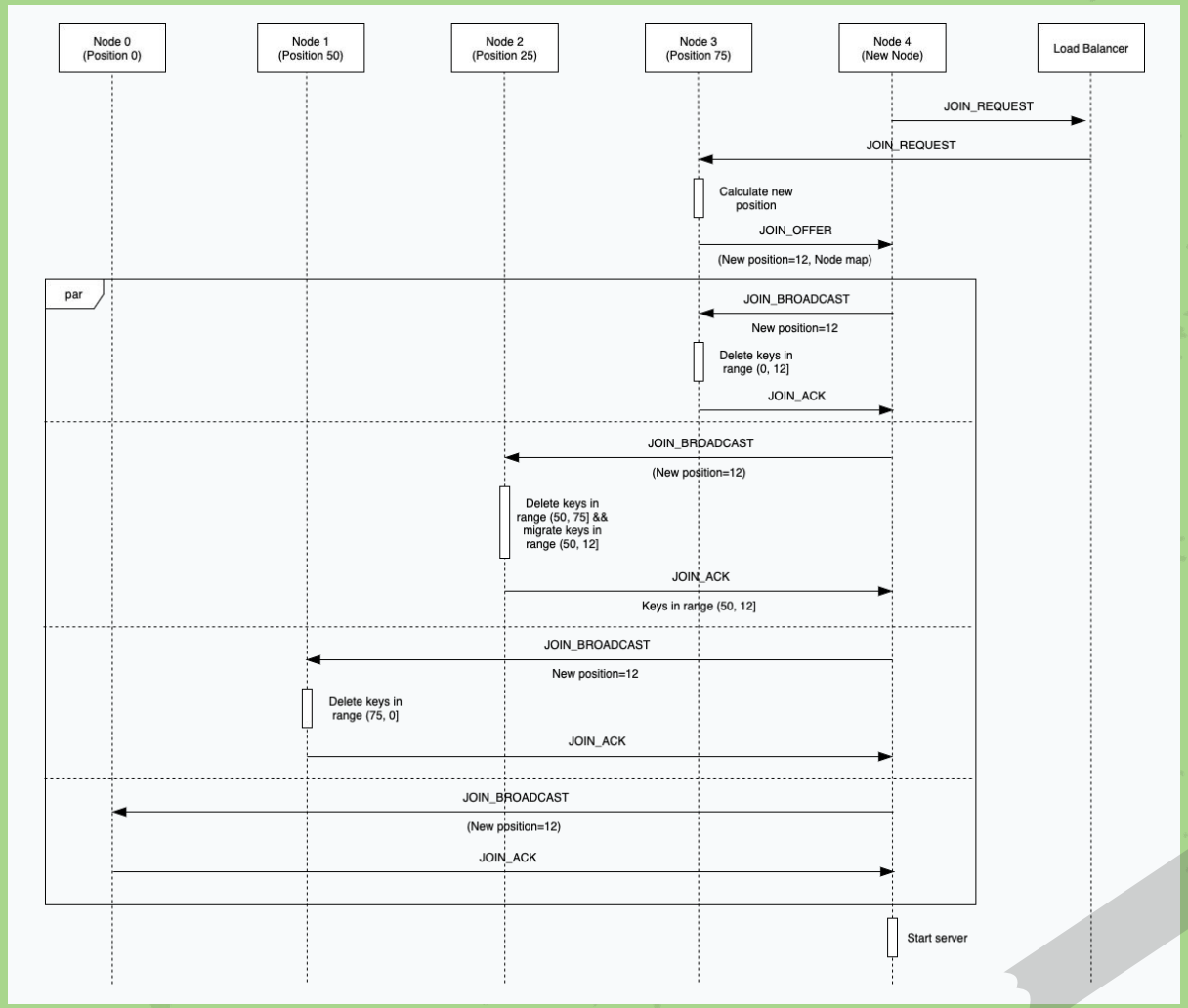
3.4 Joining


N0
0
N3
75
25
N2
50
N1
Initial System State


Node 0 (Position 0)
Node 1 (Position 50)
Node 2 (Position 25)
Node 3 (Position 75)
Node 4 (New Node)
Load Balancer
JOIN_REQUEST
JOIN_REQUEST
Calculate new position
JOIN_OFFER
(New position=12, Node map)
par
JOIN_BROADCAST
New position=12
Delete keys in range (0, 12]
JOIN_ACK
JOIN_BROADCAST
(New position=12)
Delete keys in range (50, 75] && migrate keys in range (50, 12]
JOIN_ACK
Keys in range (50, 12]
JOIN_BROADCAST
New position=12
Delete keys in range (75, 0]
JOIN_ACK
JOIN_BROADCAST
(New position=12)
JOIN_ACK
Start server

3.4 Joining

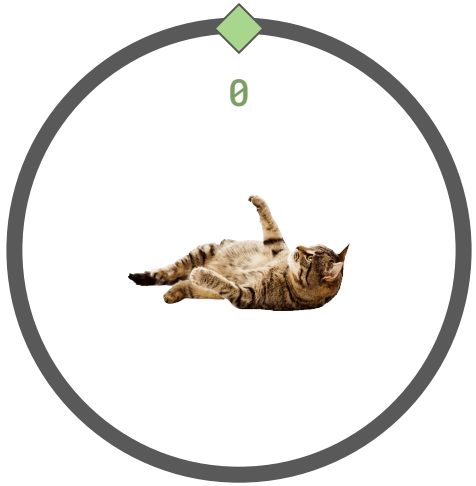

N0
N4
0
12
N3
75
25
N2
50
N1
Updated System State


Node 0 (Position 0)
Node 1 (Position 50)
Node 2 (Position 25)
Node 3 (Position 75)
Node 4 (New Node)
Load Balancer
JOIN_REQUEST
JOIN_REQUEST
Calculate new position
JOIN_OFFER
(New position=12, Node map)
par
JOIN_BROADCAST
New position=12
Delete keys in range (0, 12]
JOIN_ACK
JOIN_BROADCAST
(New position=12)
Delete keys in range (50, 75] && migrate keys in range (50, 12]
JOIN_ACK
Keys in range (50, 12]
JOIN_BROADCAST
New position=12
Delete keys in range (75, 0]
JOIN_ACK
JOIN_BROADCAST
(New position=12)
JOIN_ACK
Start server

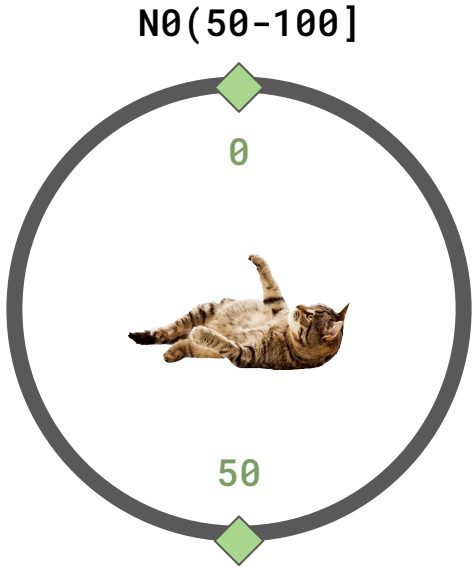

Demo Time:
Joining!

1. Node 0

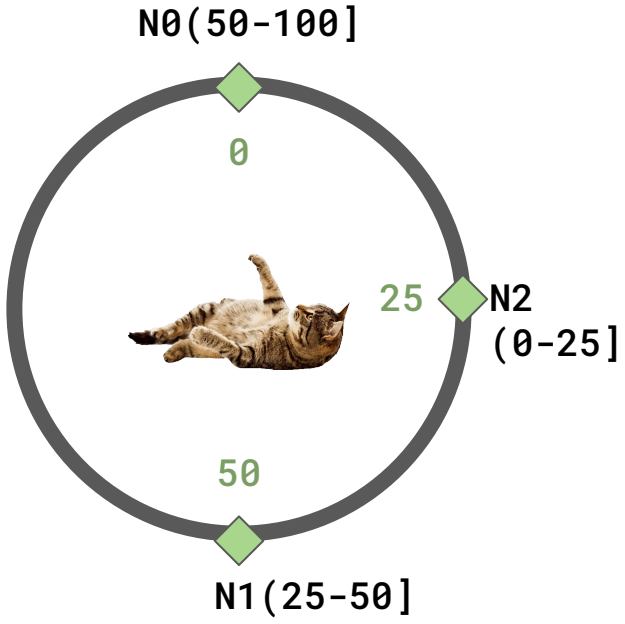
N0(0-100]



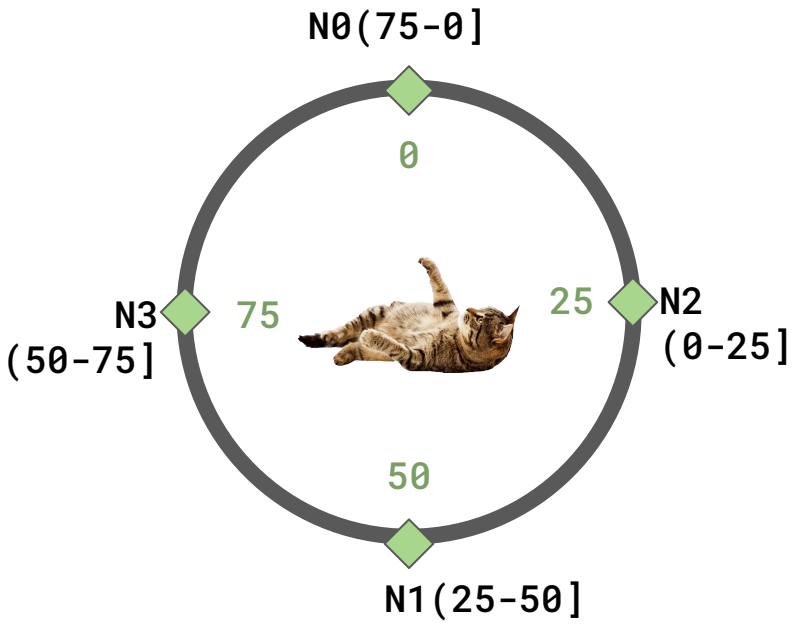
2. Node 1



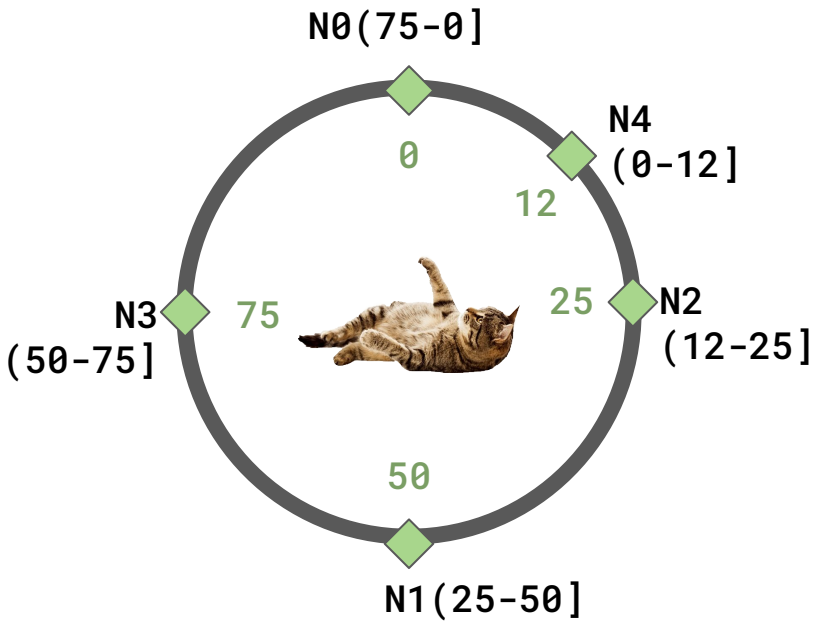
3. Node 2



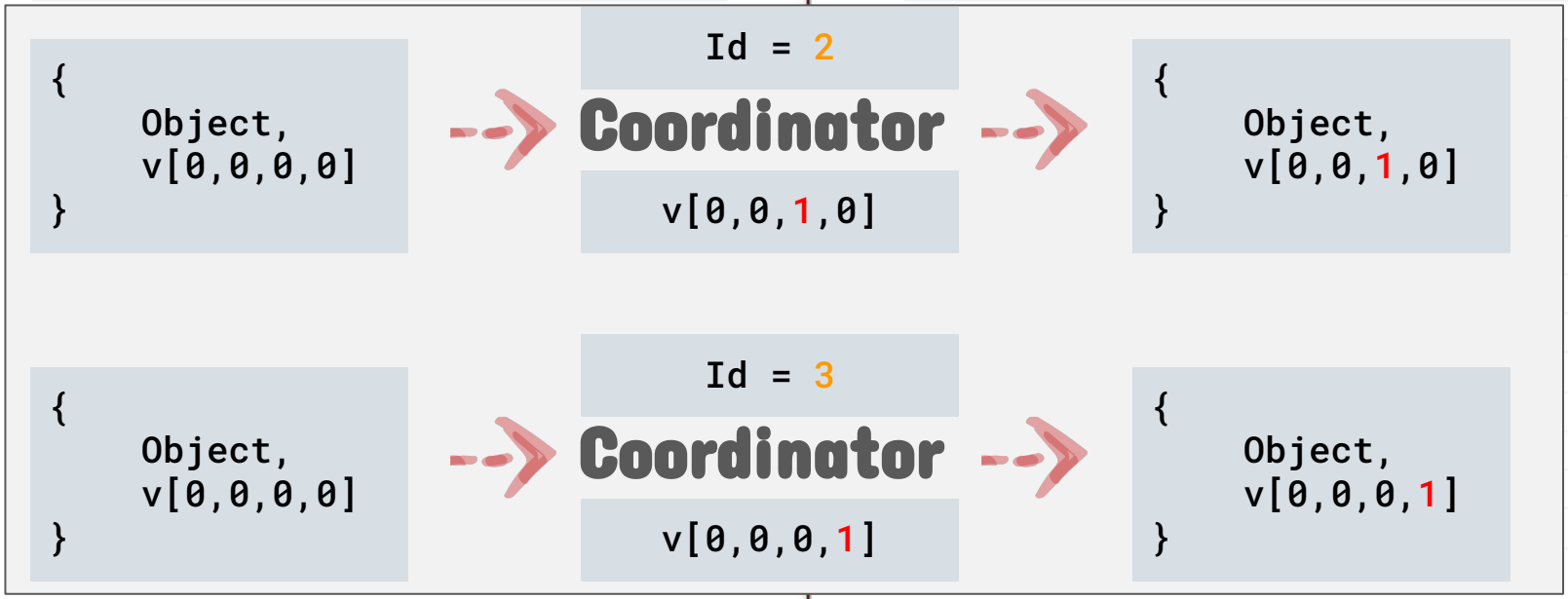
4. Node 3



5. Node 4

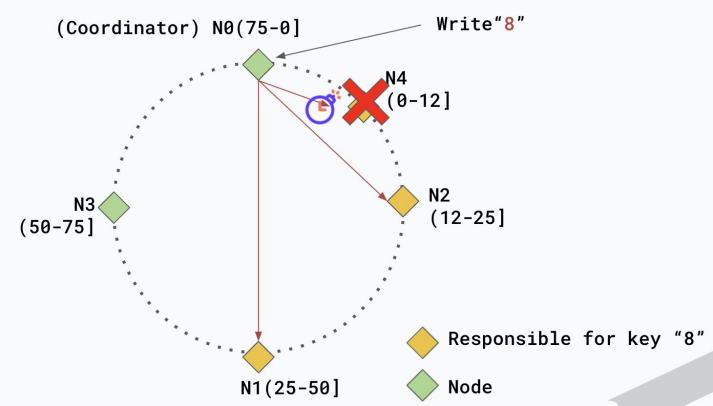


3.5 Vector Clock

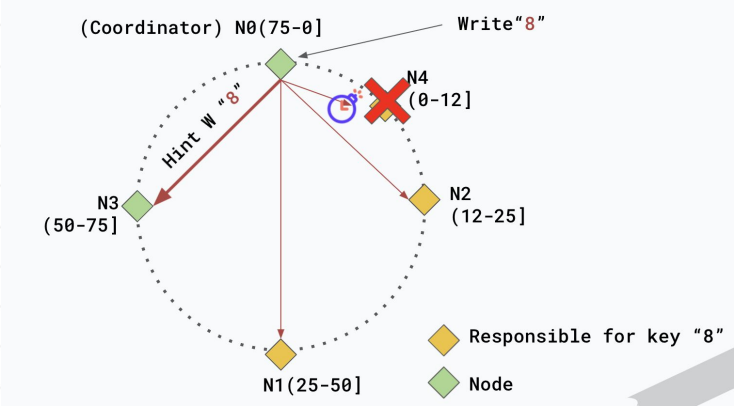



Demo Time:
Front End
Vector Clock!

