CS 739 Final Project

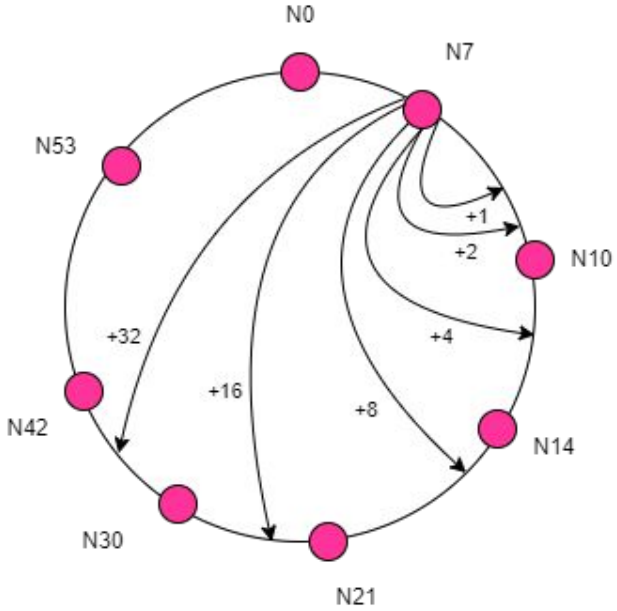
P2P DHT Storage using Chord

Madhav Kanbur
Lincoln Spartacus James
Himanshu Pandotra
Pavithran Ravichandiran

Introduction

- GOAL: a peer-to-peer decentralized storage system.
- Chord protocol for peer lookup in log(N).
- Peers are distributed over a ring.
- Efficient load balancing via consistent hashing.

  Node hash: Ip_addr + port; **data hash** = Client's public key + file name
- Replication of data over the peers for fault tolerance.
- Flat namespace w.r.t each client
- Multi User support via Cryptographic authentication for security



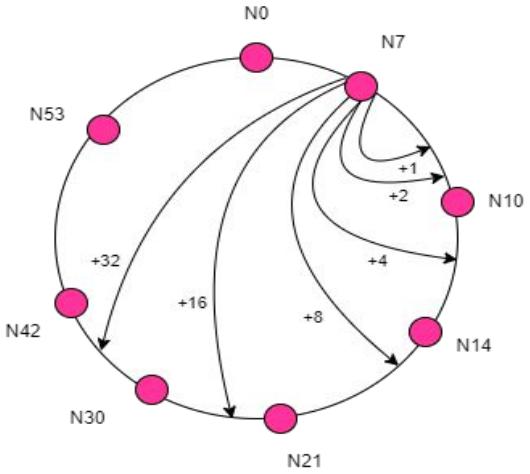
Chord Ring

6-bit ID space (8 peers)

Chord Design

Finger Table for Node 7

Entry (i)	Id+2^i	Succ
0	N7+1	N10
1	N7+2	N10
2	N7+4	N14
3	N7+8	N21
4	N7+16	N30
5	N7+32	N42



Chord Ring

- Each node has a **finger table** that acts a dynamic routing table for node lookups across the chord ring.
- For **m bit** chord ring, we have **2^m chord identifiers**. **m=6** in above figure.
- The algorithm proves that having **m entries** in each finger table performs node lookups in O(log N) time.

Server API

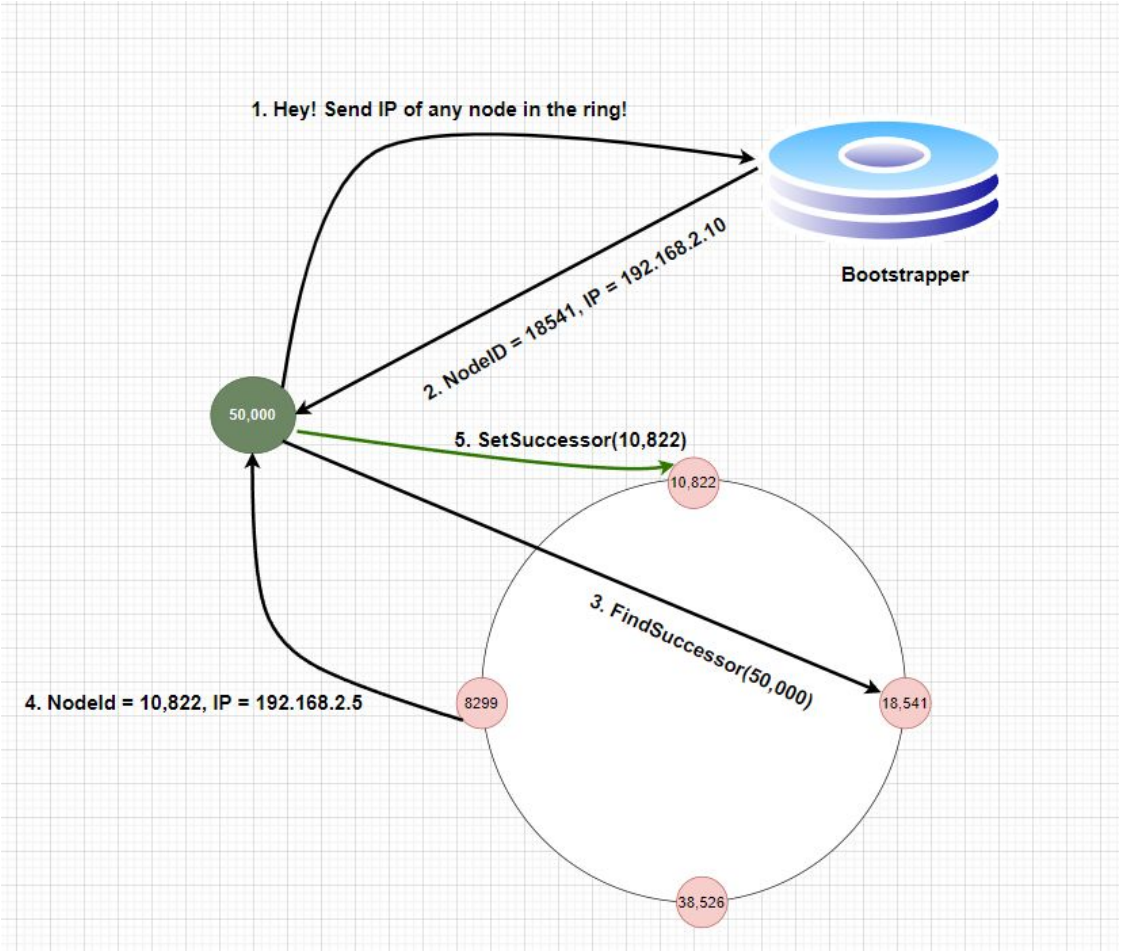
Client Exposed API

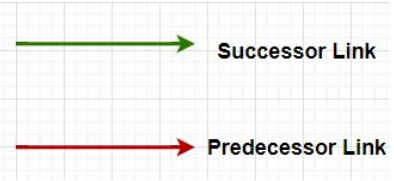
- **getFile**(f) : Returns file F to client
- **putFile**(f) : Stores file F on the coordinator node

Internal Chord API

- **join**(n) : Join existing ring via node n
- **findSuccessor**(k) : Find the coordinator for key k
- **notify**(n') : Updates predecessor to n' (if conditions are met)
- **replicateFile**(f) : Replicate file f on this node
- **getSuccessorList**() : Returns successorList
- **stabilize()** : Verifies if successor is accurate, calls notify() on successor
- **syncSuccessorList()** : Maintains the successorList
- **fixFingerTable()** : Maintains the finger table

Chord Join()



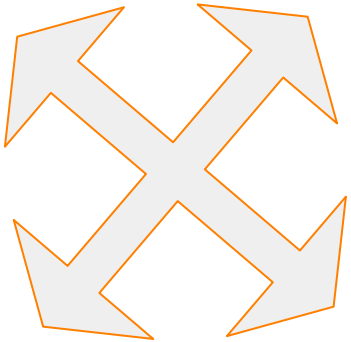


- New Chord nodes contacts the bootstrapper during startup to find the entry point.
- Similarly, Clients use the bootstrapper during initial startup to find the Chord ring entry point. Subsequent Read/Put Operations go directly to the Peer node without contacting the bootstrapper

Chord Pointer Maintenance

```
// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);
```

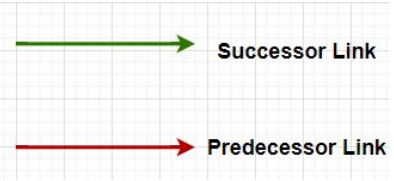
```
// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';
```



