P2P Hash Table

Leyang Li, Zanxiang Yin

Project repository: https://github.com/Leoreoreo/p2pHashtable

1. Goal

This project aims to create a distributed hash table (DHT) that is **scalable**, **available**, **data-persistent**, and **decentralized** with **dynamic membership**.

The DHT is constructed based on the **Chord protocol**. The completely decentralized Chord architecture avoids the single point of failure. In this system, nodes are organized in a circular topology, and each node is responsible for a specific range of the hash space. Each node keeps connection with its predecessor and successor, and nodes in its finger table. The finger table enables efficient **O(logN)** routing where N is the number of nodes in the Chord.

The design allows the system to scale seamlessly as new nodes join and existing nodes crash, providing an adaptable solution for environments with dynamic node participation. We also replace the original Chord's stabilization protocol with an instant crash discovery and recovery method to eliminate outdated finger table pointers and support more frequent node joins and crashes. The mechanism is implemented by replicating each node's finger table and its pointed node information to its successor. This ensures that in the event of a node crash, its successor, equipped with all necessary information (data replica, finger table, and pointed node information), can seamlessly take over its predecessor's responsibilities.

The DHT employs replication for data persistence. By keeping a replication for each node on its successor, the system ensures zero data loss as one node crashes. The read and write operations follow **chain replication principle**, ensuring monotonic reads and sequential consistency. Read operations are routed to the successor of the target node, while update operations are first processed by the target node and then propagated to its successor for consistency.

2. Architecture

This section will describe in detail how the system's internals work. This includes the diagrams and detailed process of node operations (handling node join, leave, and crash), and client operations (insert, remove, lookup).

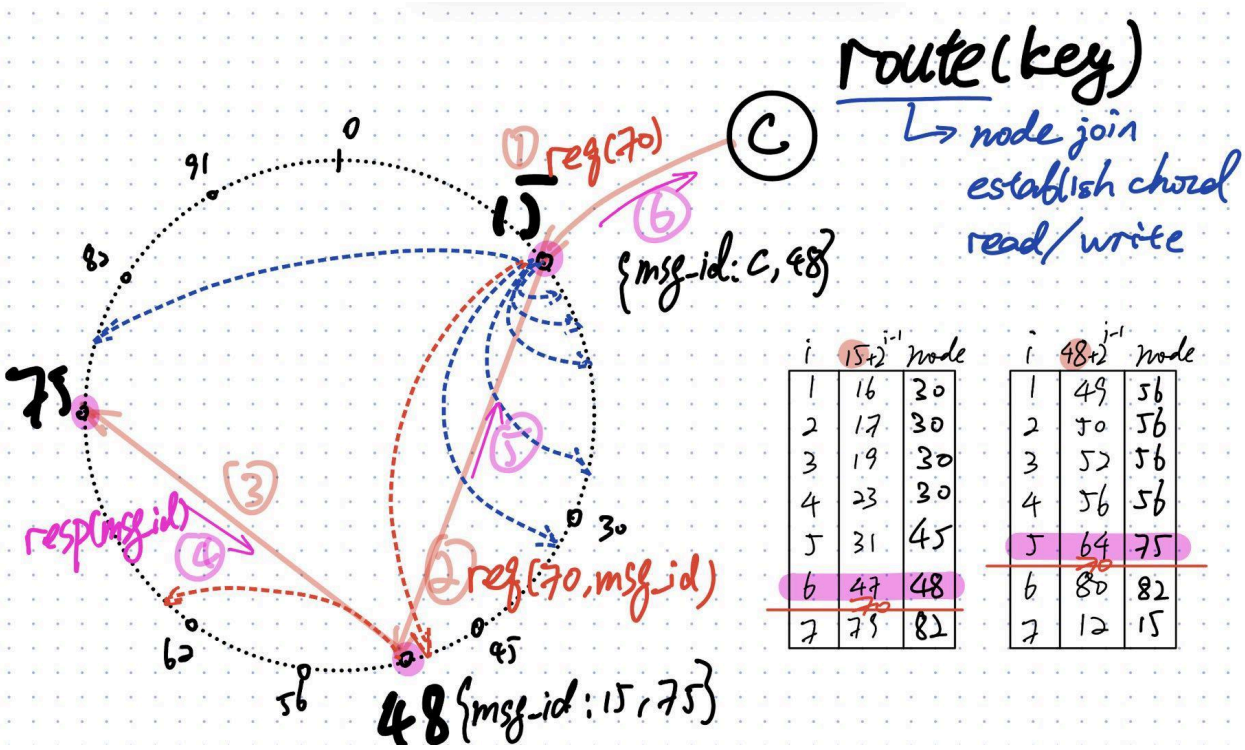
2.1. Node

Each node (server process) in the Chord ring represents a peer responsible for storing a portion of the hash space. Nodes are identified by a unique identifier which determines the node's position in the ring and its responsibility for a specific range of keys.