3.6 Hinted Hand-off



Write to N4 times out



Hinted write to N3

Will be explained in further detail later under fault tolerance!

04

Scalability

4. Scalability

Joining

New nodes can be added during traffic spikes

New position calculated based on largest gap

Load Balancing

Coordinator is selected by the load balancer

4. Scalability Experiments

- Load tests ran using Vegeta
- Nodes, load balancer and attacking host were run on a single machine
- Varied number of nodes (5, 10, 20) and request rates (10/s, 100/s, 500/s)
- Measured success rate, average latency and 99th percentile latency
- Sent read-only requests, write-only requests and a mix of read and write requests with random keys for each test
- Ran each test case 3 times and took average values
- Each test lasted for 1 min

4. Scalability Experiments

Vegeta

10 requests/s

100 requests/s

500 requests/s

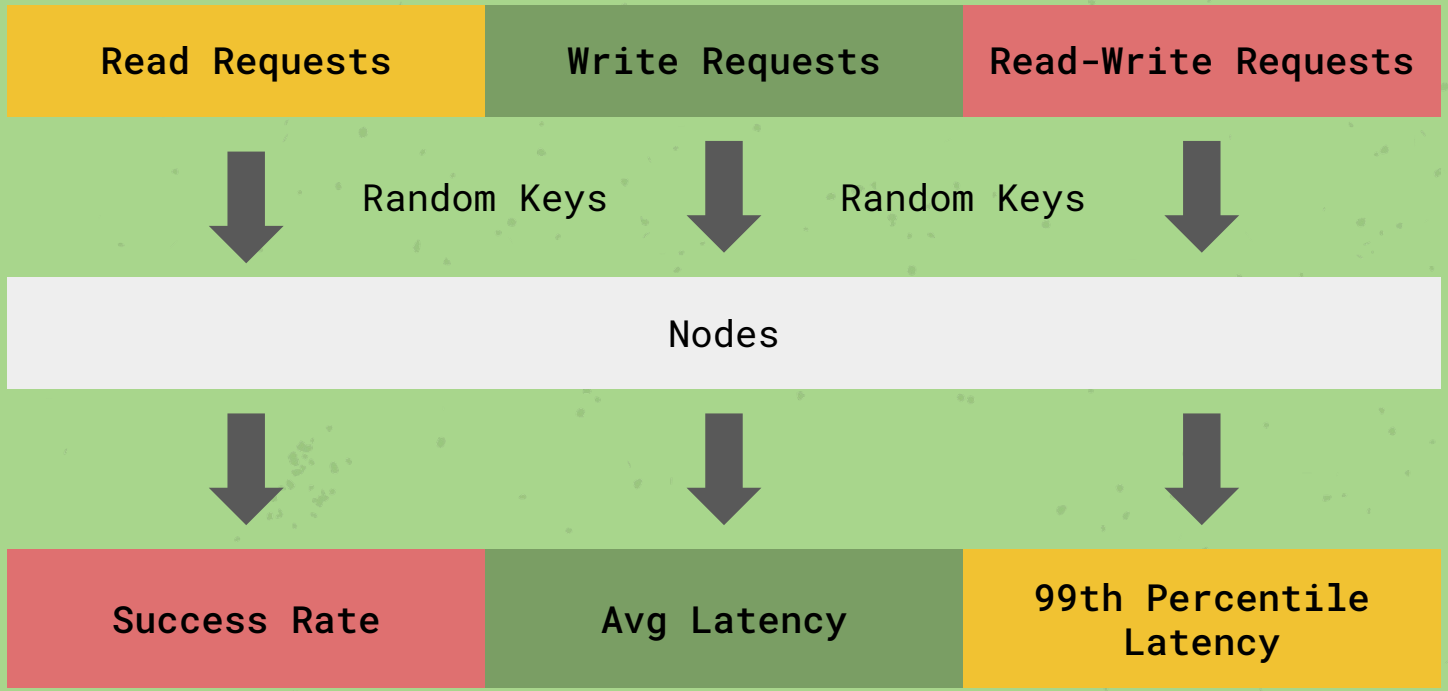
Case 1: 5 Nodes

Case 2: 10 Nodes

Case 3: 20 Nodes

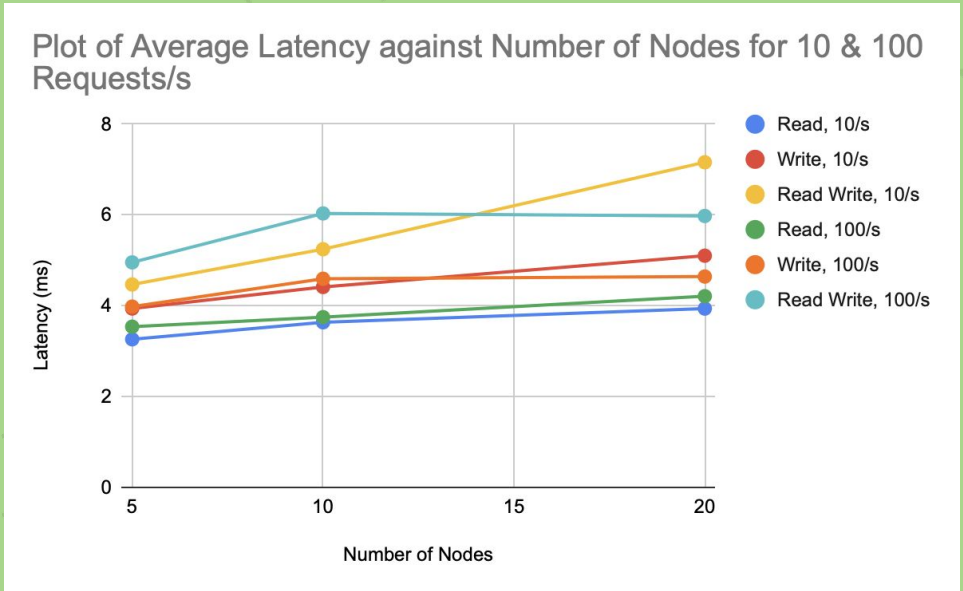
Read Requests	Write Requests	Read-Write Requests