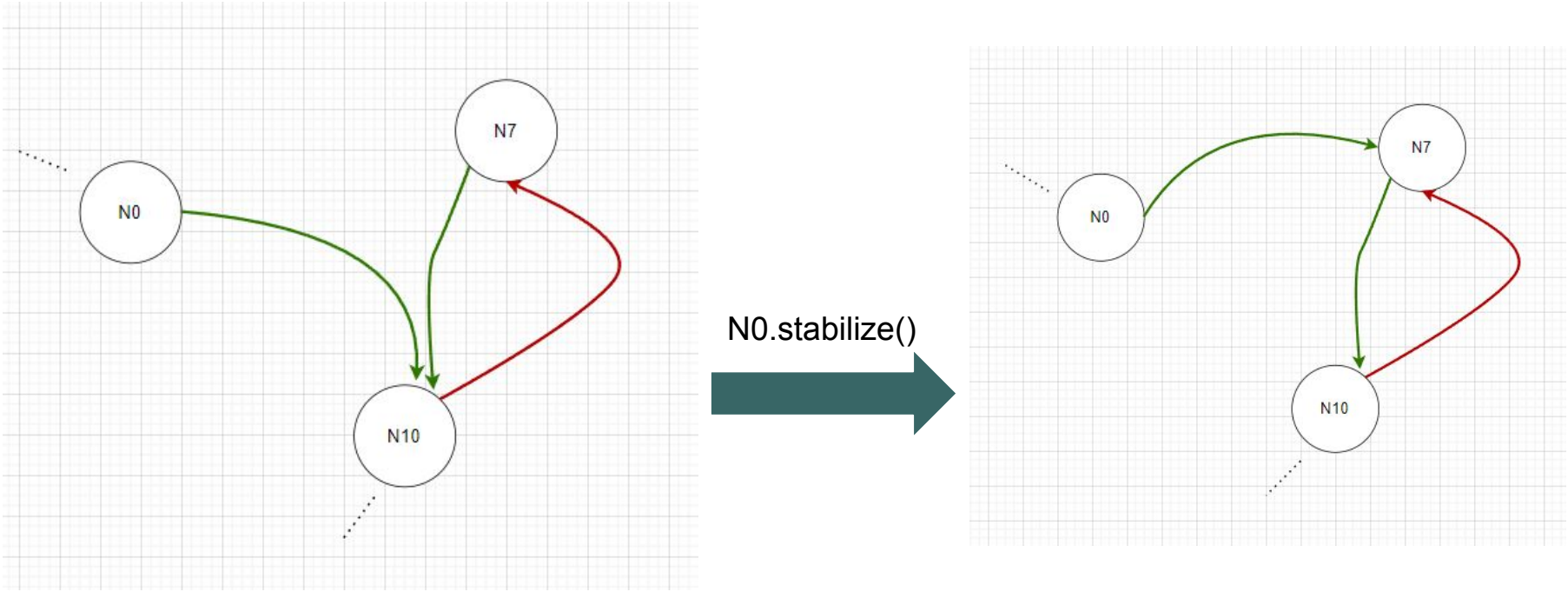
```
// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find successor(n + 2^(next-1));
```

```
// Fetch successor list from our successor and
// make it our own by deleting last element
// and prepending the successor
n.sync_successor_list():
        new_list = successor.get_SuccessorList()
        Delete last element from new_list
        New.prepend(successor)
        Self.SuccessorList = new_list
```

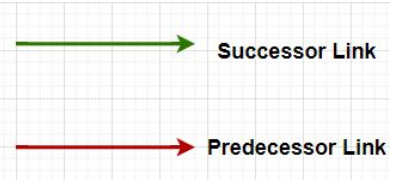
Chord Stabilize()



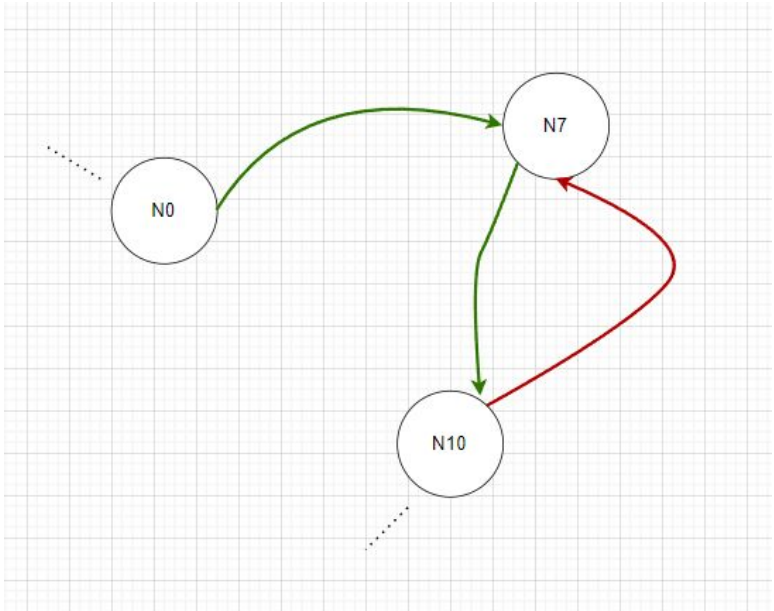
- Runs periodically to verify N's successor and keep it up to date.



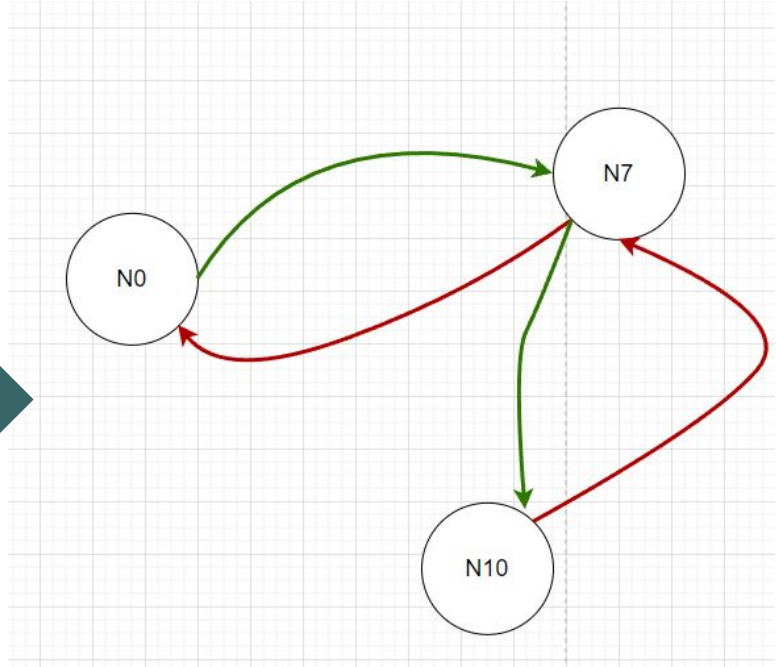
Chord Notify(N')



- Informs successor of our presence to update succ's predecessor.
  Used by stabilize()

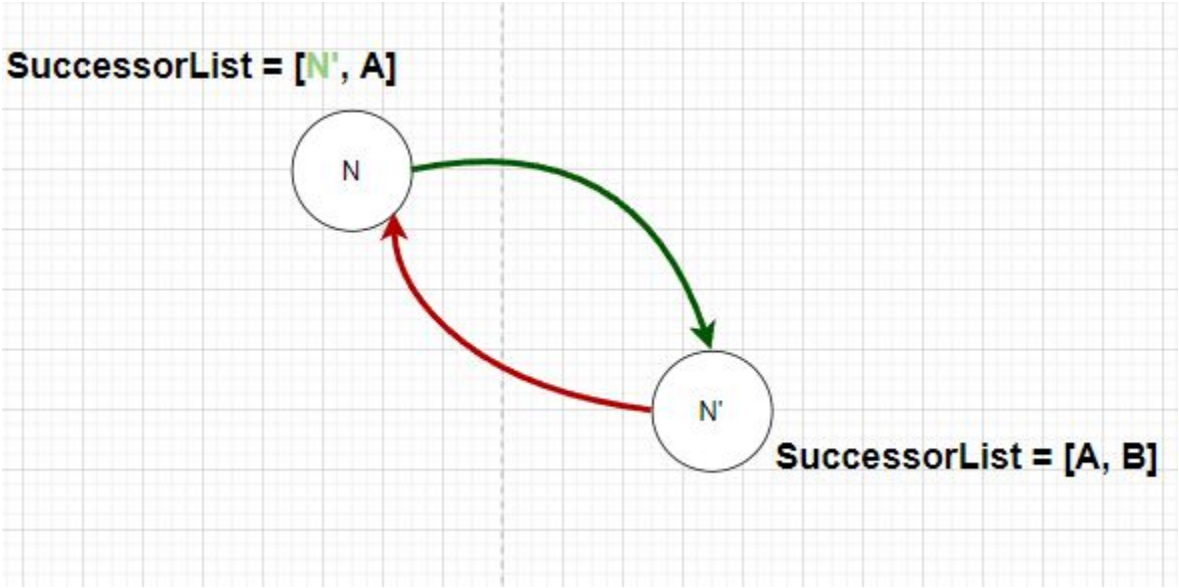


N7.notify(N0)

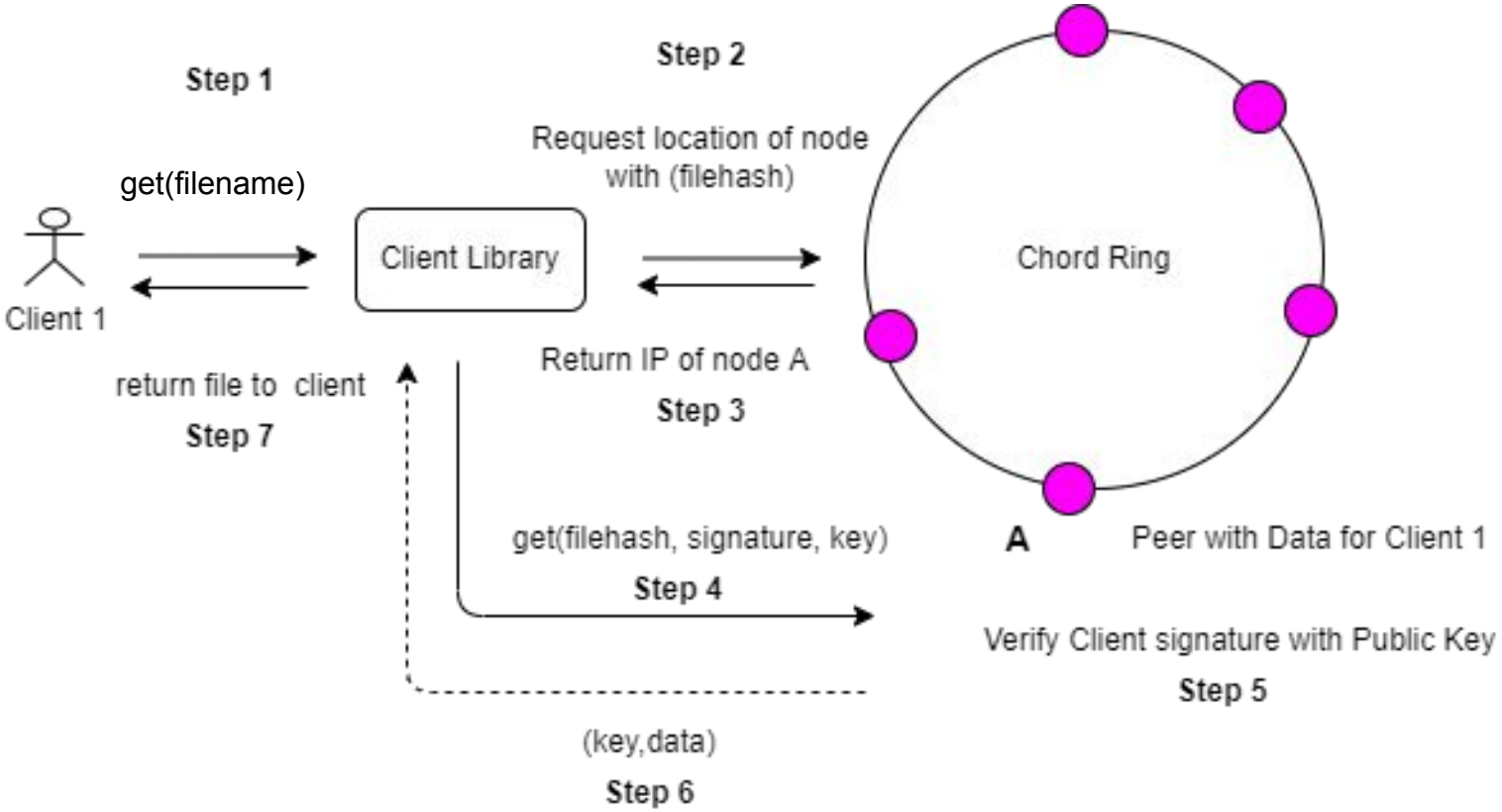


Chord SyncSuccessorList()

- Build node N's successorList (SL) by fetching successor's SL



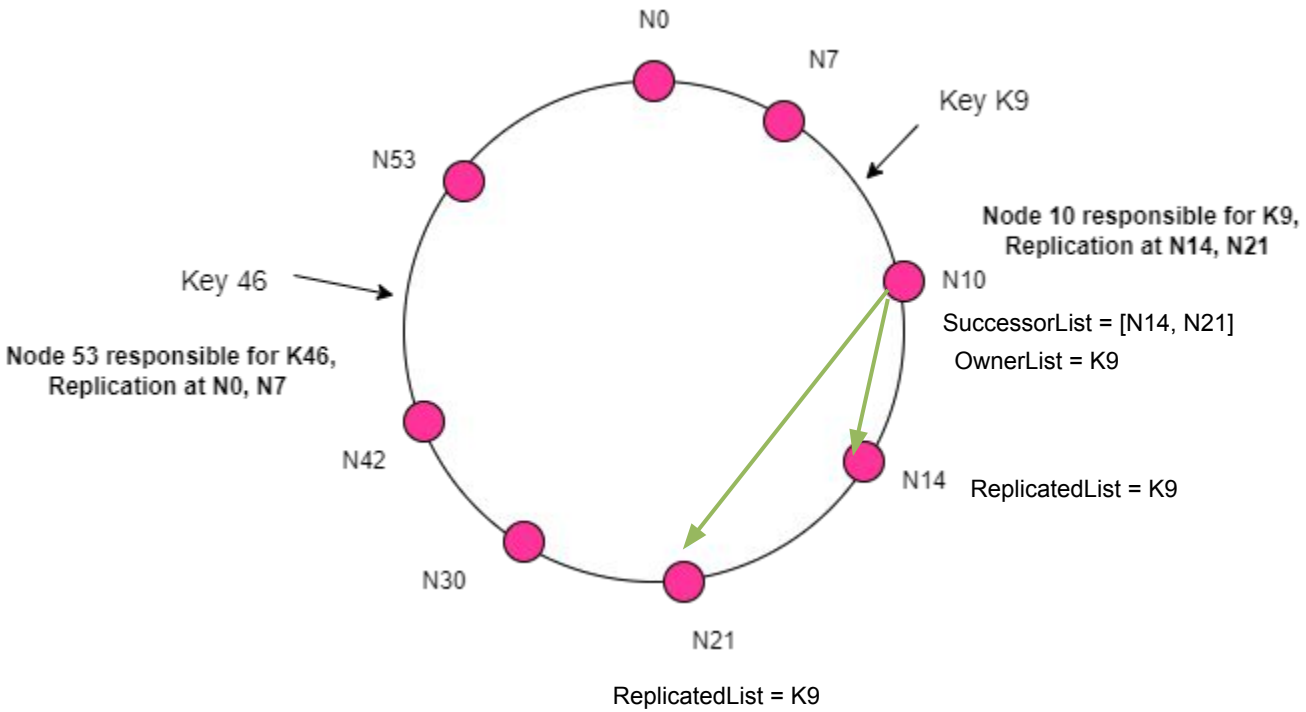
Overview of Get()



Key Replication

- Node responsible for key ‘k’ is the **coordinator**
- Each node maintains a SuccessorList of the next N successors
- Coordinator replicates keys on the entire SuccessorList
- Default size = 2 (replication factor is 3)
- To distinguish between “owner” keys & “replicated” keys, we maintain 2 lists (helpful during recovery)
- Replication is **asynchronous** (done in background)

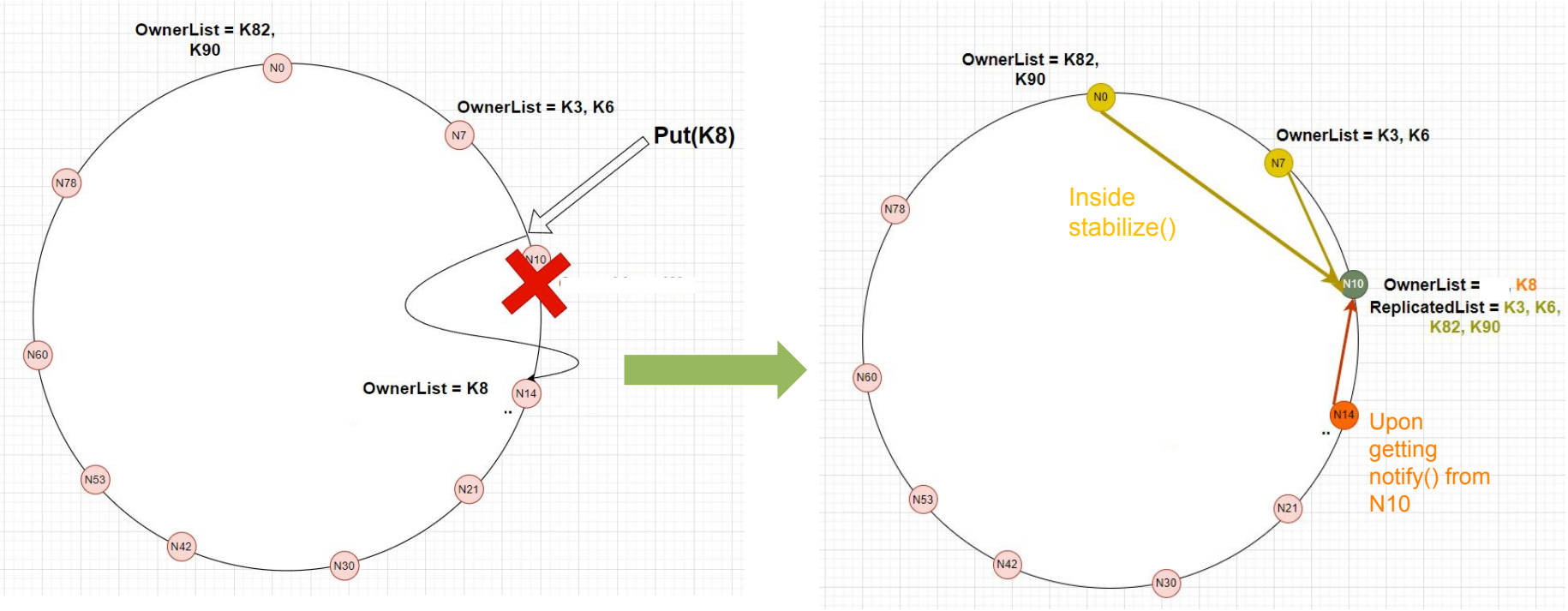
Key Replication (N = 2)



Node Recovery

- When node ‘n’ crashes, its immediate successor becomes the coordinator.
- On recovery, all N predecessors of new node ‘n’ replicate their **predecessor[i].ownerList => n.replicatedList**
- n’s successor replicates **subset** of its ownerList to n i.e. **succ.ownerList => n.ownerList**