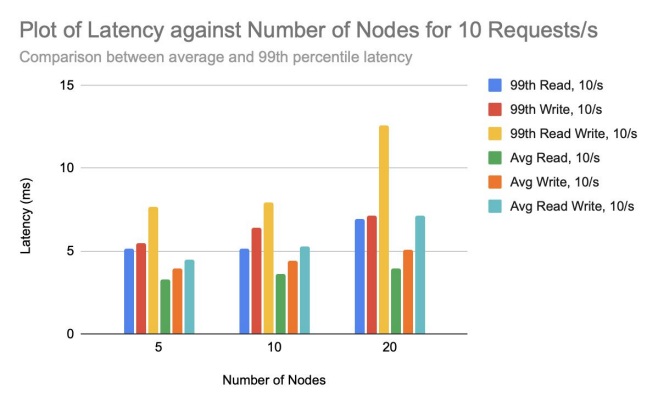
4. Scalability Experiments

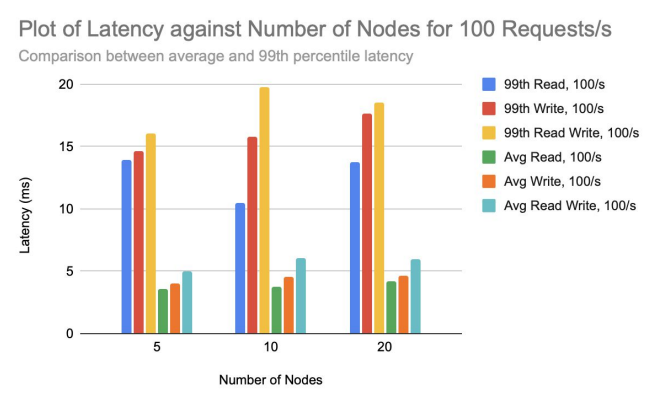


4. Scalability Experiments

Request Rate: 10/s & 100/s

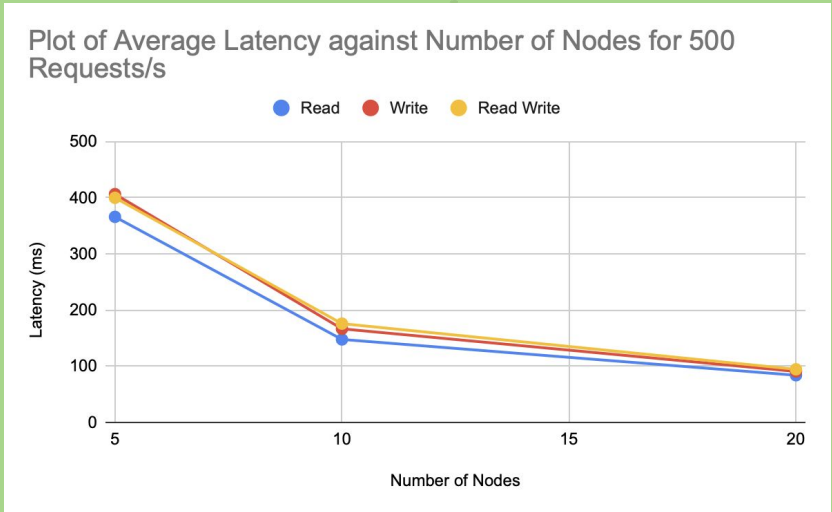


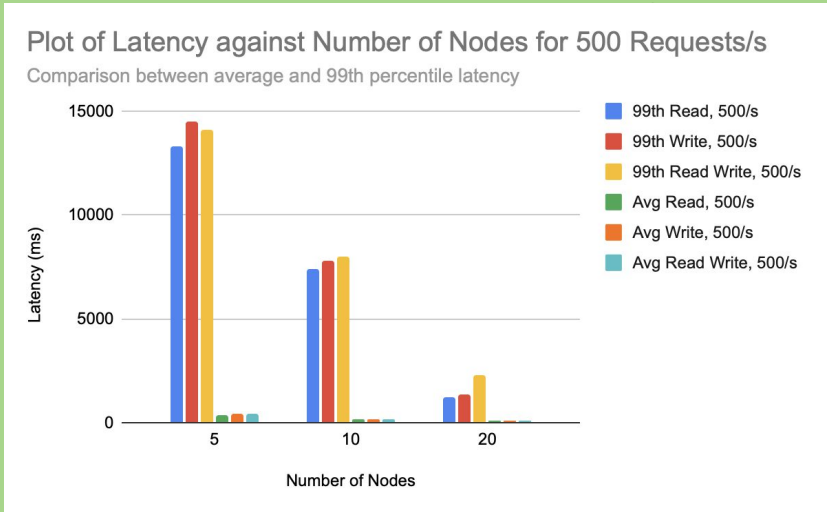




4. Scalability Experiments

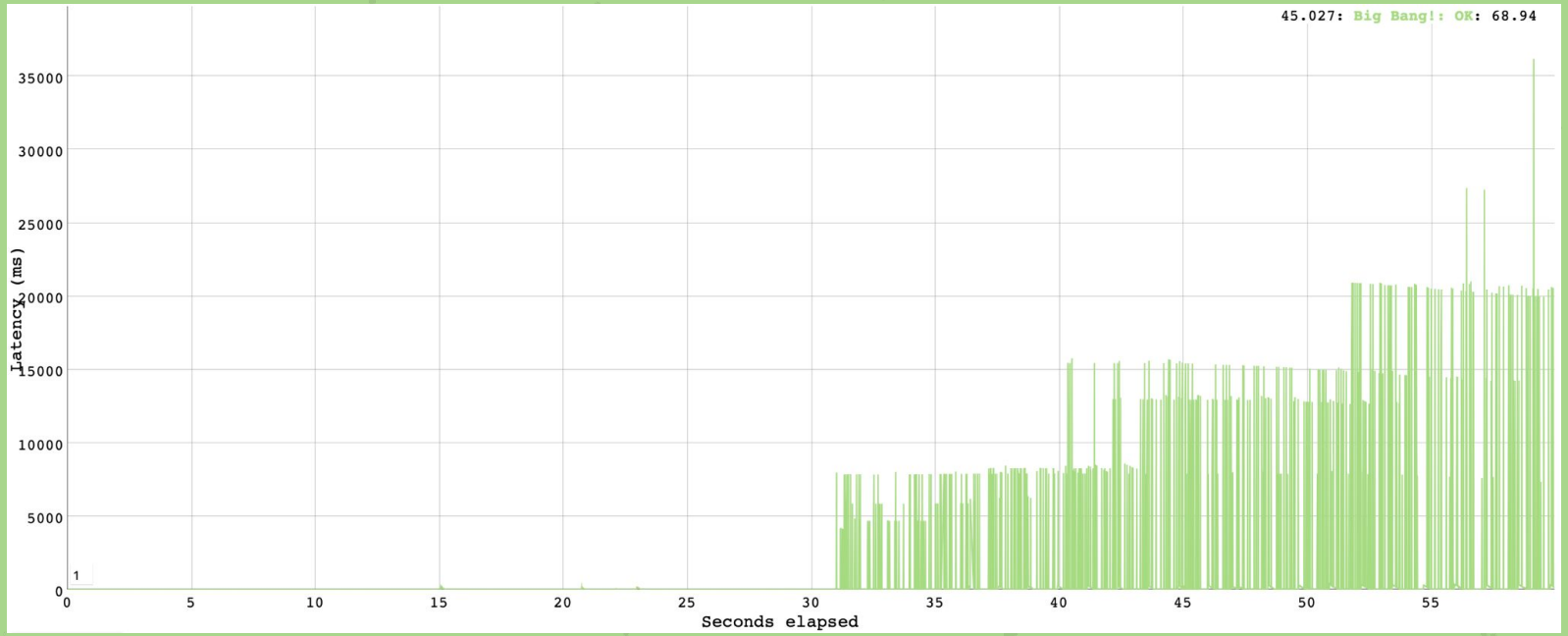
Request Rate: 500/s





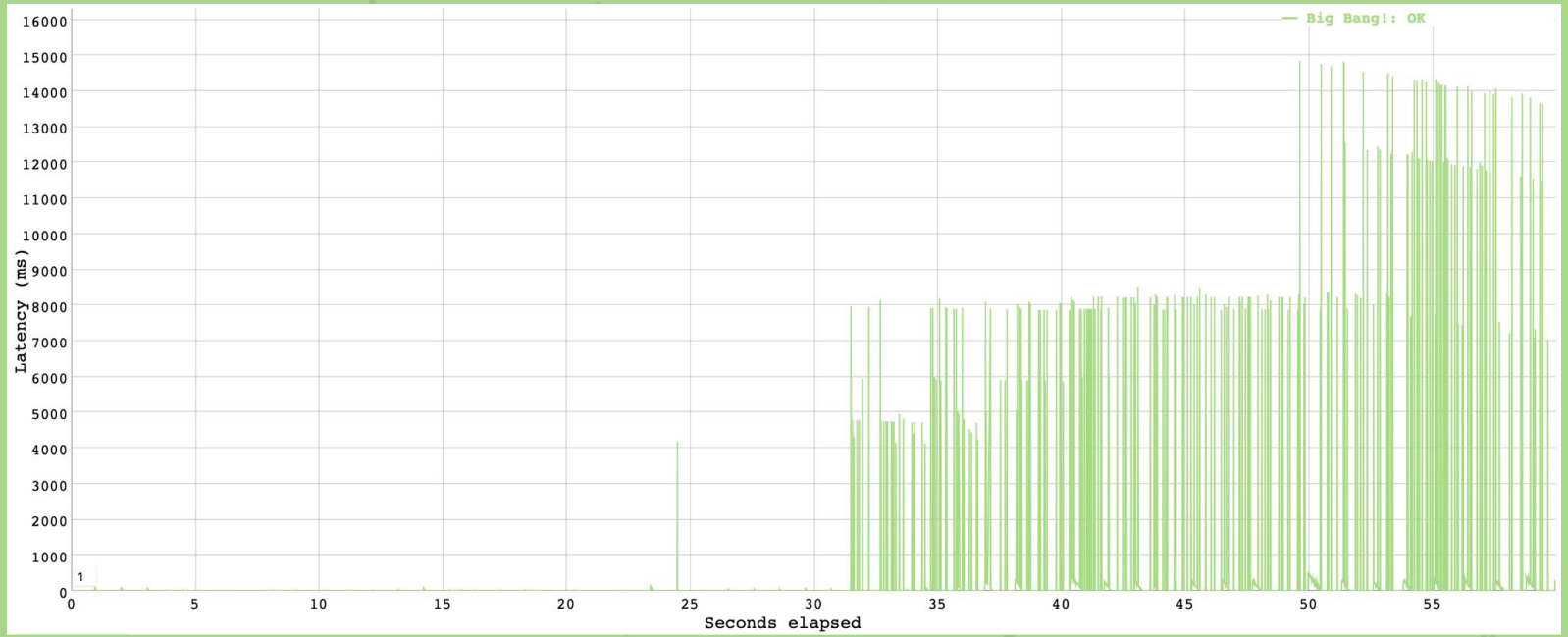
4. Scalability Experiments

Distribution of latencies for 5 nodes and 500 requests/s



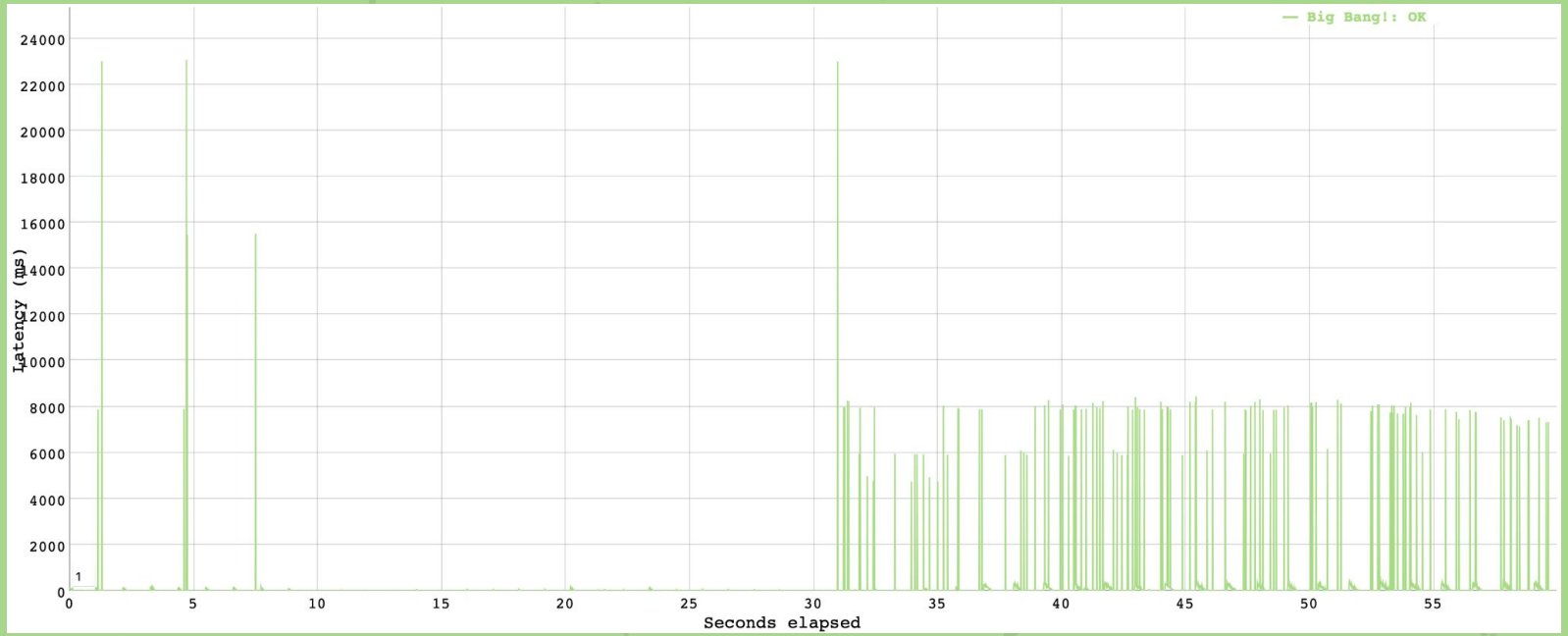
4. Scalability Experiments

Distribution of latencies for 10 nodes and 500 requests/s



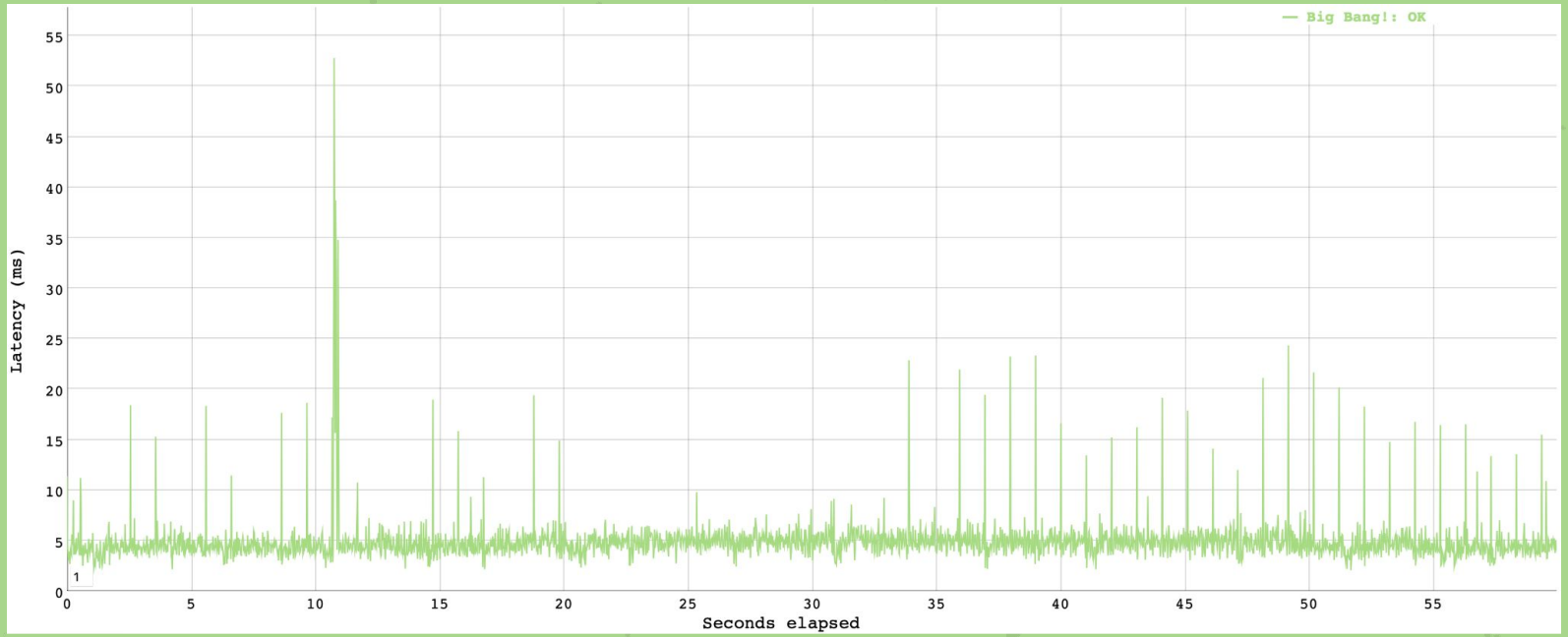
4. Scalability Experiments

Distribution of latencies for 20 nodes and 500 requests/s



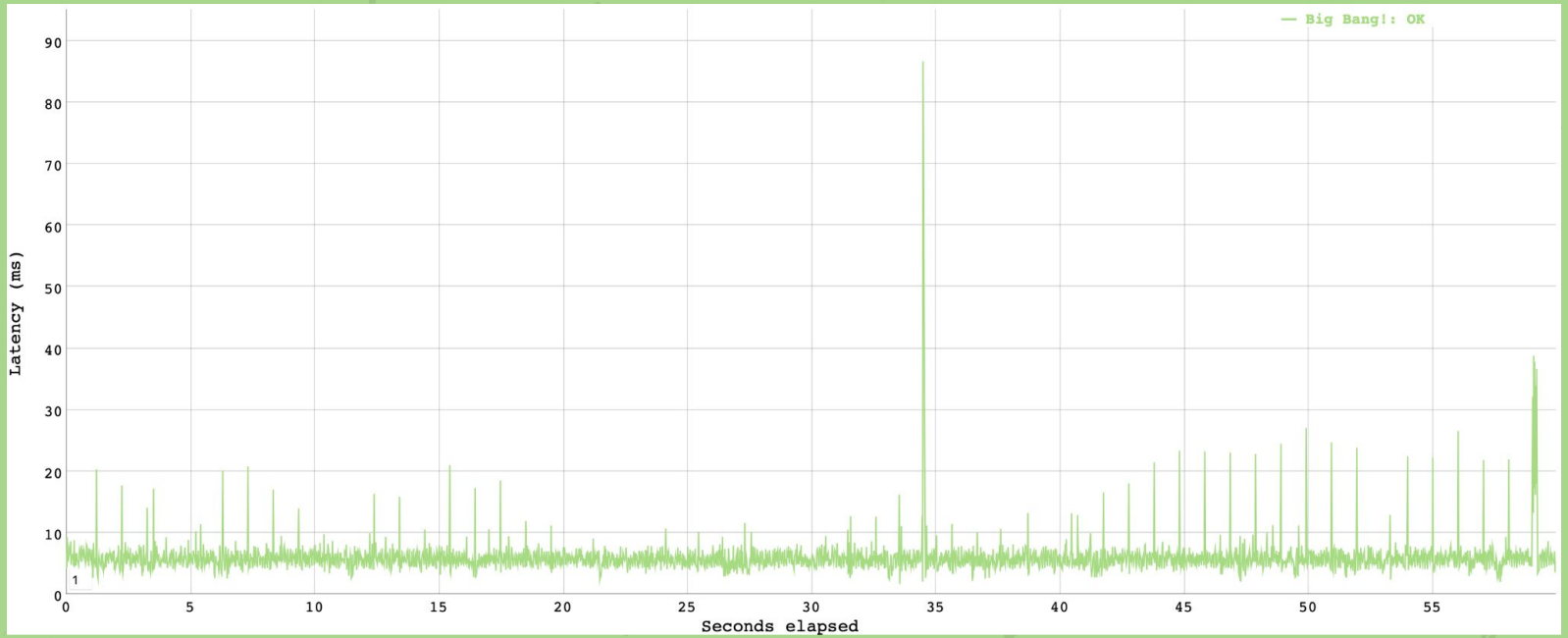
4. Scalability Experiments

Distribution of latencies for 5 nodes and 100 requests/s



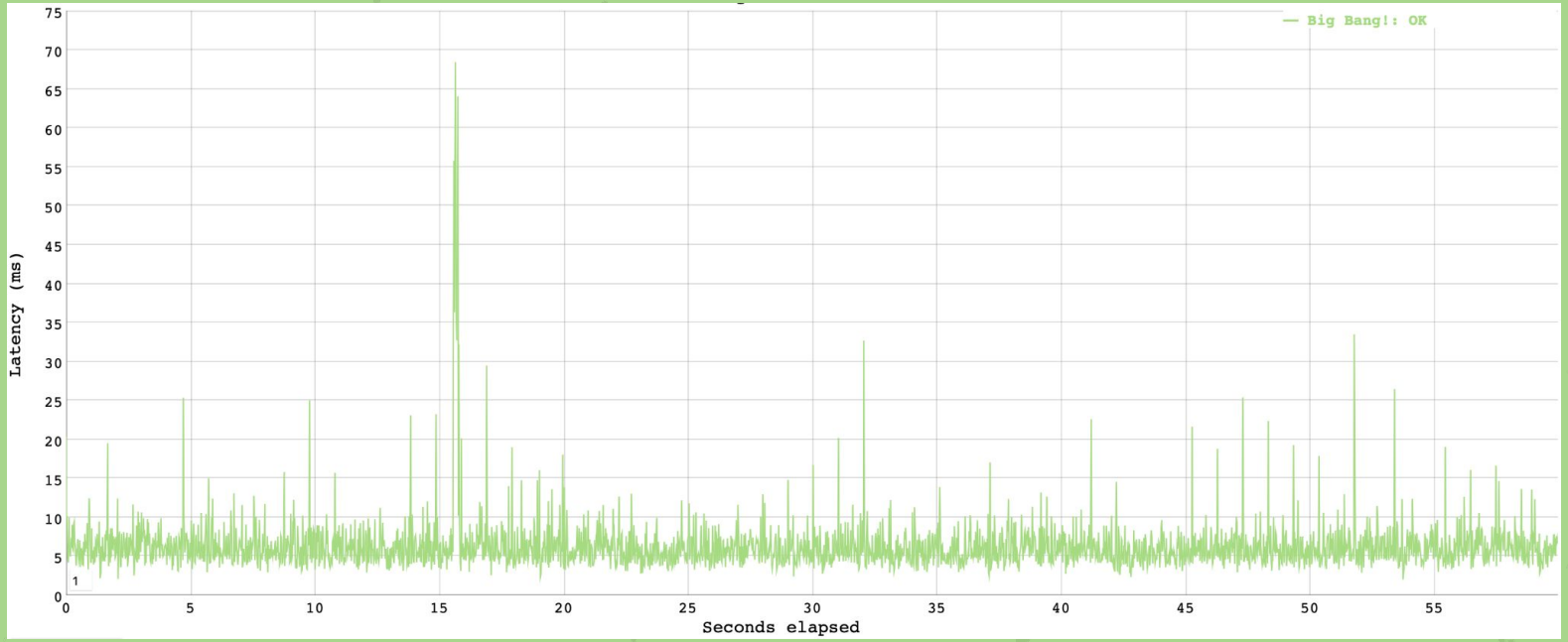
4. Scalability Experiments

Distribution of latencies for 10 nodes and 100 requests/s



4. Scalability Experiments

Distribution of latencies for 20 nodes and 100 requests/s



05

Consistency

5. Consistency

Main mechanism that ensures Eventual Consistency in our system

Vector Clock Conflict Detection → Write Always-Append Mechanism → Raise version-conflict to Client via HTTP response + forward different versions for client-side merging

5. Consistency

How this works

Concurrent Write requests:

obj1[0,0,0,0,1]

obj2[1,0,0,0,0]

Ambiguous Ordering Detected

Append **BOTH** object versions to memory

Raise version-conflict to Client via HTTP response + forward **BOTH** versions for client-side merging

Note: Conflict Resolution should be handled by the service using DynamoDB, in this case, our front-end application will be executing a merge on conflicts via some predefined rules.