Node Recovery



Multi User Support via Crypto Auth

Client Side: Every get() & put() has these -

a. Message (filename)

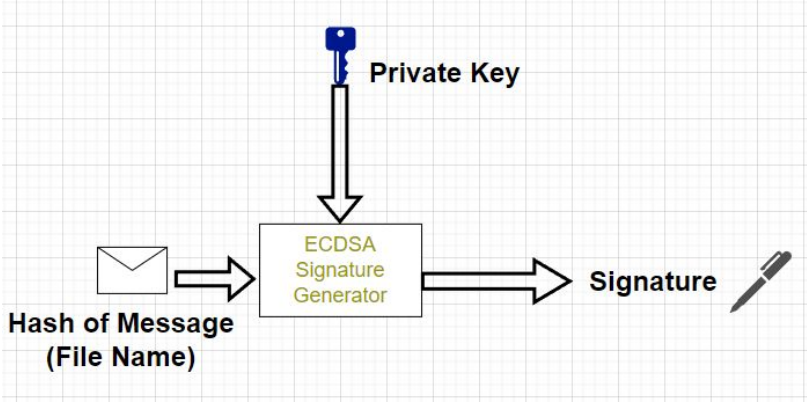
b. Public Key

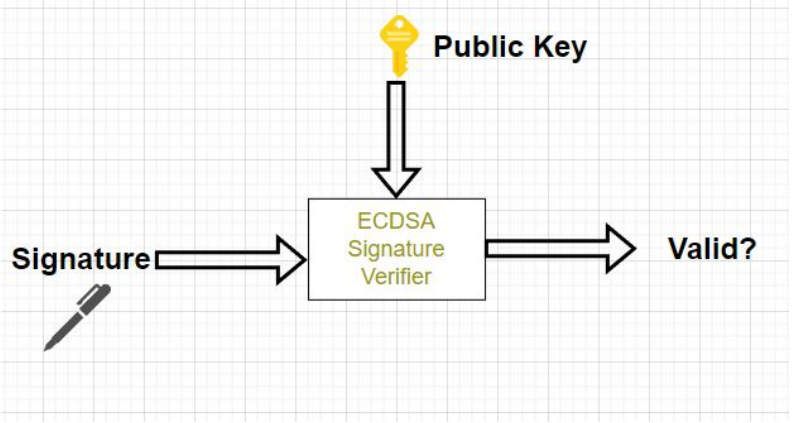
c. Signature

signature → sign_message(message, private_key)

Server Side:

Valid/Invalid → verify(message, public_key, signature)





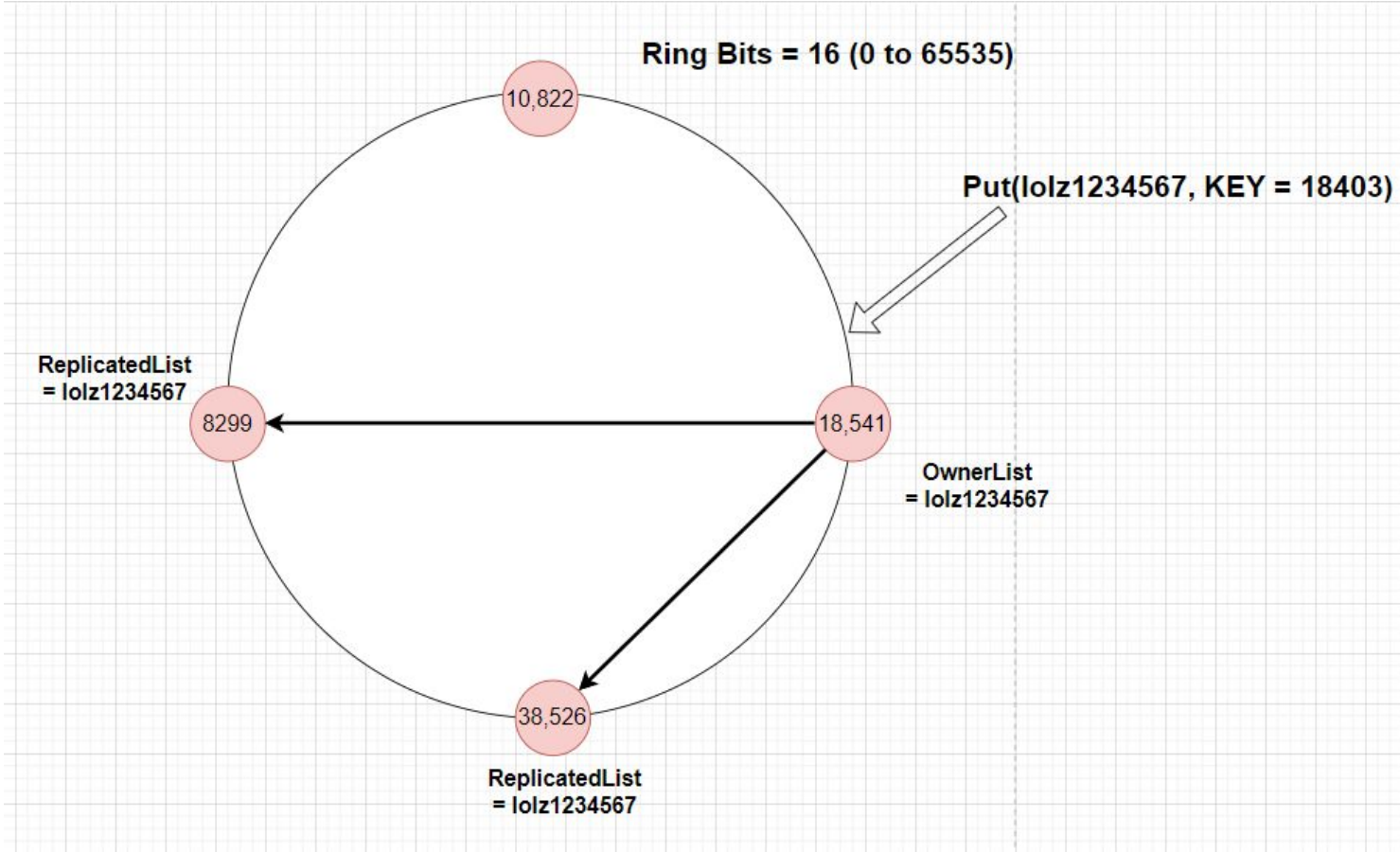
Consistency

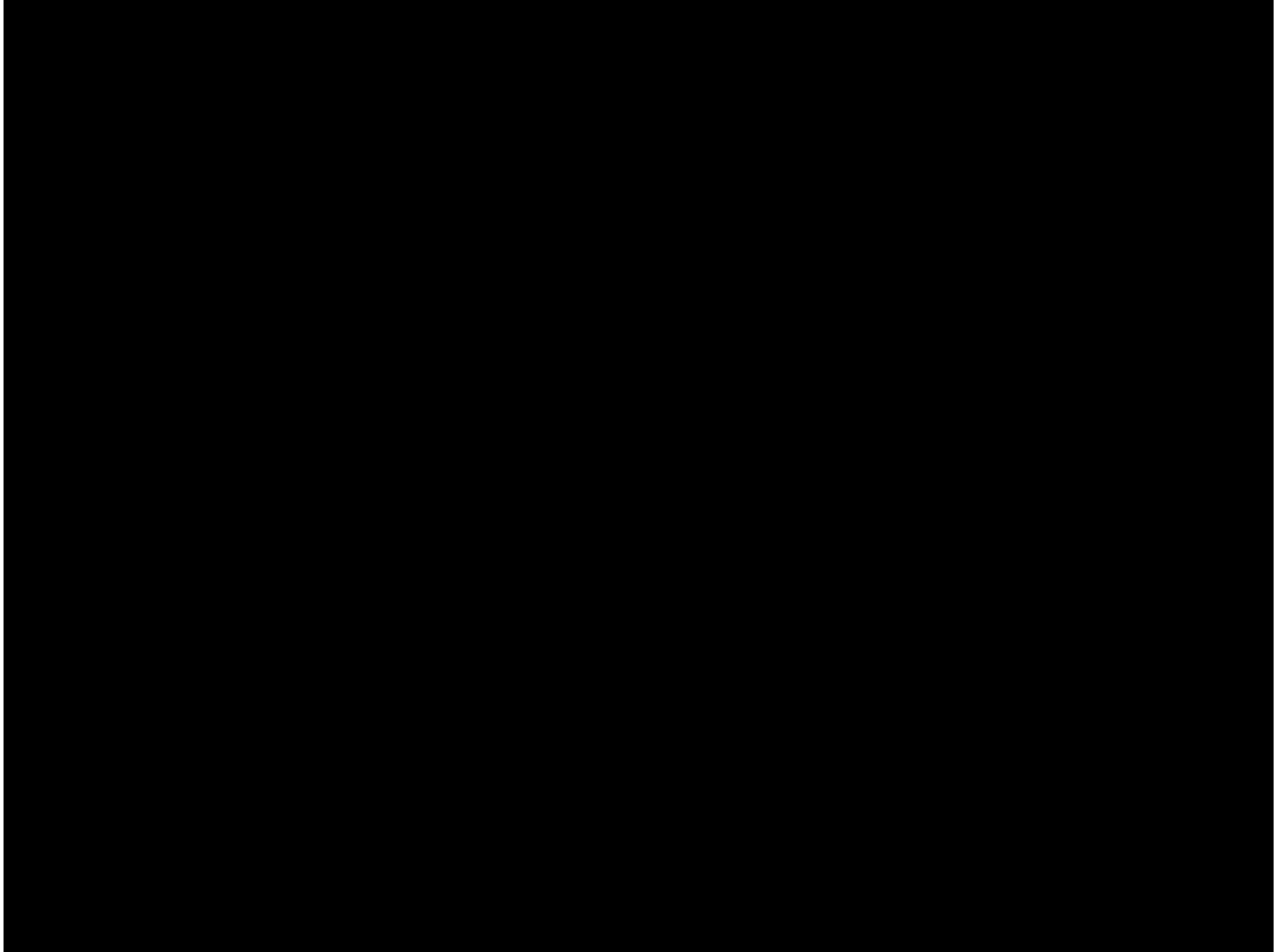
- Due to the asynchronous nature of replication, we provide **Eventual Consistency** guarantees**.**

- If a node crashes before fully replicating a file, the new coordinator will return an old copy (if it exists else KeyNotFoundError).

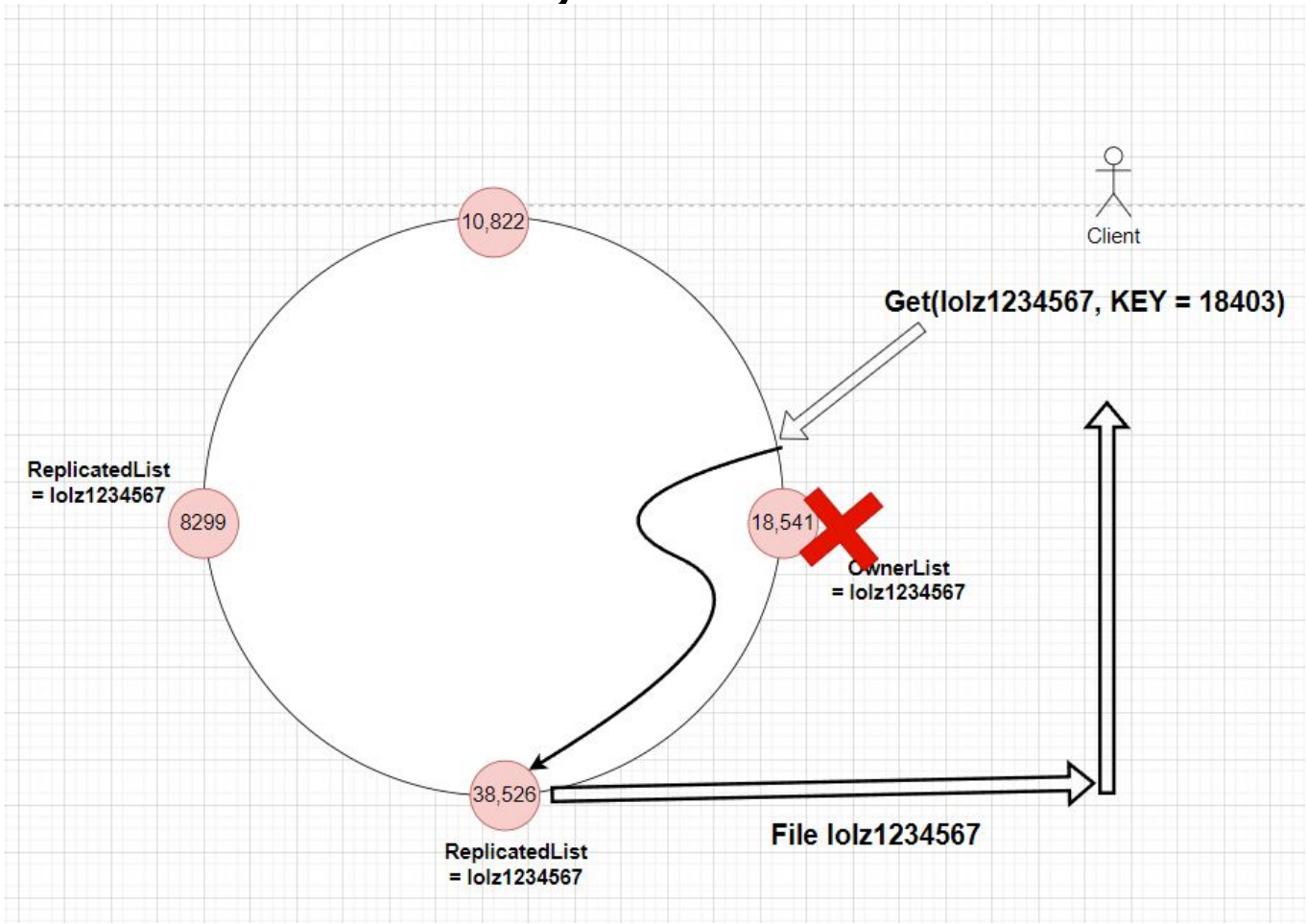
- What about versioning? - Ultimate truth is whatever current coordinator has