5. Consistency

A closer look at the append mechanism

Assume one of the writes reach the node first:

Currently in memory:

Key1: [obj1[0,0,0,0,1]]

Incoming write:

Key1: obj2[1,0,0,0,0]

Before writing, compare Vector Clocks

[0,0,0,0,1]

[1,0,0,0,0]

Ambiguous Ordering Detected

New memory state:

Key1: [obj1[0,0,0,0,1], obj2[1,0,0,0,0]]

5. Consistency

2 scenarios we considered

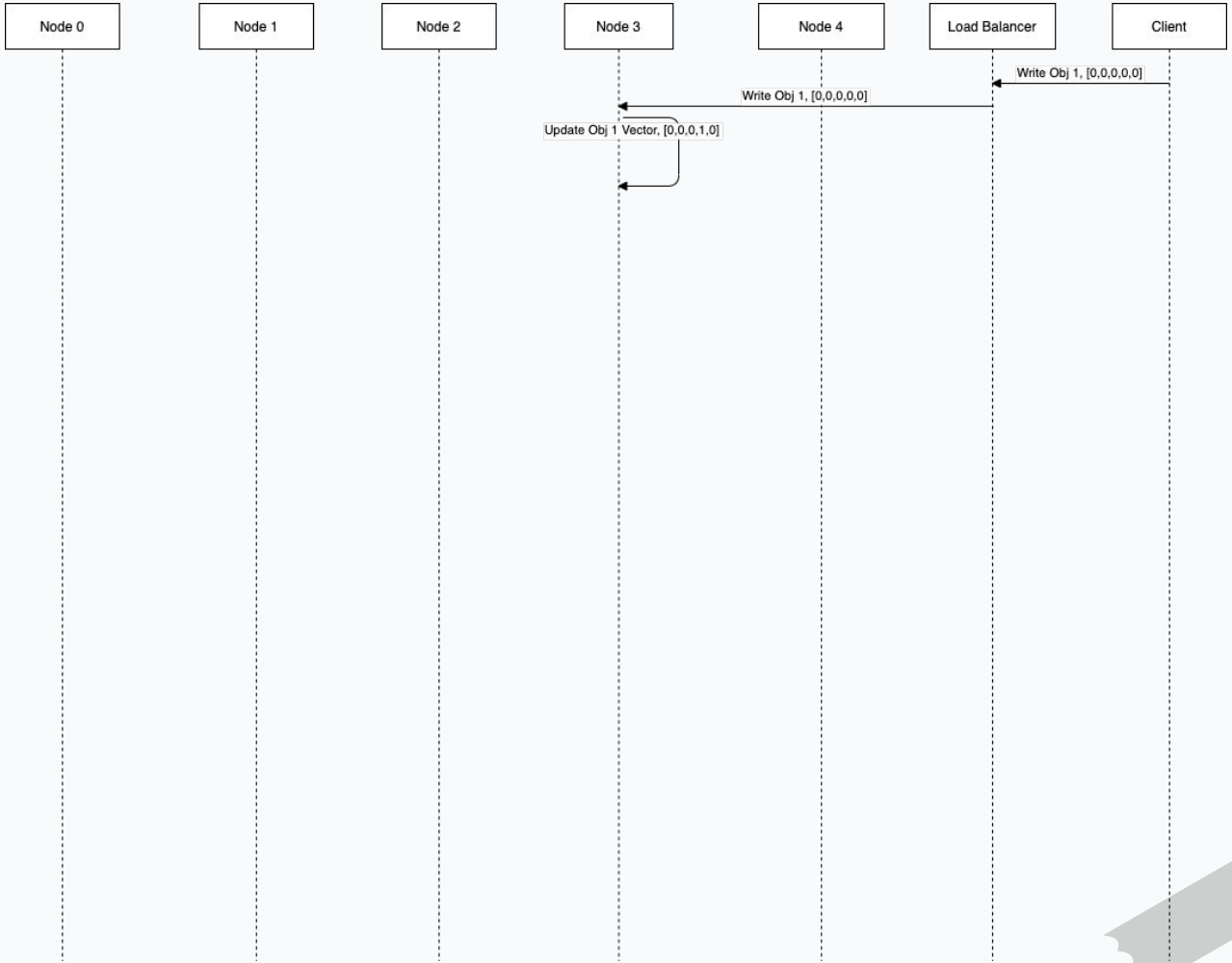
Sequential Write Requests

Concurrent Write Requests

5. Consistency

Sequential Write Requests

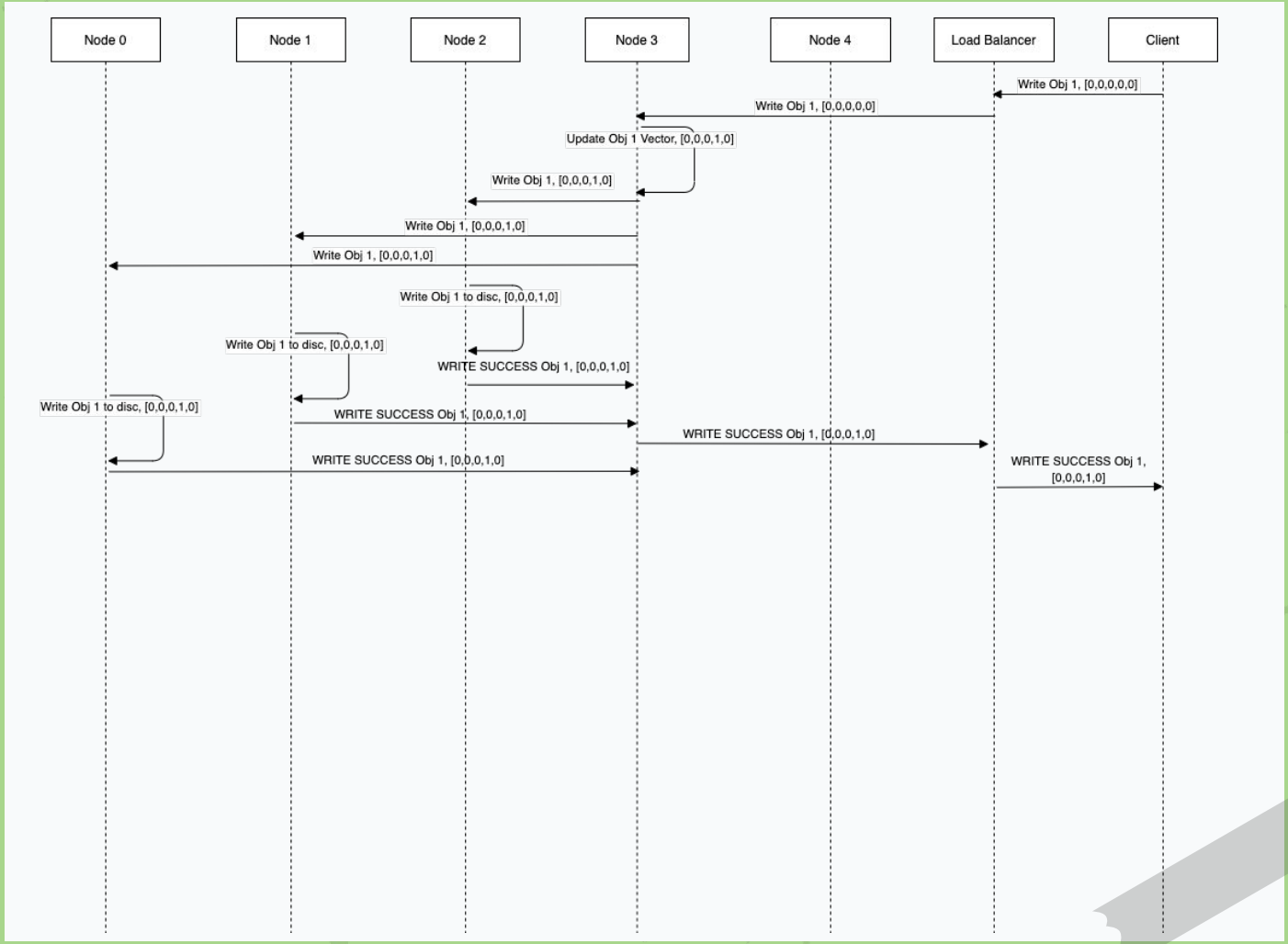
S/n	Write	Data
1	Obj 1 [00000]	Obj 1 [00010]
2		



5. Consistency

Sequential Write Requests

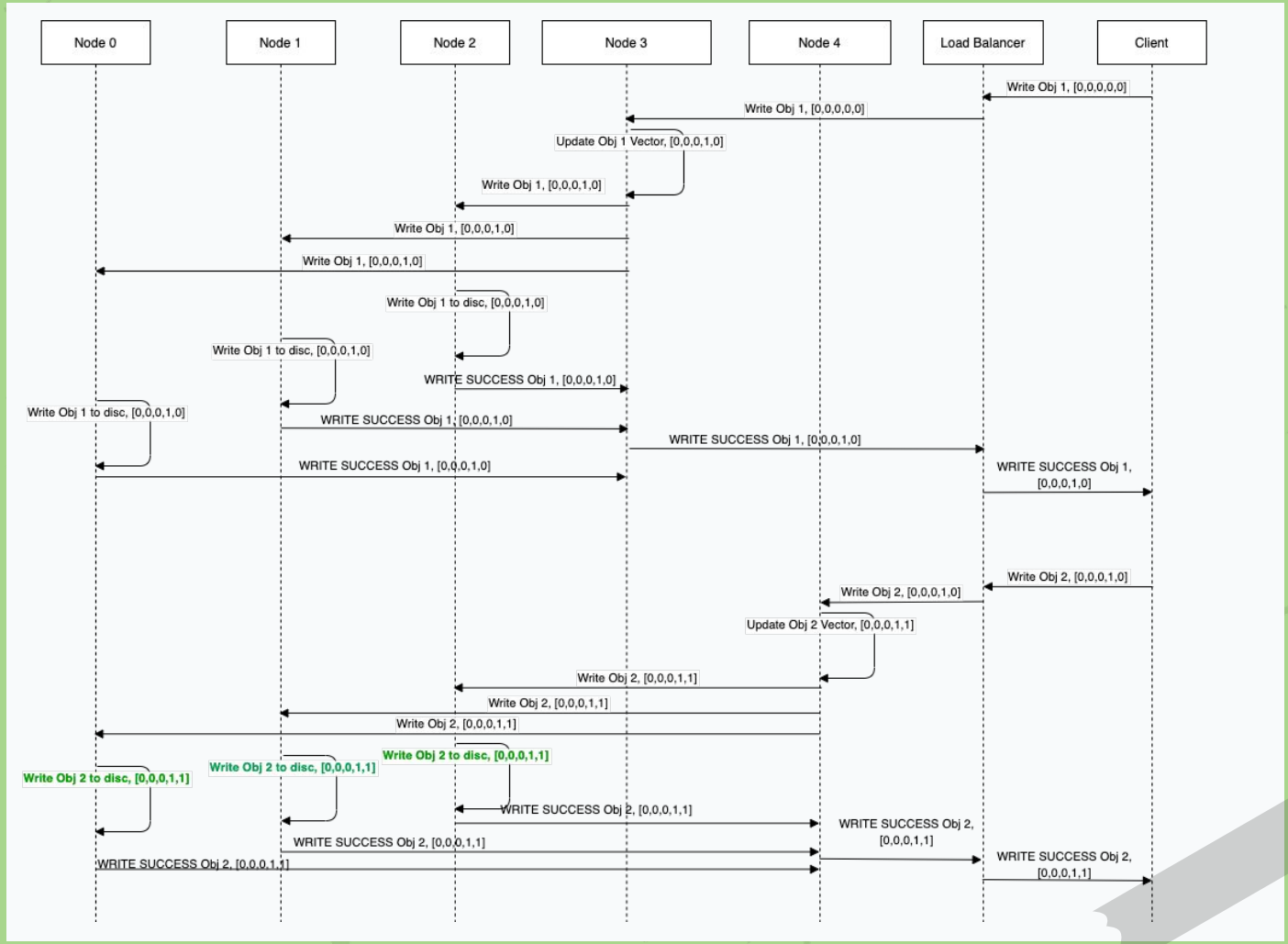
S/n	Write	Data
1	Obj 1 [00000]	Obj 1 [00010]
2		



5. Consistency

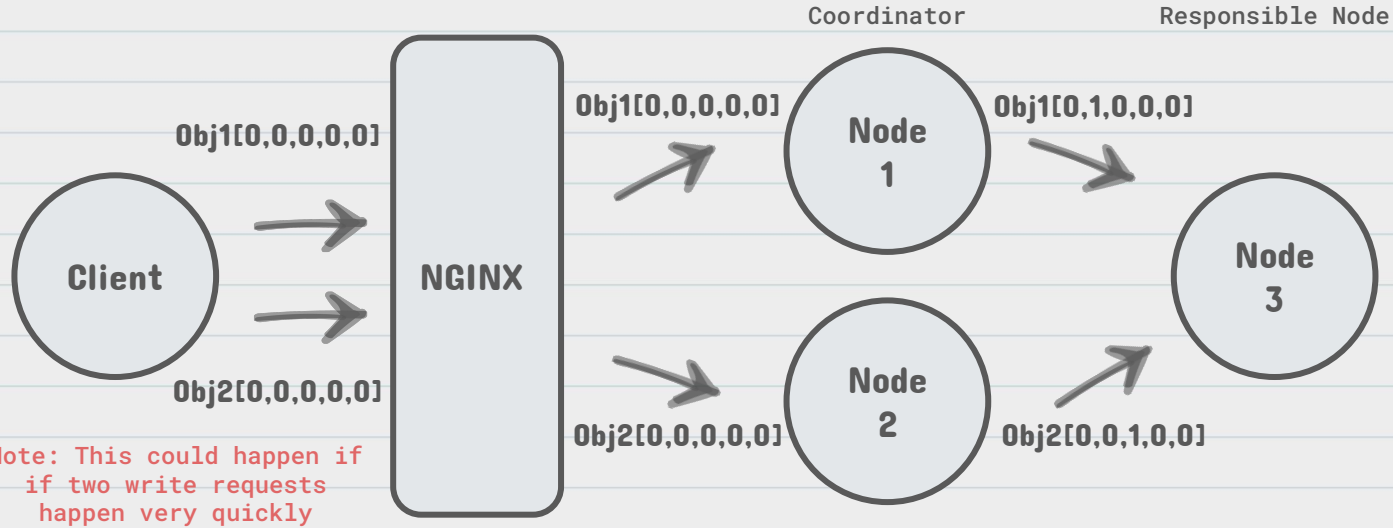
Sequential Write Requests

S/n	Write	Data
1	Obj 1 [00000]	Obj 1 [00010]
2	Obj 2 [00010]	Obj 2 [00011]