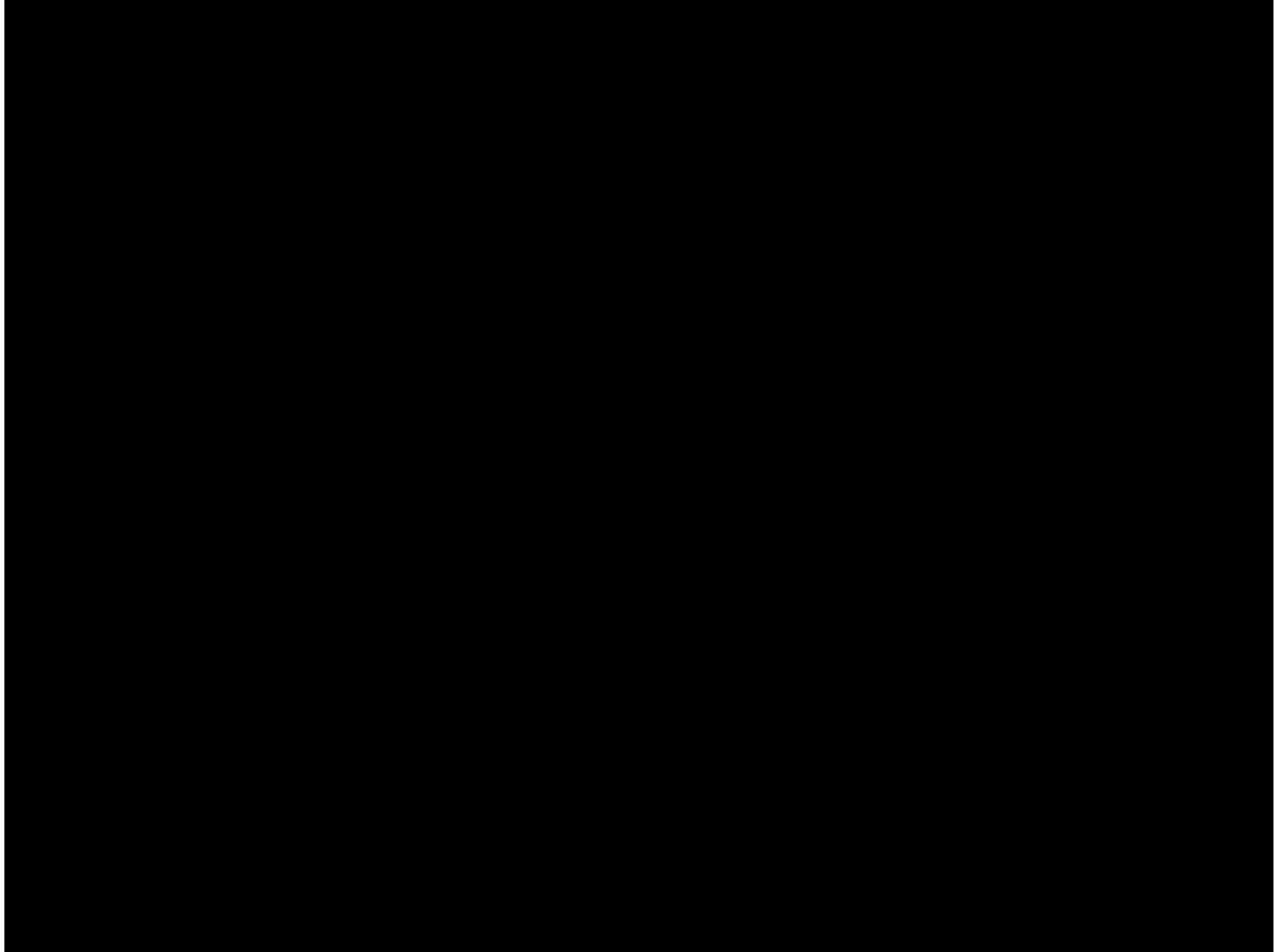
Demo #1 : Basics



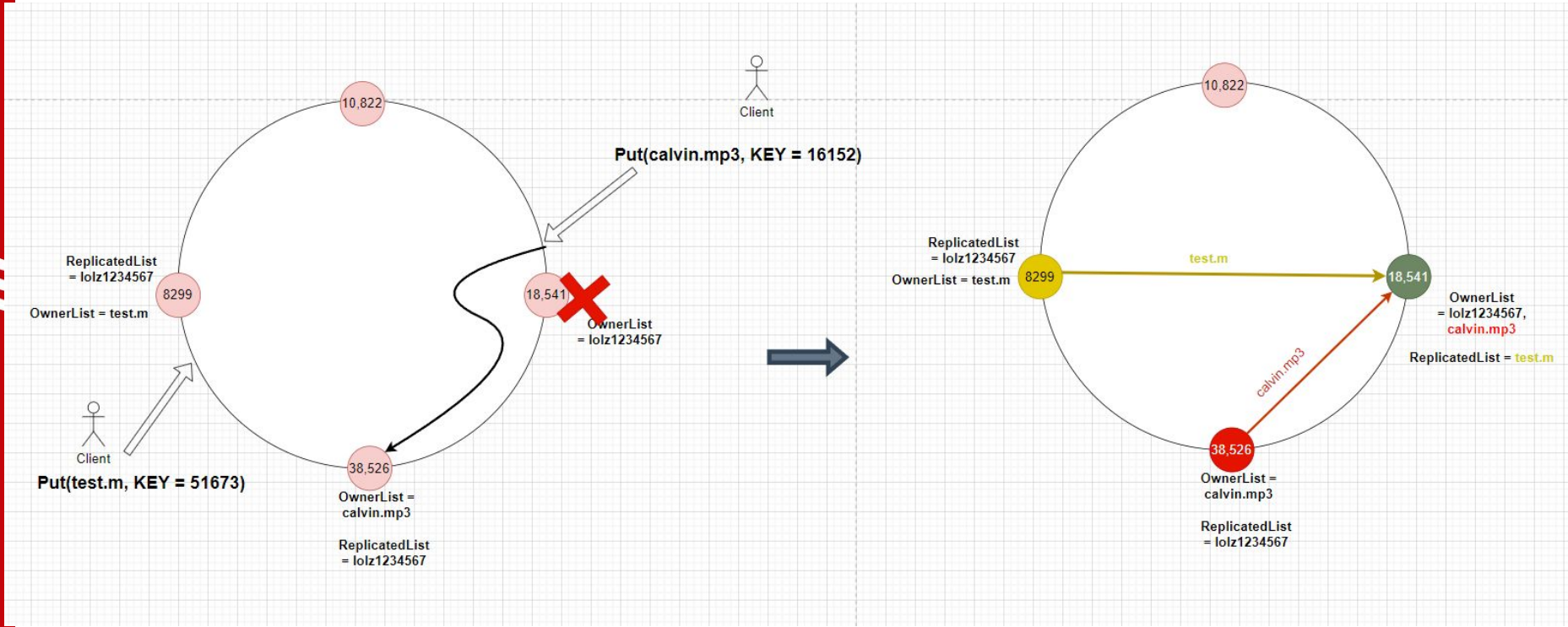


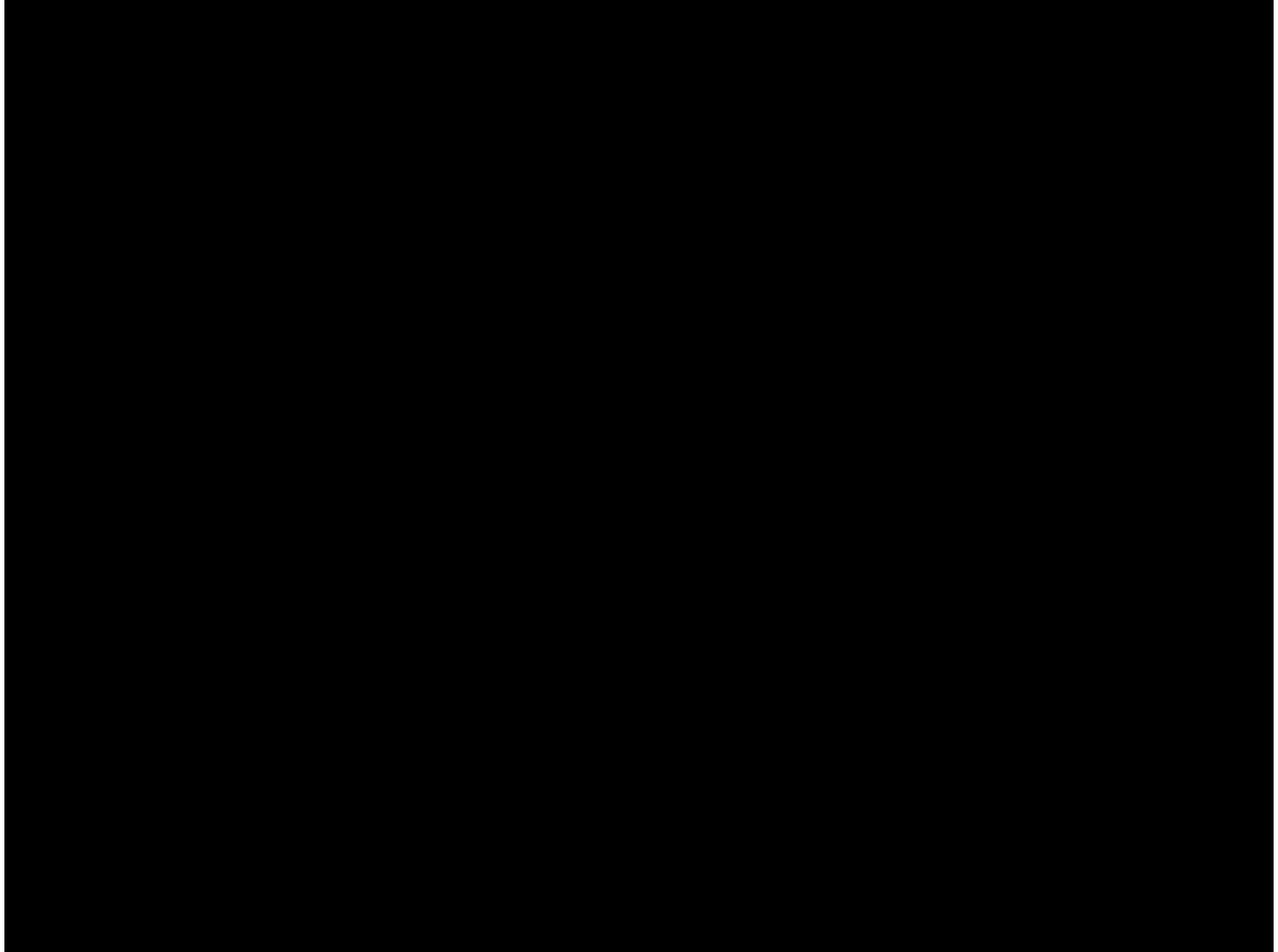
Demo #2 : Availability





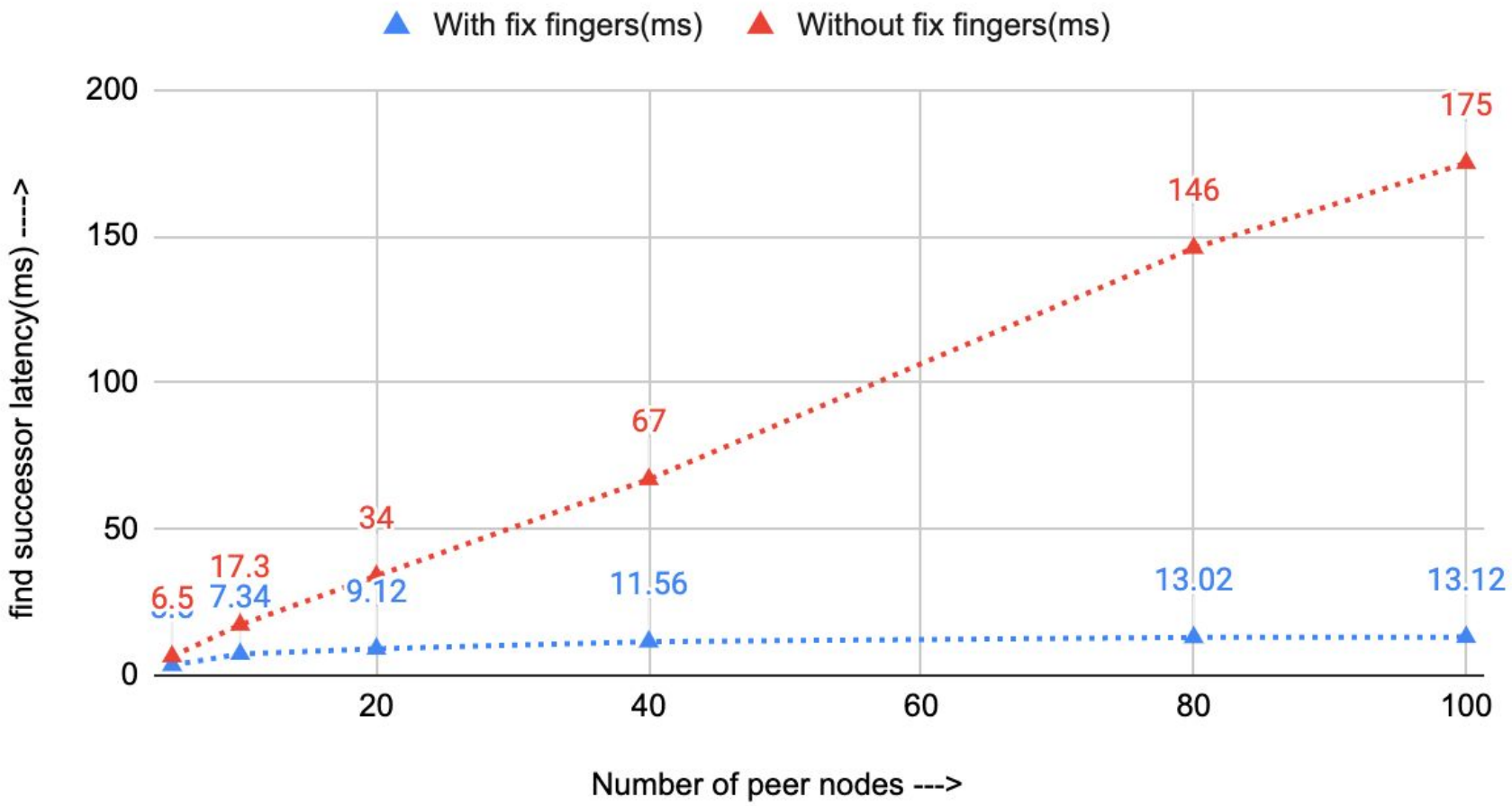