5. Consistency

How 2 versions of the same cart could possibly exist during concurrent writes

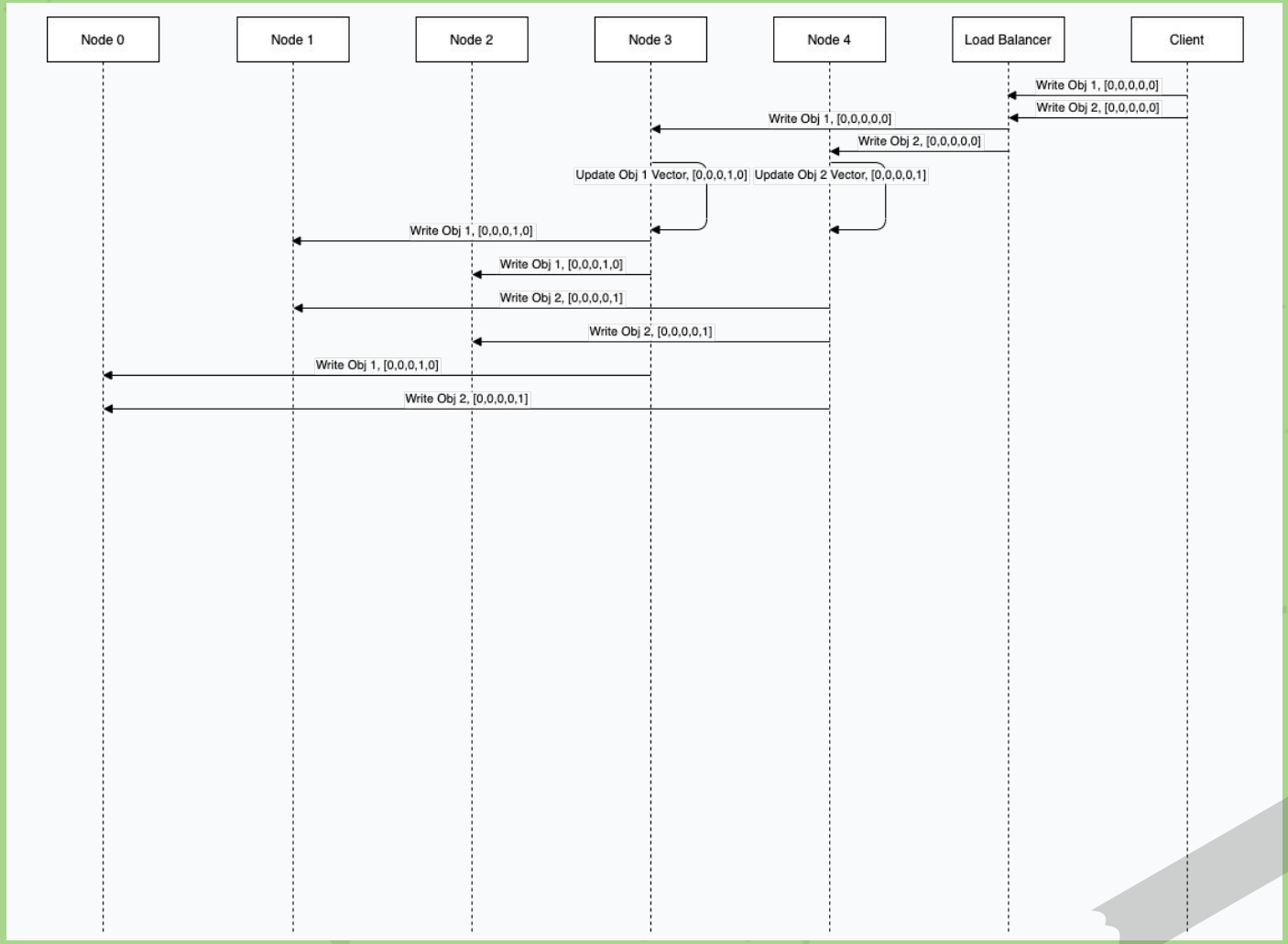


Note: This could happen if if two write requests happen very quickly

5. Consistency

Concurrent Write Requests

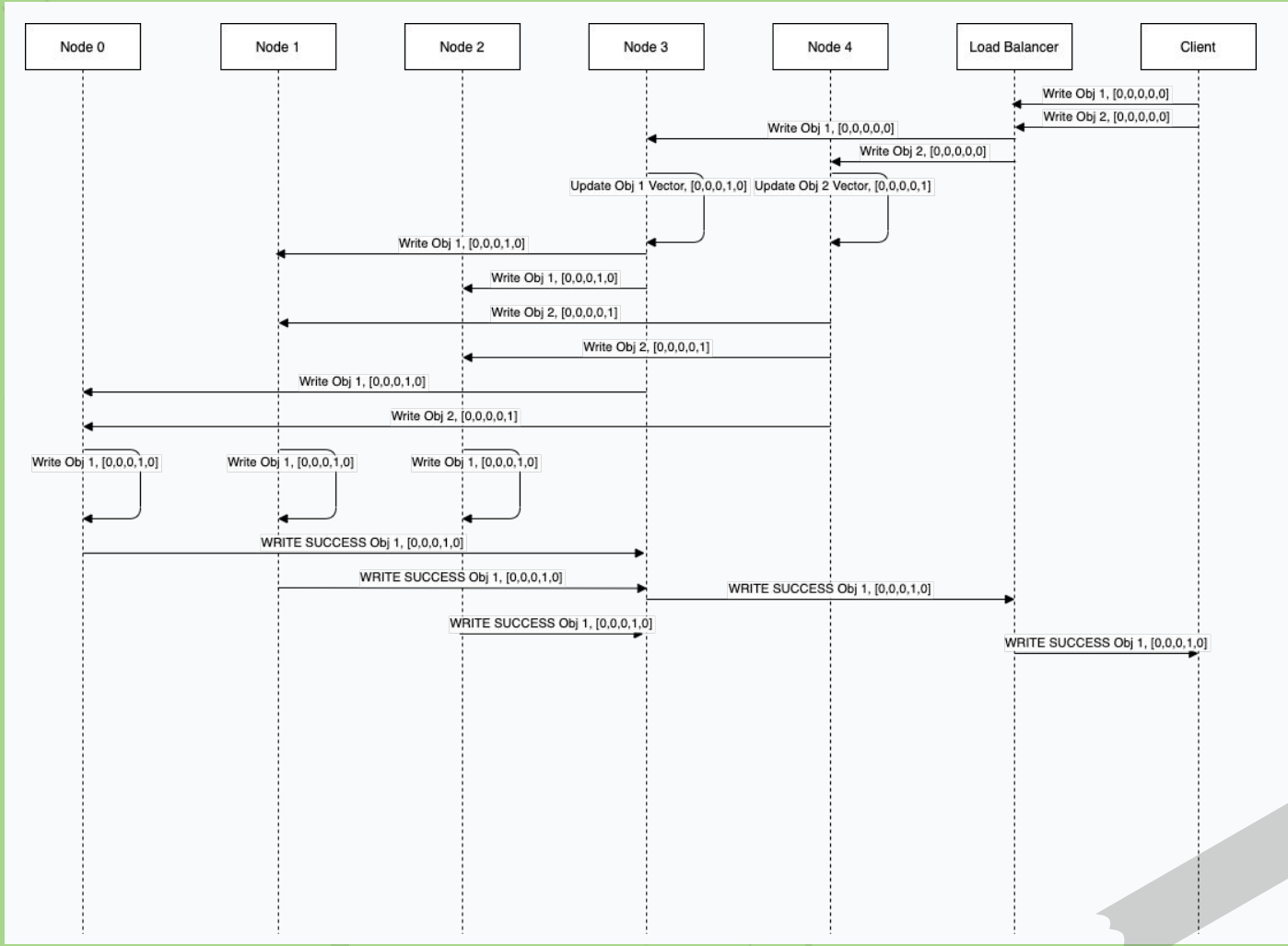
S/n	Write	Data
1		
2		



5. Consistency

Concurrent Write Requests

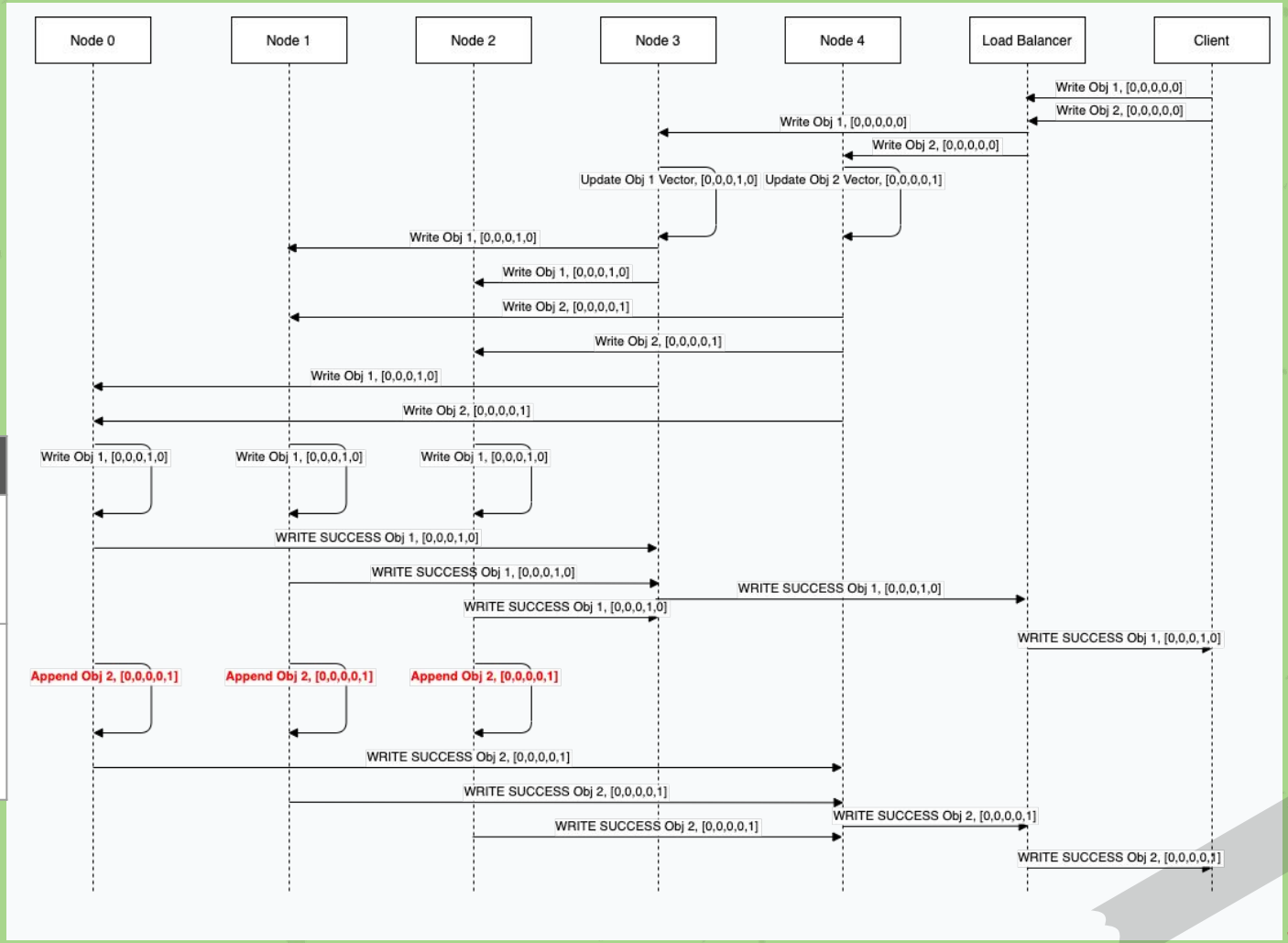
S/n	Write	Data
1	Obj 1 [00000]	Array{ Obj 1 [00010] }
2		



5. Consistency

Concurrent Write Requests

S/n	Write	Data
1	Obj 1 [00000]	Array{ Obj 1 [00010] }
2	Obj 2 [00000]	Array { Obj1 [00010, Obj 2 [00001] }

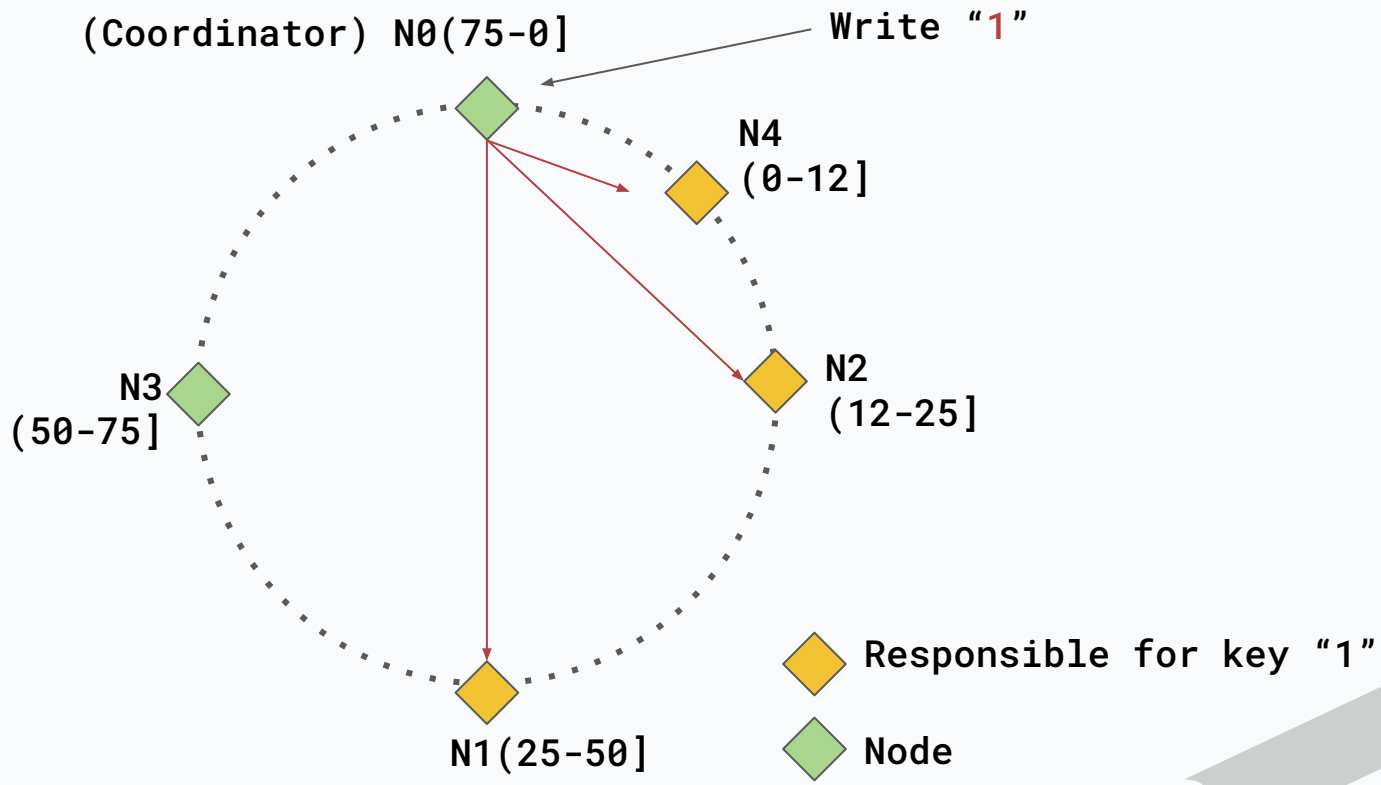



Demo Time:
Consistency!

06

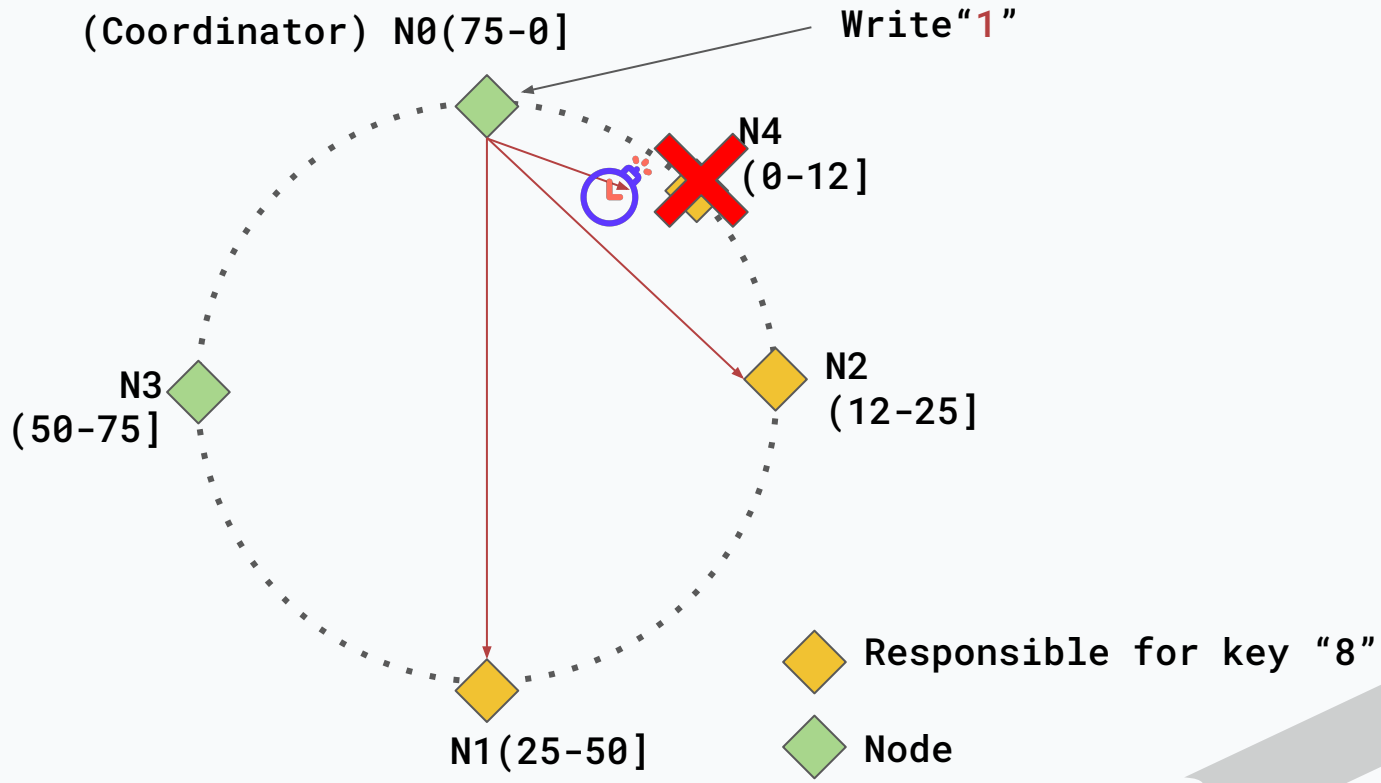
Fault Tolerance

6.1 Normal Writing

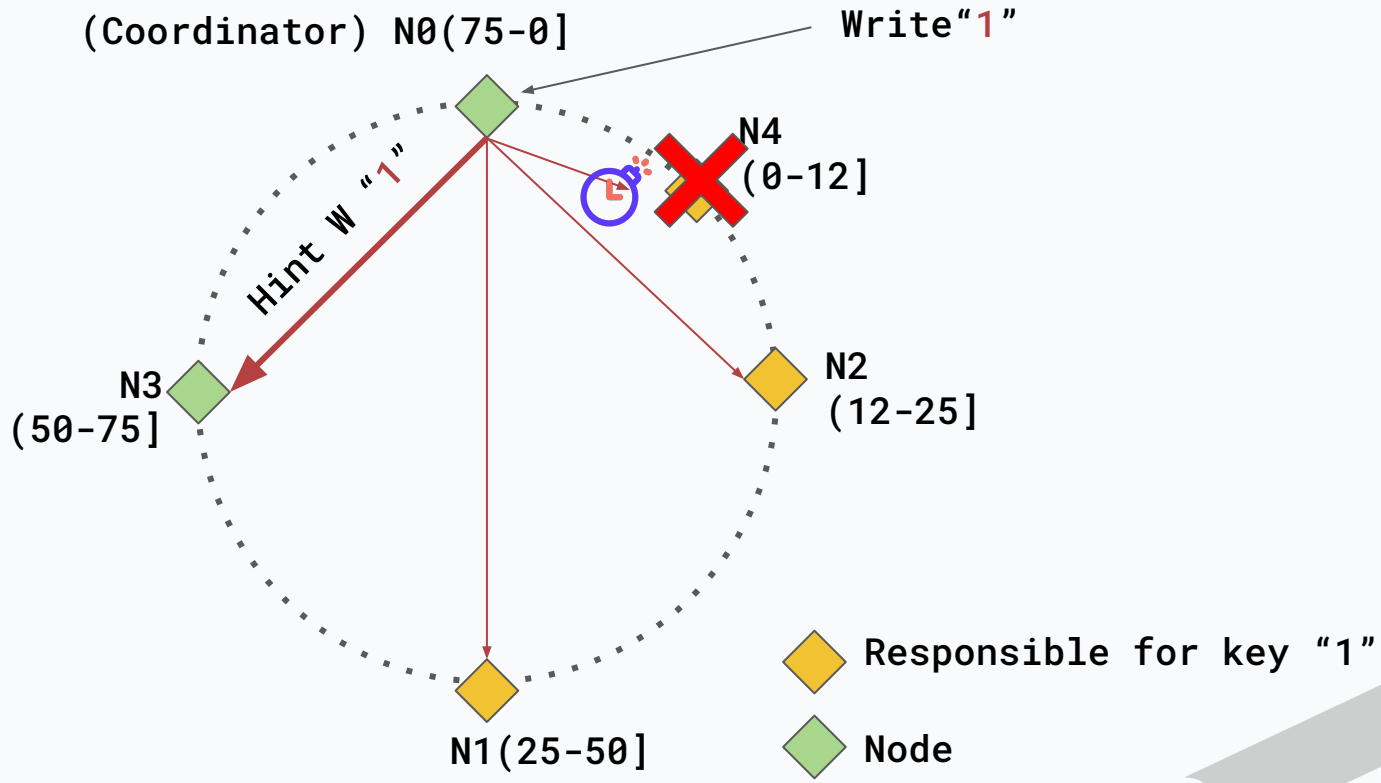



Demo Time:
Fault Tolerance,
Hinted Handoff

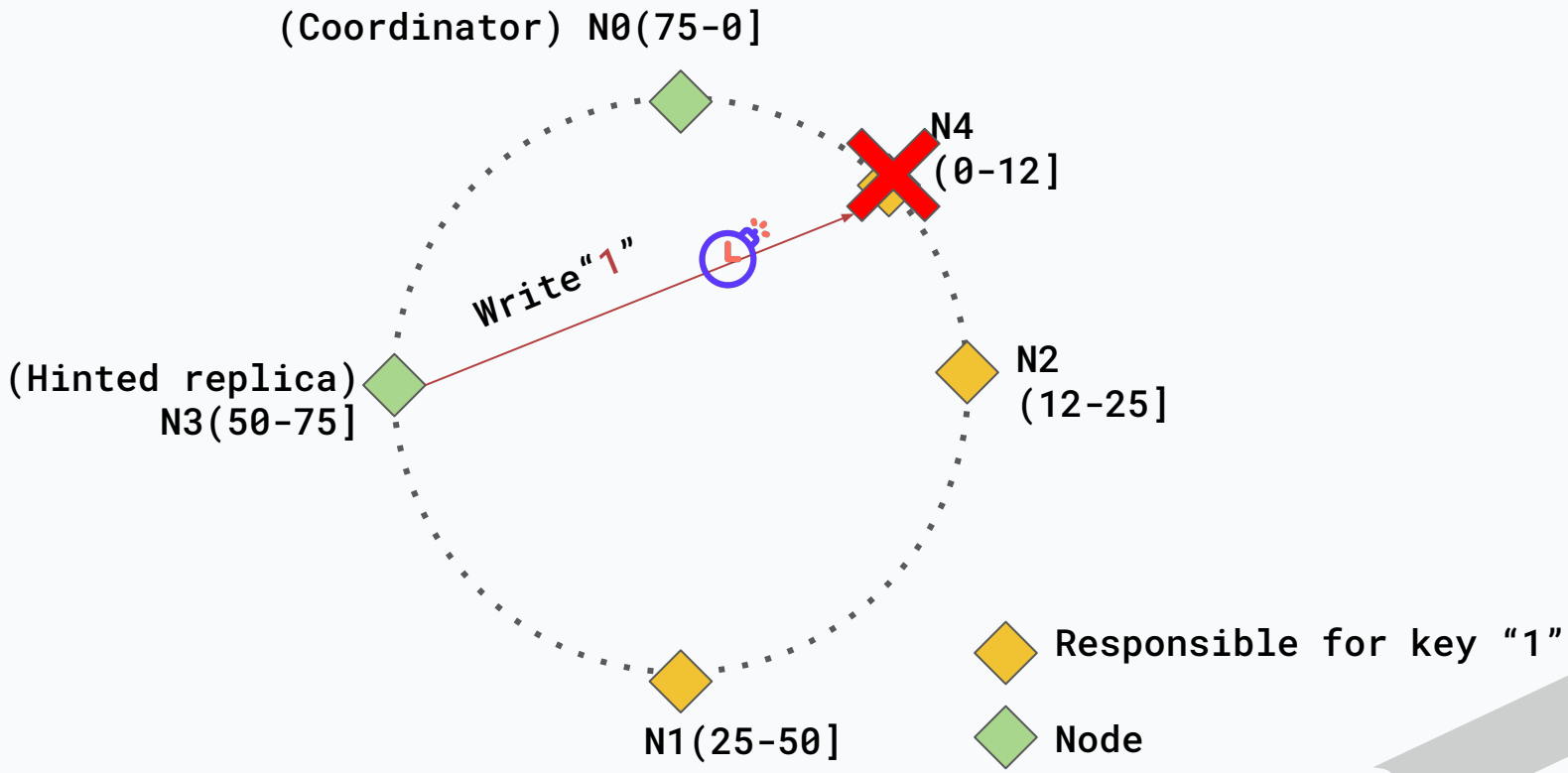
6.2 Single Node failure when Writing



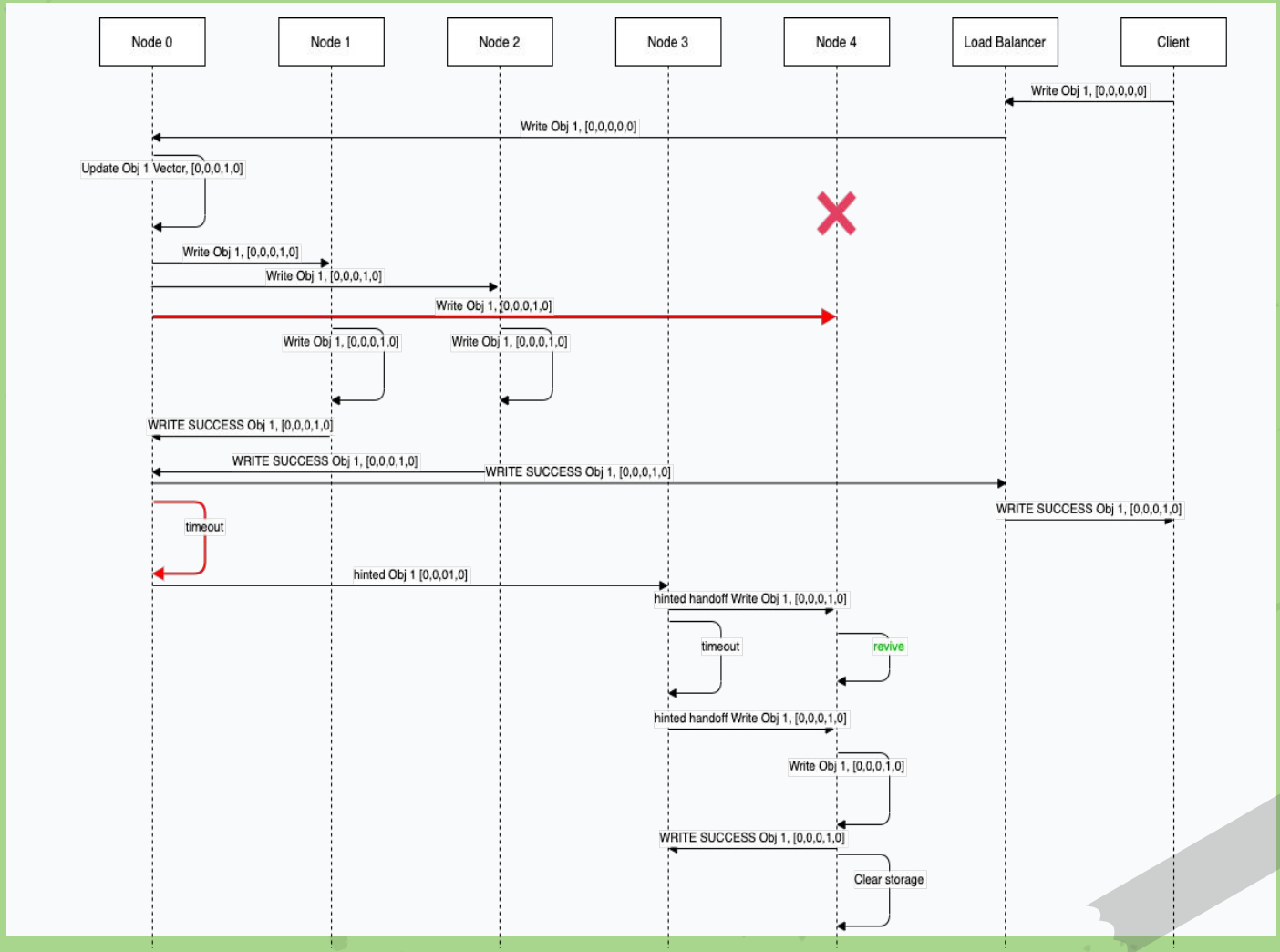
6.2 Single Node failure when Writing



6.2 Single Node failure when Writing

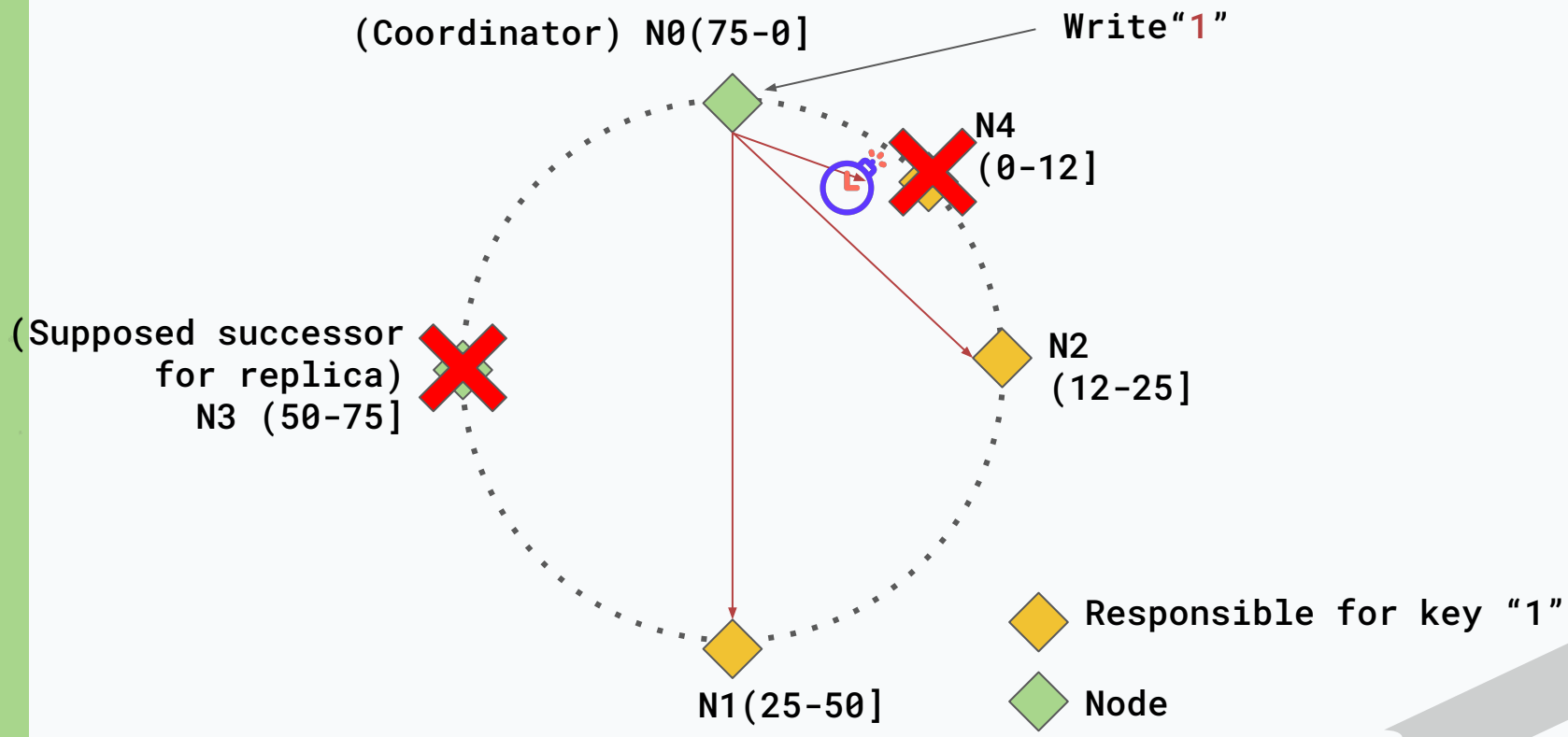


6.2 Single Node Failure when writing

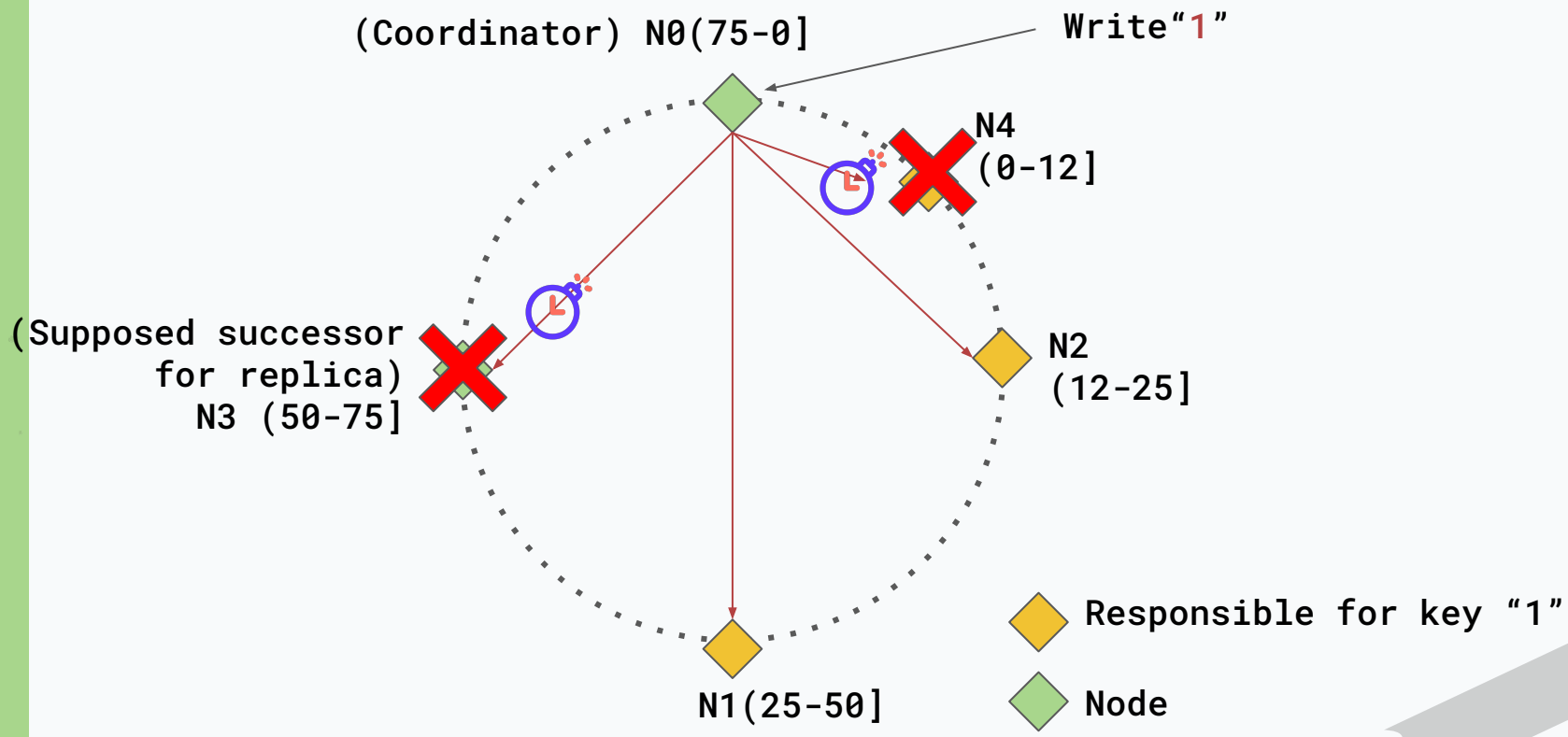



Demo Time:
Fault Tolerance,
Hinted Handoff

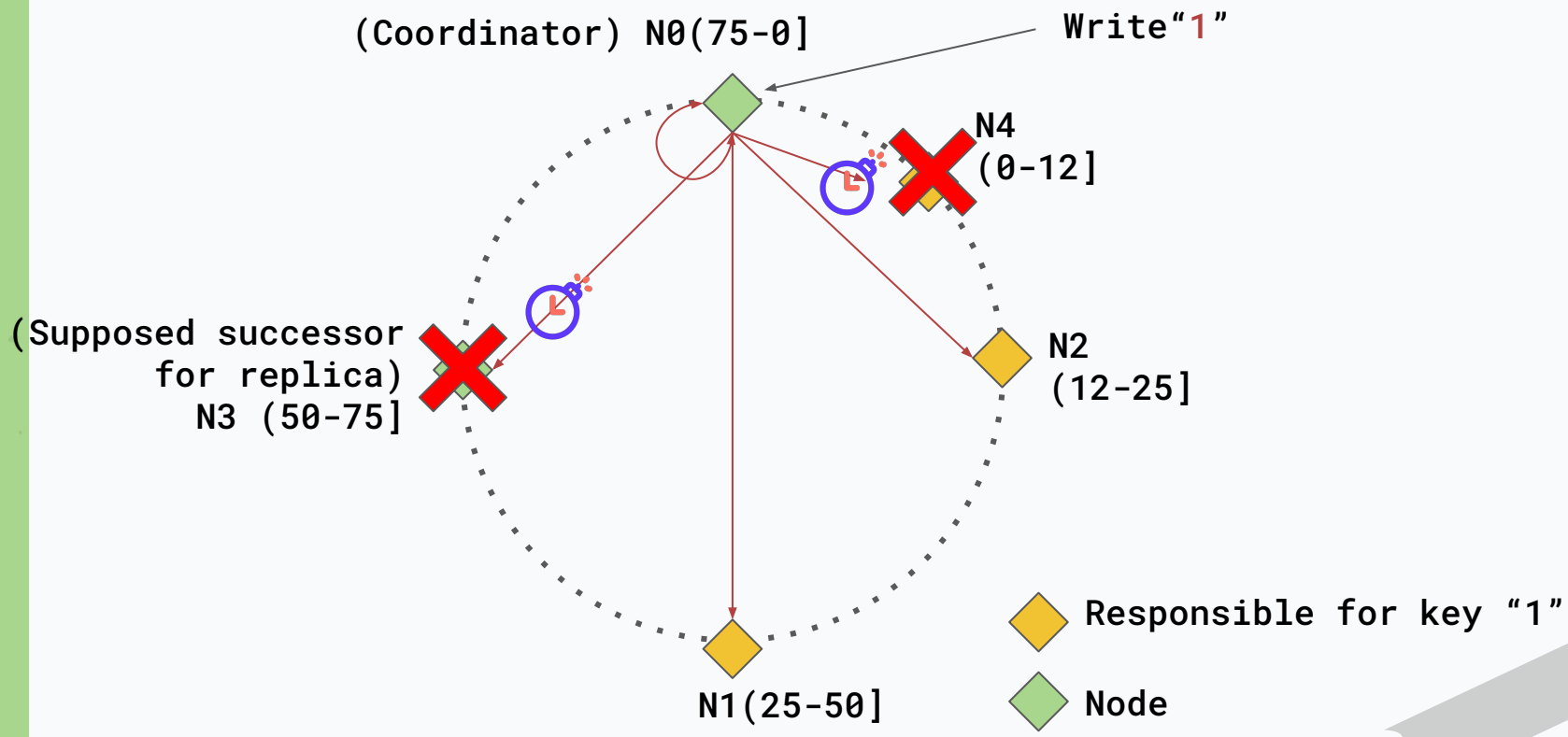
6.3.1 Multiple Node failures when Writing