Demo #3 : Recovery, Key Transfer



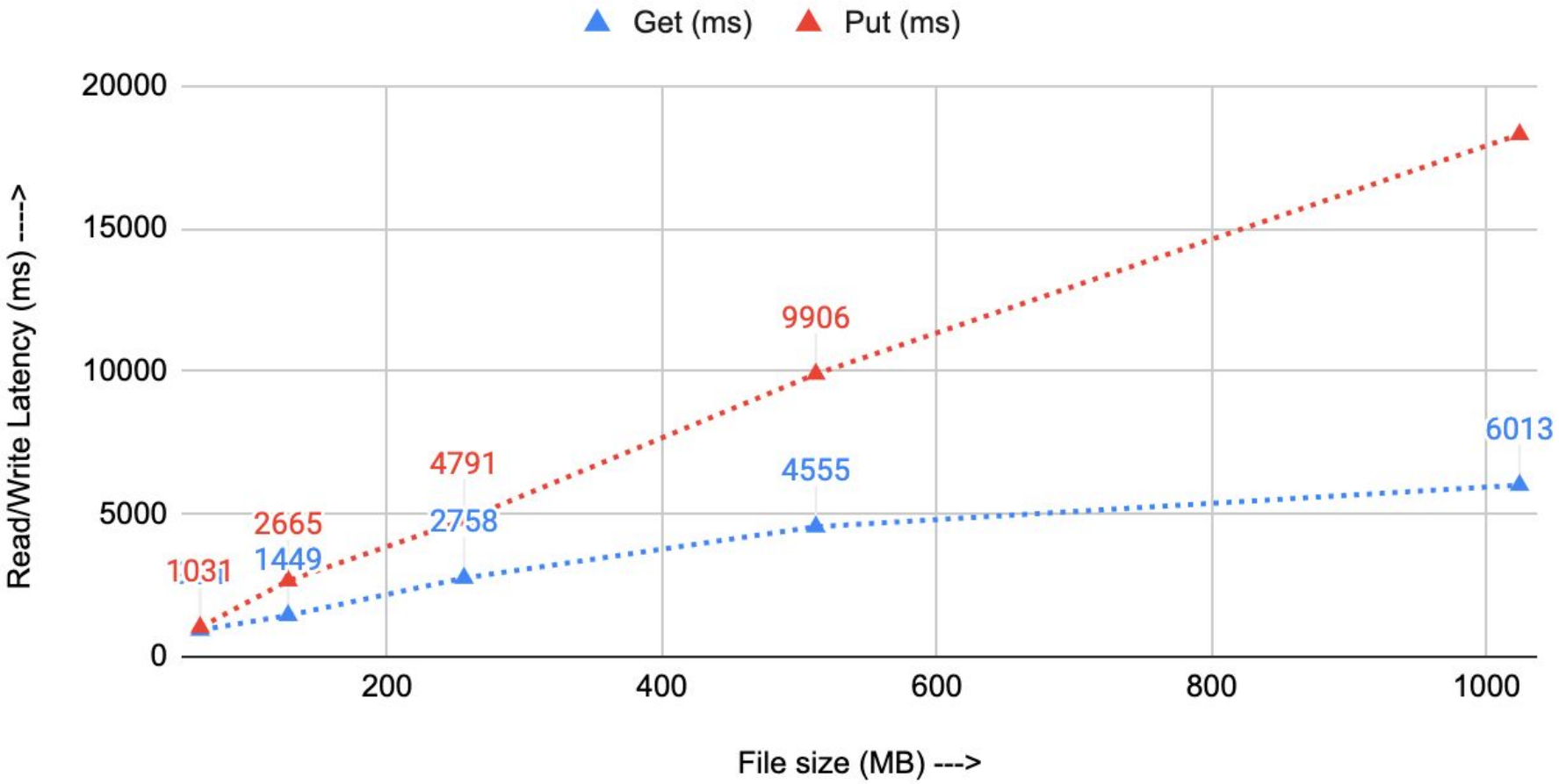


Performance Results

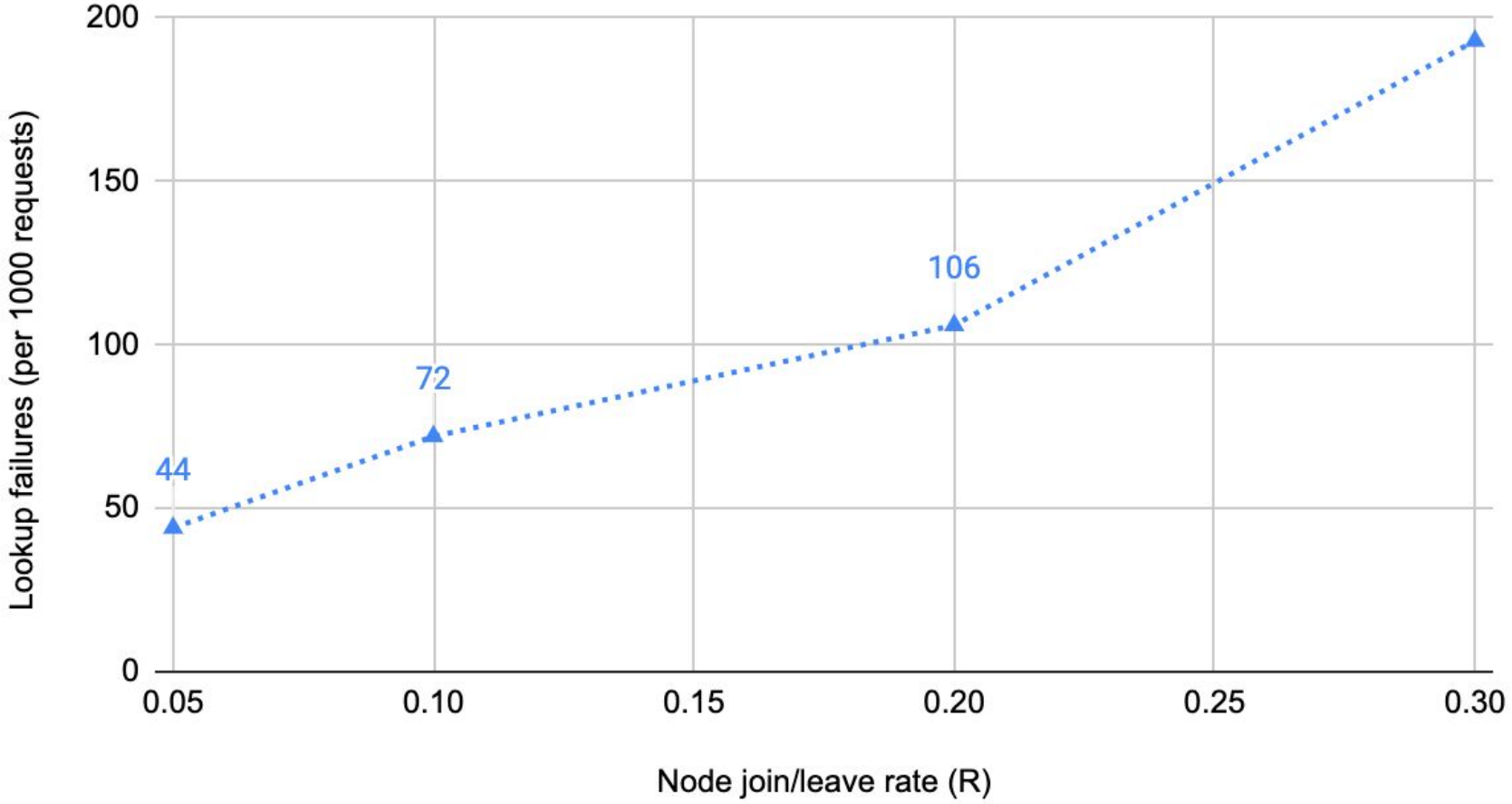
Find Successor Latency(ms) vs Number of nodes



Read/Write Latency vs File Size



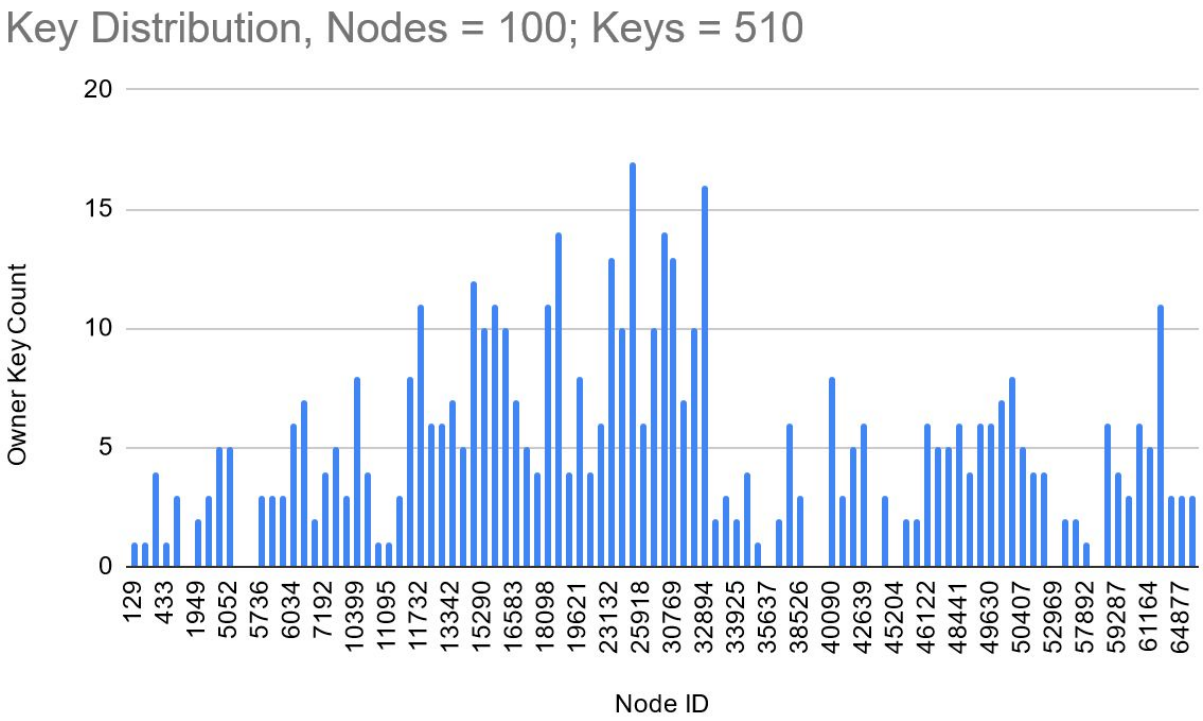
Lookup failures (per 1000 requests) vs Node join/leave rate (R)



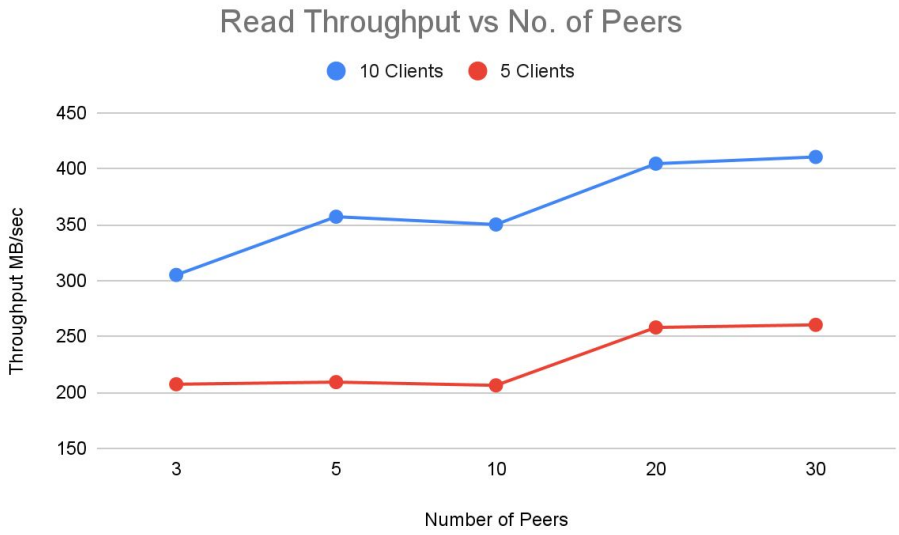
Key Distribution in Chord Ring

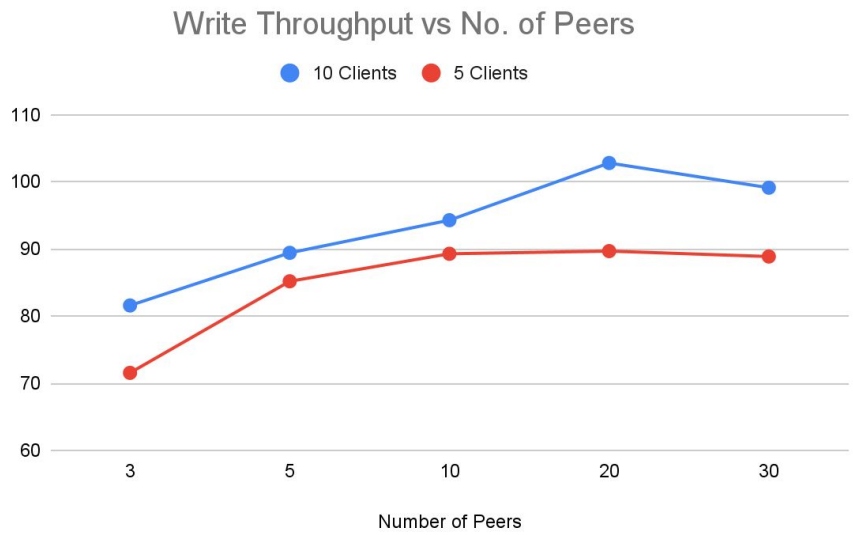
Avg keys per node = 510/100 = 5.1

Standard Deviation = 3.82720848
Normalized SD = 0.75

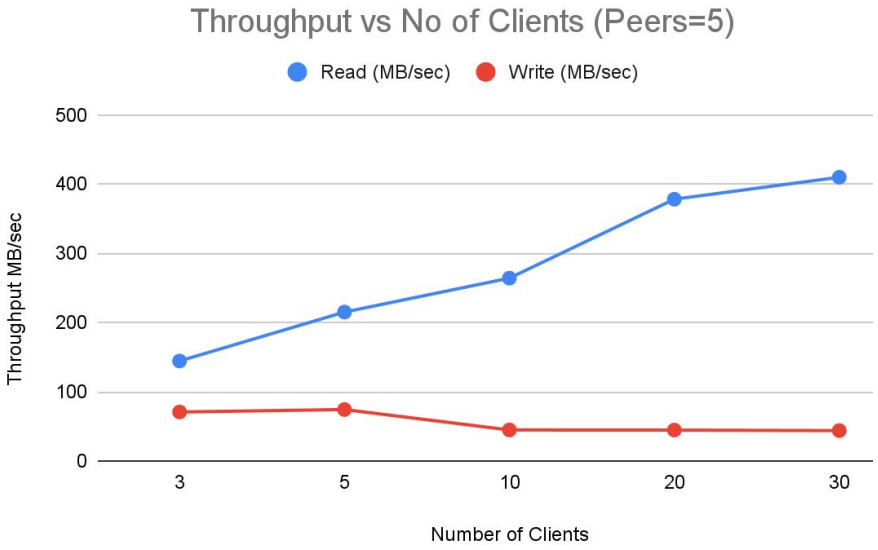


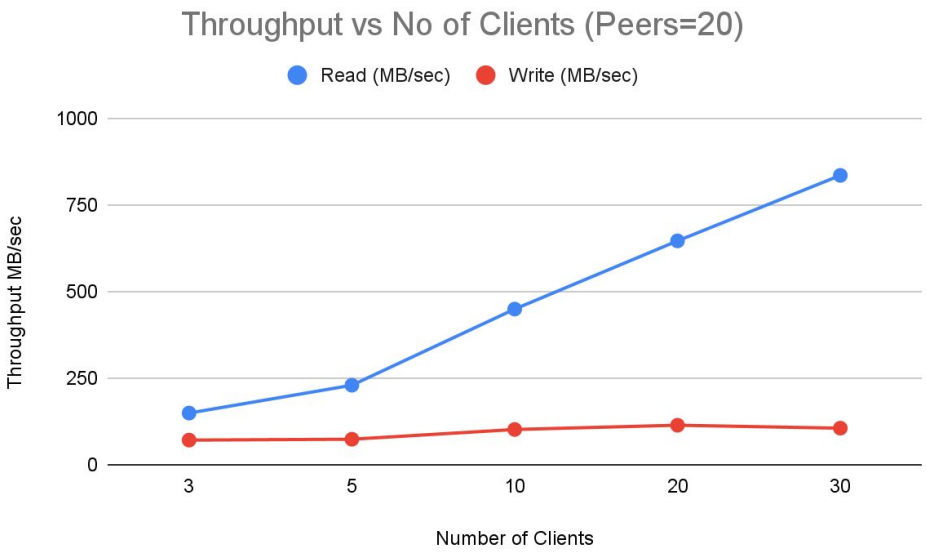
Throughput Results





Throughput Results





Future Work

- Garbage Collection - reclaim disk space
- Distribute load for get() requests among SuccessorList
- Use Merkel Trees to avoid transferring entire key set during recovery
- Support key versioning with timestamps
- Reduce internal RPC traffic by piggybacking, low level optimizations
- Implement Chord.leave() for voluntary departure

Thank You

Algorithm - CHORD

Roughly - things to cover

1. Intro:
   a. Symmetric nodes. No leader
   b. Flat namespace w.r.t a user
2. Ring formation
   a. Ring bits
3. Use of consistent hashing.
   a. How file keys are mapped to nodes
   b. How new nodes are placed in the ring
   c. Uniform key distribution and load balancing
4. Stabilize thread
5. Notify
6. Fix fingers
7. Successor list and replication
8. Consistency model Implemented
9.