6.3.1 Multiple Node failures when Writing

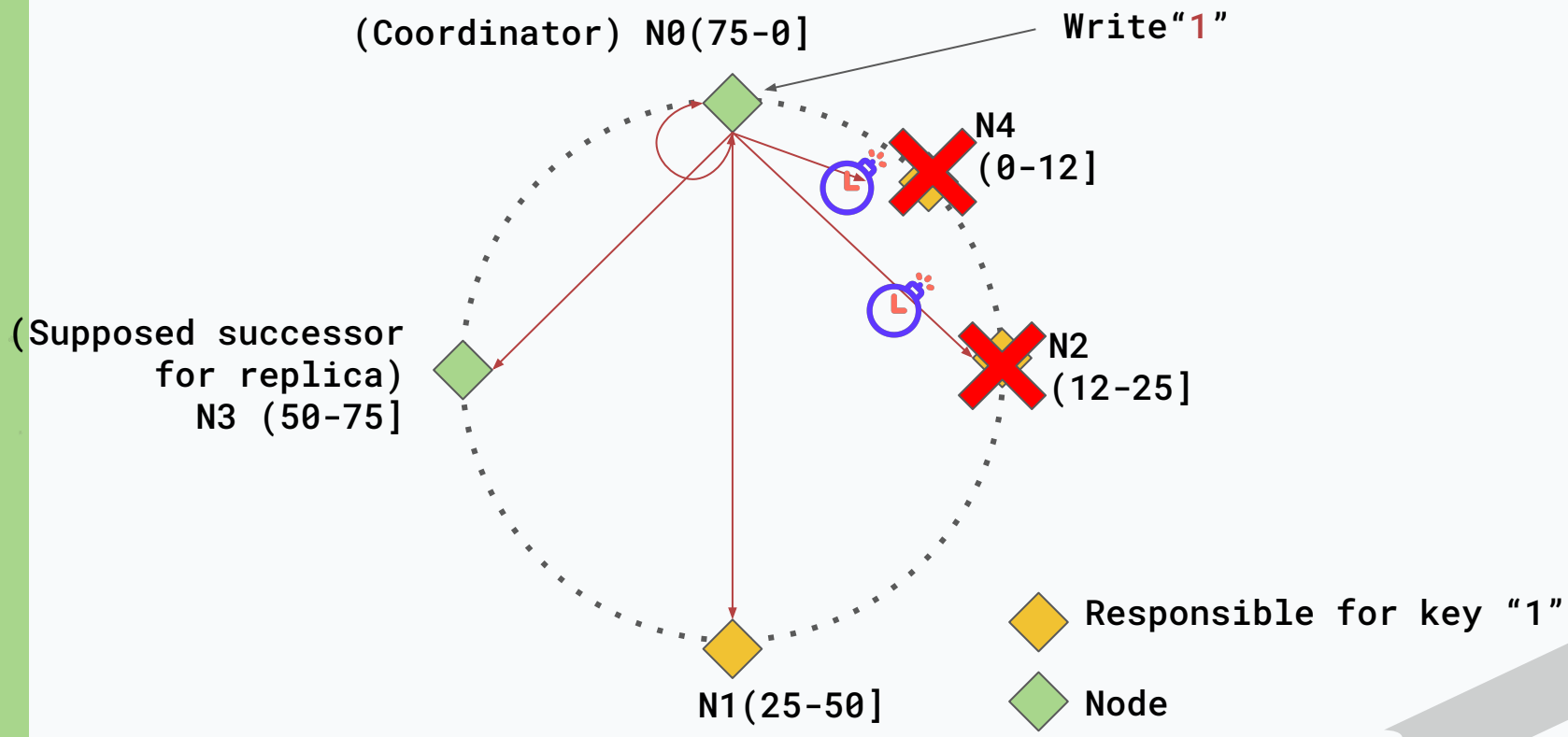


6.3.1 Multiple Node failures when Writing

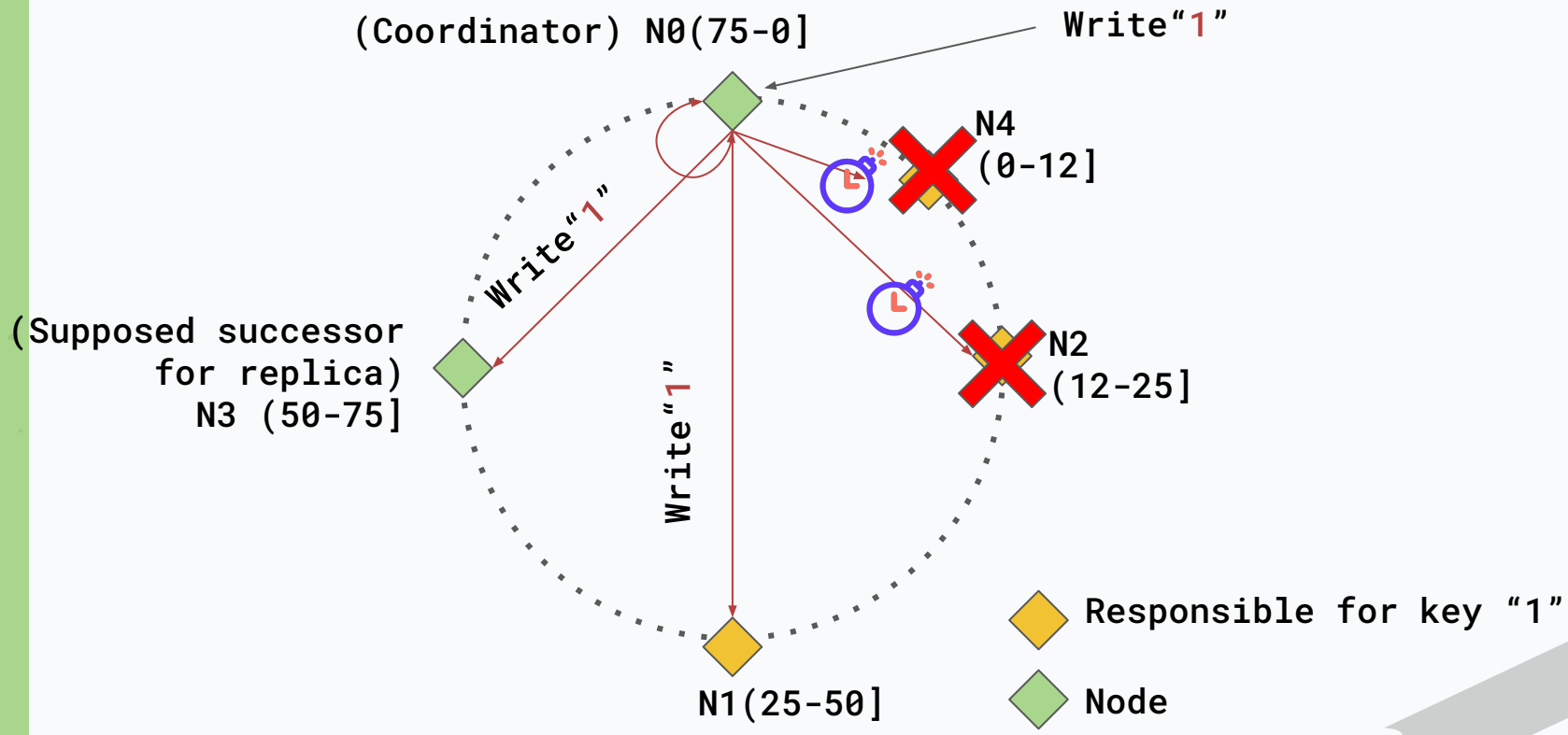



Demo Time:
Fault Tolerance,
Hinted Handoff

6.3.2 Multiple Node failures when Writing

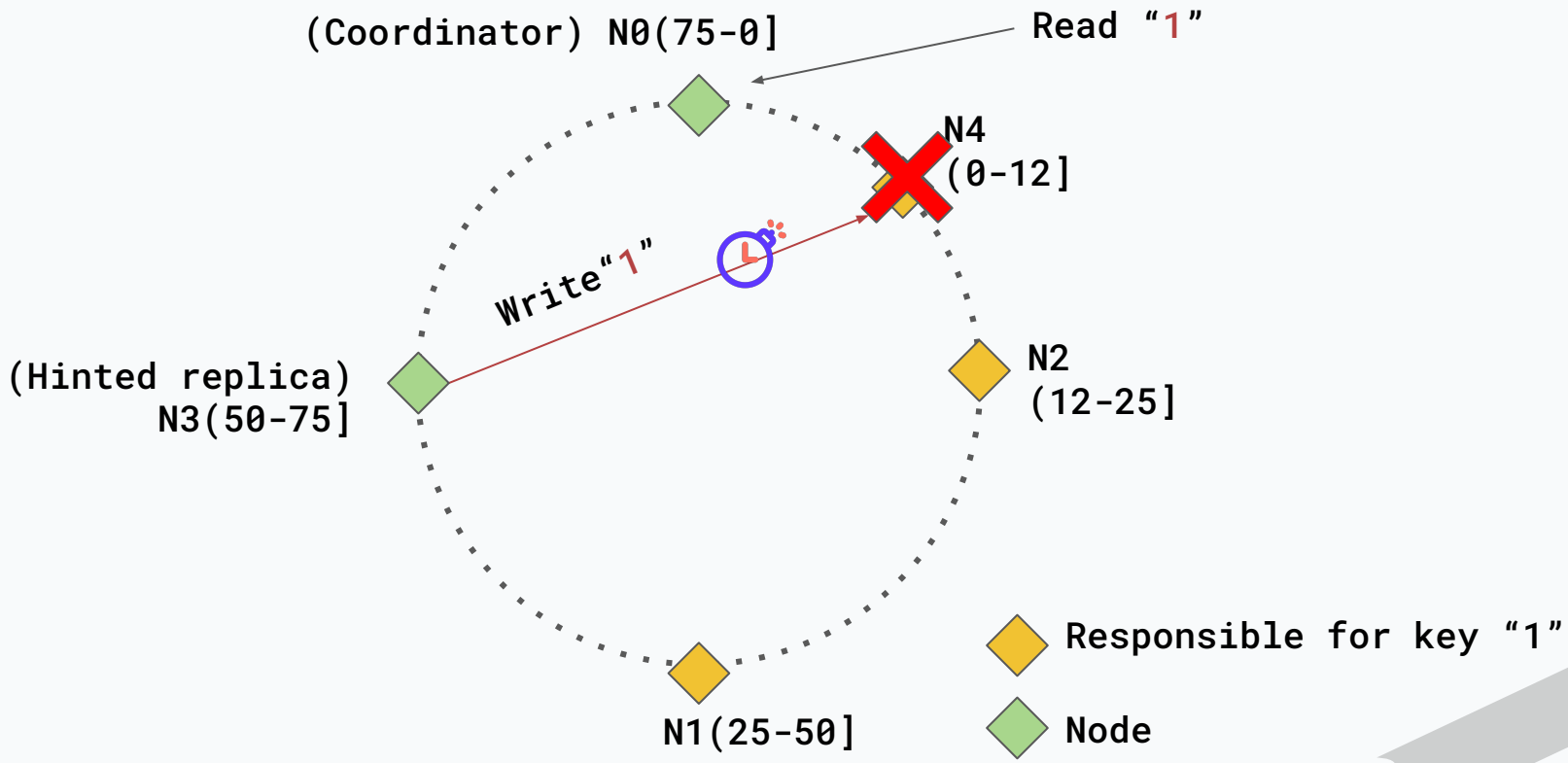


6.3.2 Multiple Node failures when Writing

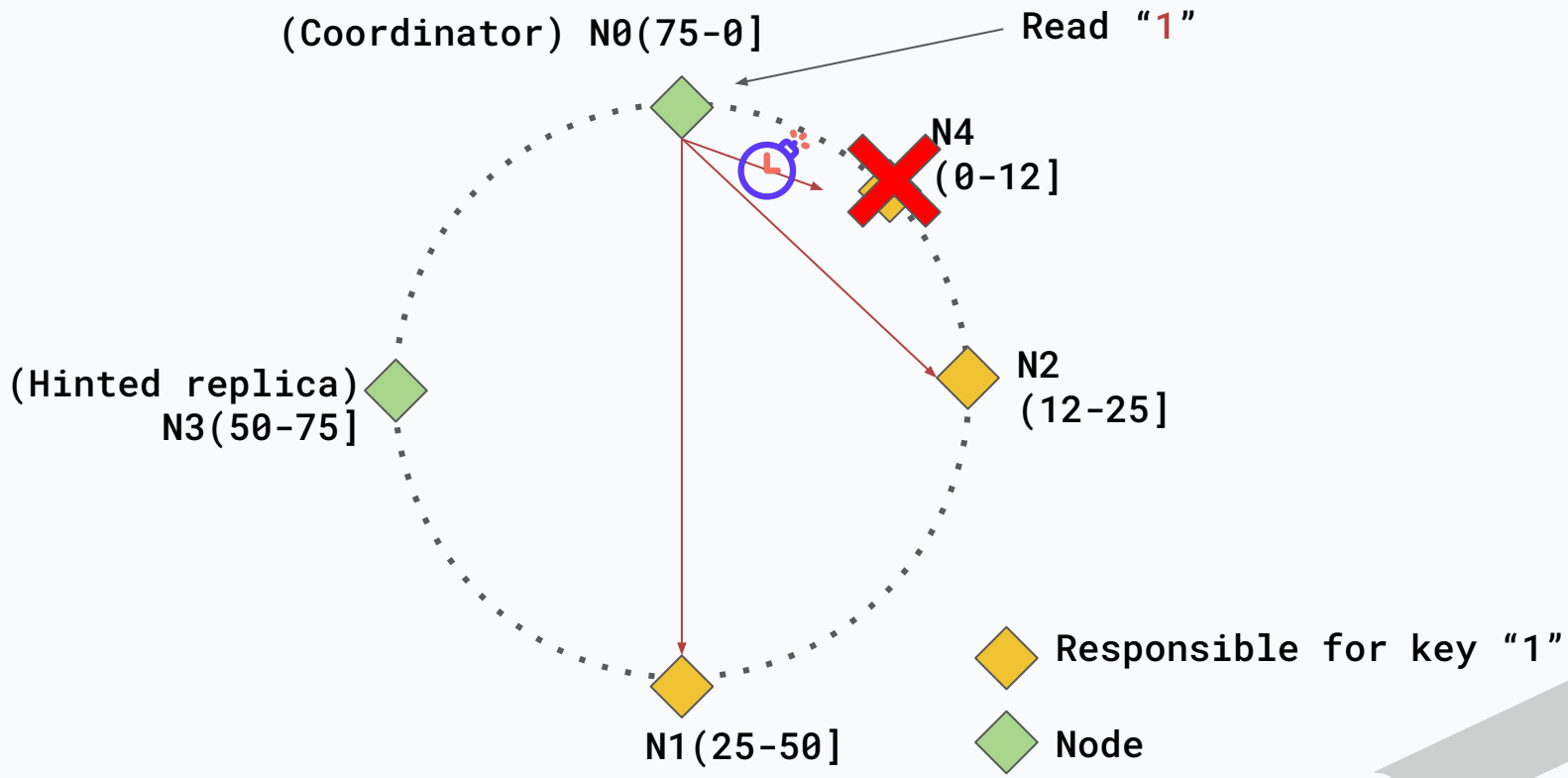



Demo Time:
Fault Tolerance,
Hinted Handoff

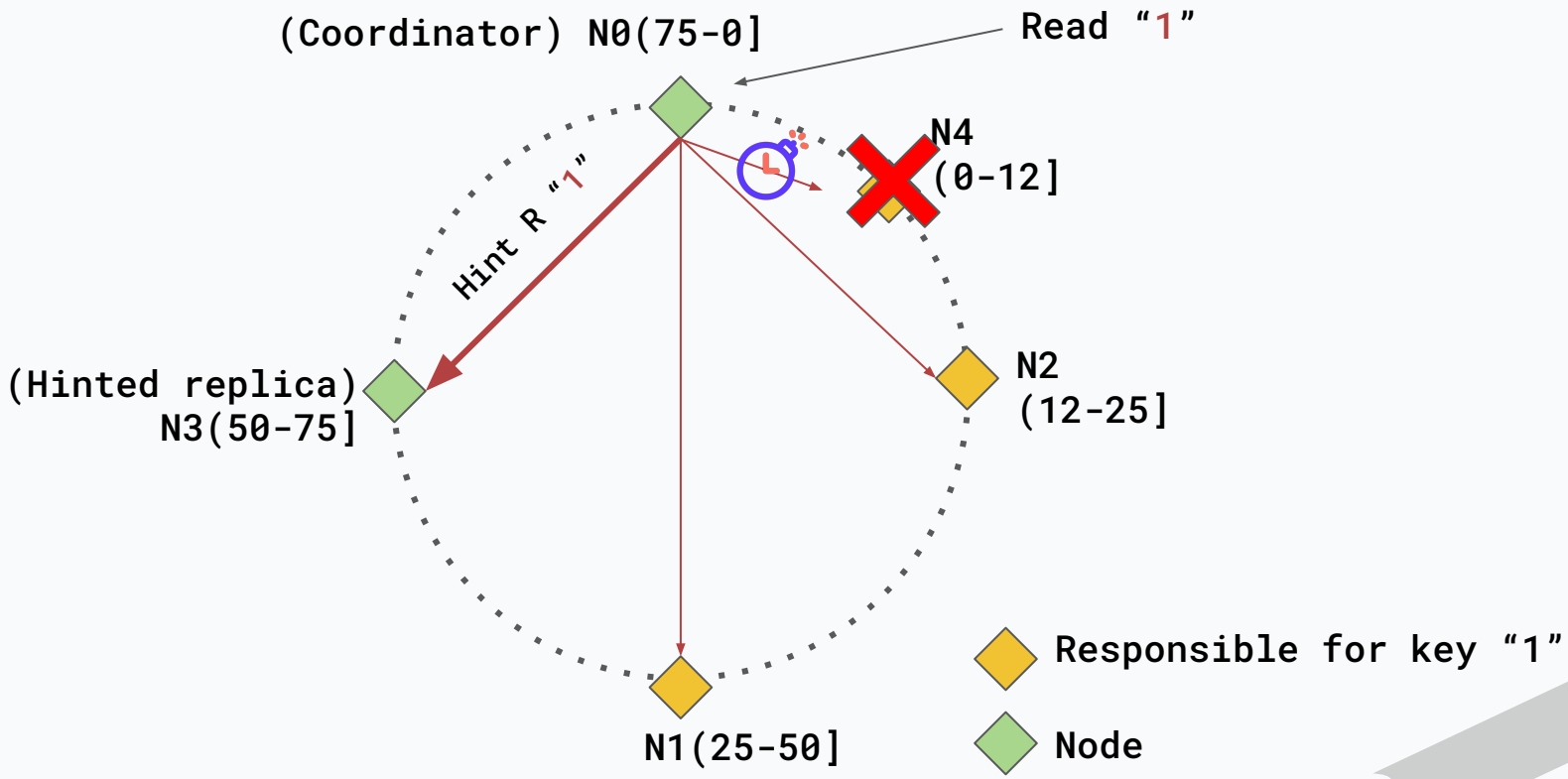
6.4 Reading During Node failure



6.4 Reading During Node failure

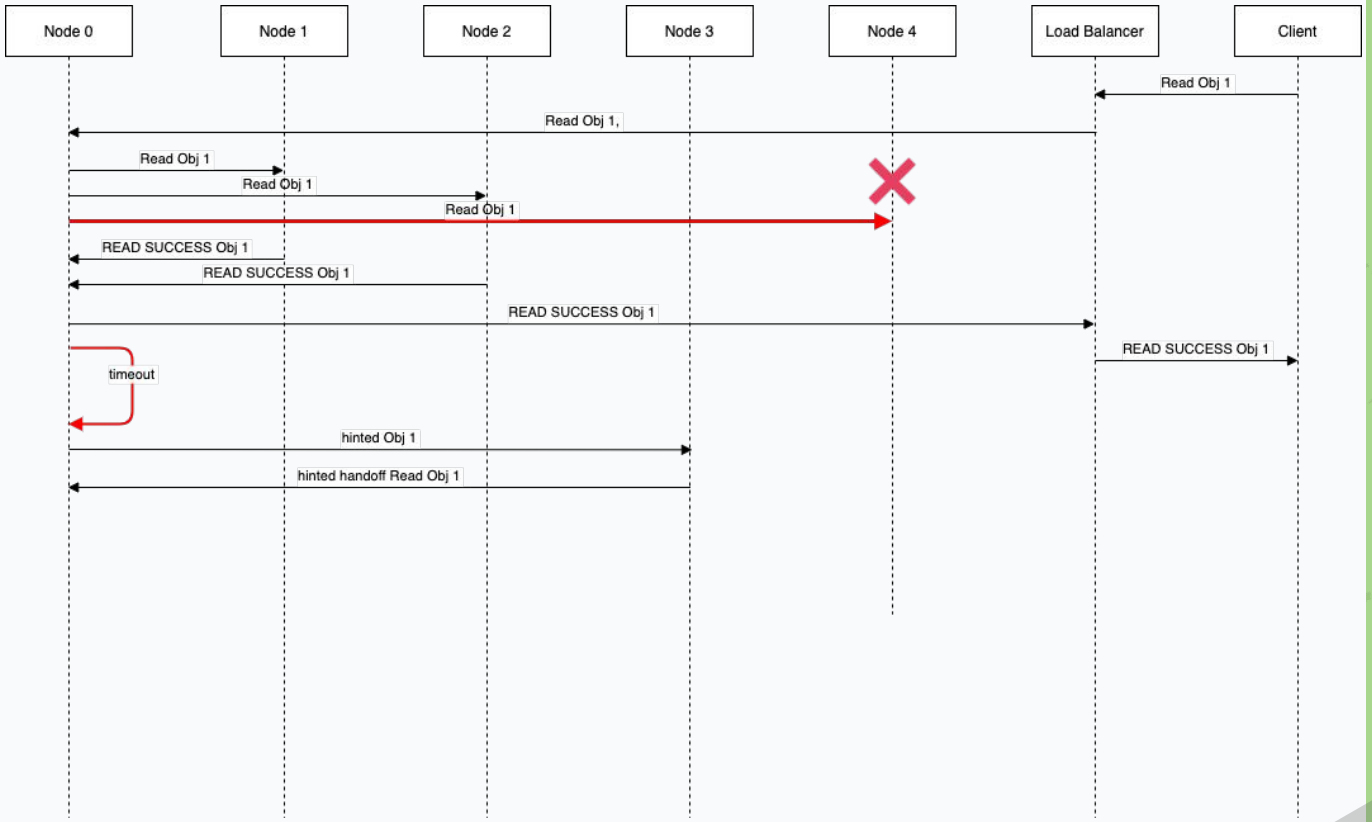


6.4 Reading During Node failure



6.4 Reading from nodes with hinted replicas

Coordinator




Demo Time:
Fault Tolerance,
Hinted Handoff

6.5 Maximum node failures

Given N nodes and F failures,
Minimum read success of R,
and minimum write success of W,

$$N - F >= \max(W,R)$$

Hence,
maximum node failure is F
where:

$$F <= N - \max(W,R)$$


Demo Time:
Fault Tolerance,
Hinted Handoff

07

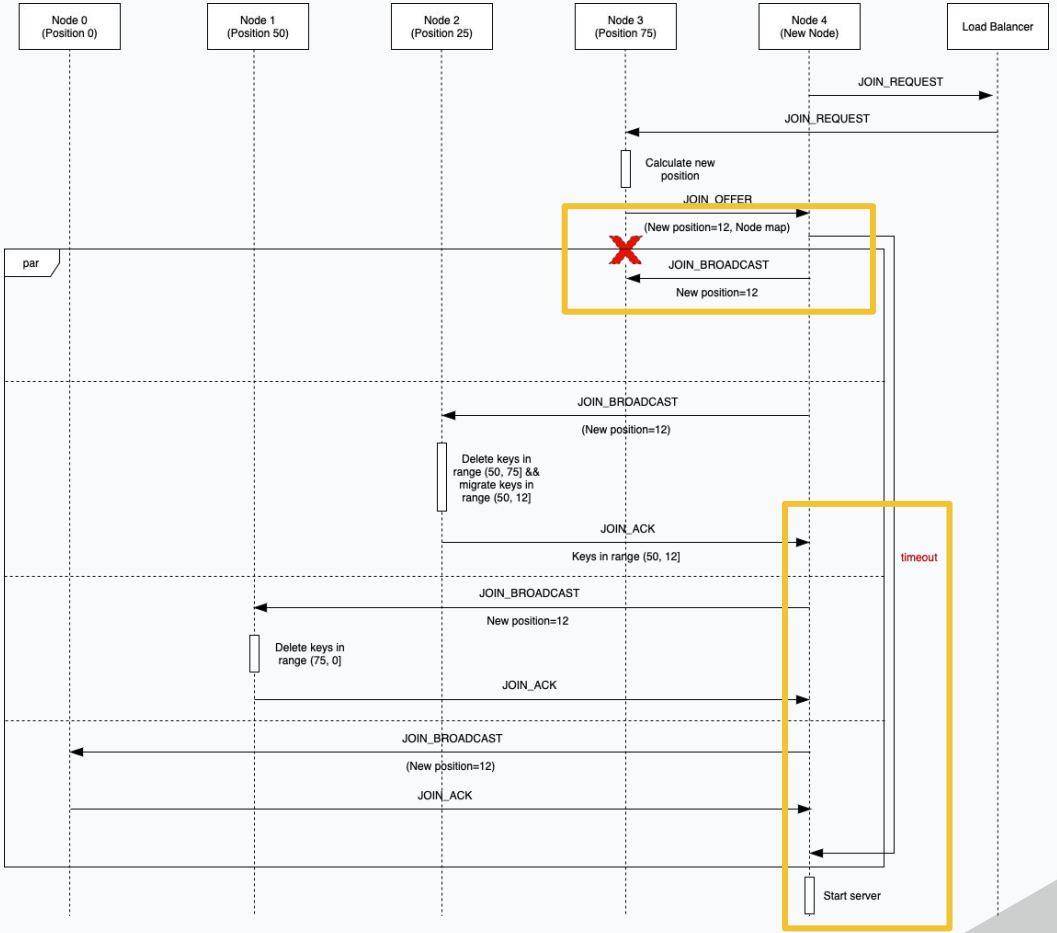
Further Developments

7 Considerations for further development

1. Implementation of leaving function for nodes
2. Fault tolerance during joining
3. Merkle Tree
4. Gossip based membership protocol and failure detection

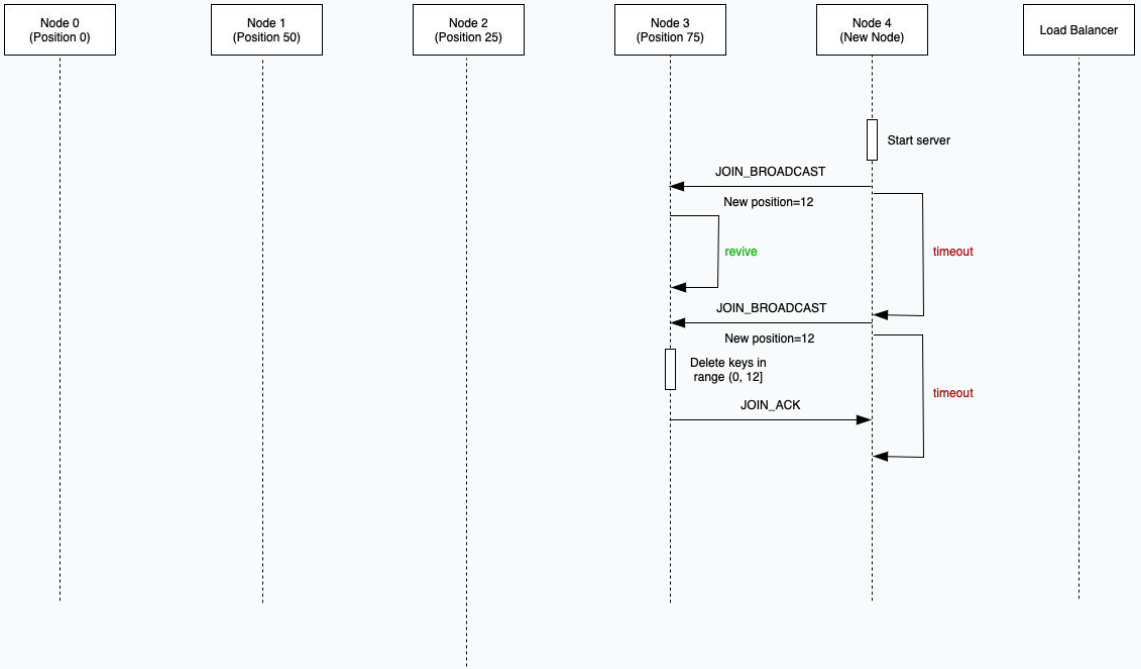
Protocol for Fault Tolerance during Joining

- Timeout is set to detect when nodes failed
- Node starts after timeout is reached



Protocol for Fault Tolerance during Joining

- New node continues to retry JOIN_BROADCAST until it receives a response



CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.

Please keep this slide for attribution.