Each Node N Maintains Its:

- **Node ID (N):**
  - N is the upper-bound of the node's responsible key value
- **Predecessor (P) TCP Connection**:
  - To receive propagated updates (insert and remove) from P for P's replication in N
  - To receive propagated reads (lookup) from P to perform lookups
  - To receive FT (finger table) and PT (pointed table) updates from P for N's PFT (predecessor's finger table) and PPT (predecessor's pointed table)
- **Successor (S) TCP Connection:**
  - To propagate client operations (insert, remove, lookup) for N's replication in S
  - To forward N's FT and PT updates to S
- **Data Storage**:
  - N's responsible key-value pairs: keys in interval (P, N]
  - Replication of P's responsible key-value pairs
- **Finger Table (FT) and TCP Connections to Recorded Nodes (FT[i])**:
  - FT is an array of nodes and sockets: FT[i] = succ(N + 2^(i-1))
  - To help locate and route messages based on key efficiently in O(logN) hops.
- **Pointed Table (PT) and TCP Connections to Recorded Nodes**:
  - A table keeping track of nodes whose finger table points at N.
  - To receive routed read (lookup)
- **Predecessor's FT and PT (PFT and PPT)**:
  - When P crashes, N takes charge of P's responsibility, and contact all PFT and PPT nodes to re-establish P's connection to N quickly
- **Request Record:**
  - Whenever N receives a message from source_socket that requires routing
  - If the message is from a client:
    - give the message a unique msg_id
  - Route the message to target_socket after searching in the finger table
  - Record the message in a dictionary as {msg_id: (source_socket, target_socket)}
  - When N receives a response with msg_id
    - forward it to source_socket and delete msg_id row from the dictionary

2.2. Message Routing:



In the Chord system, the main goal of routing is to efficiently resolve a key k to the address of the node responsible for it, (succ(k)). Each node keeps track of its immediate successor (succ(p+1)). When a node p receives a request to resolve k, it checks if the key k falls within its responsibility. If it does, the node resolves the request. If not, it forwards the request to its successor. While this method is simple, it is inefficient because resolving a key might require traversing half the ring, making it unsuitable for large-scale systems.

Our Chord uses a finger table. Each Chord node maintains a finger table containing $s \leq m$ entries. If FTp denotes the finger table of node p, then

$$\text{FTp}[i] = succ(p + 2\text{^}i - 1)$$

To look up a key k, node p will then immediately forward the request to node q with index j in p's finger table where:

$$q = \text{FTp}[j] \leq k < \text{FTp}[j + 1]$$

or $q = \text{FTp}[1]$ when $p < k < \text{FTp}[1]$. When the finger-table size s is equal to 1, a Chord lookup corresponds to naively traversing the ring linearly, as we just discussed.

These references act as shortcuts, allowing a node to jump closer to the target in fewer steps. When a node receives a request to resolve k, it uses the finger table to find the closest node to k and forwards the request. This approach reduces the number of hops needed for resolution, resulting in a logarithmic O(log(N)) complexity.

By using finger tables, Chord achieves fast and scalable routing, making it highly efficient for large distributed systems.

2.3. Node Join

Stage 1: Route to position in the Chord

1. New Node C (with node_id C) contacts a random Node R with a join request.
2. R routes C to the appropriate position (between Node B and Node D)
3. C closes connection with R, establishes connection with D, and updates succ(C) = D

Stage 2: Join the Chord ring

4. C notifies D; D informs B (pred(D)) about C's info, and updates pred(D) = C
5. B connects to C, updates succ(B) = C, and informs C of its information (node_id, PT, FT)
6. C updates pred(C) = B and updates PPT, PFT

Stage 3: Update affected nodes' finger tables

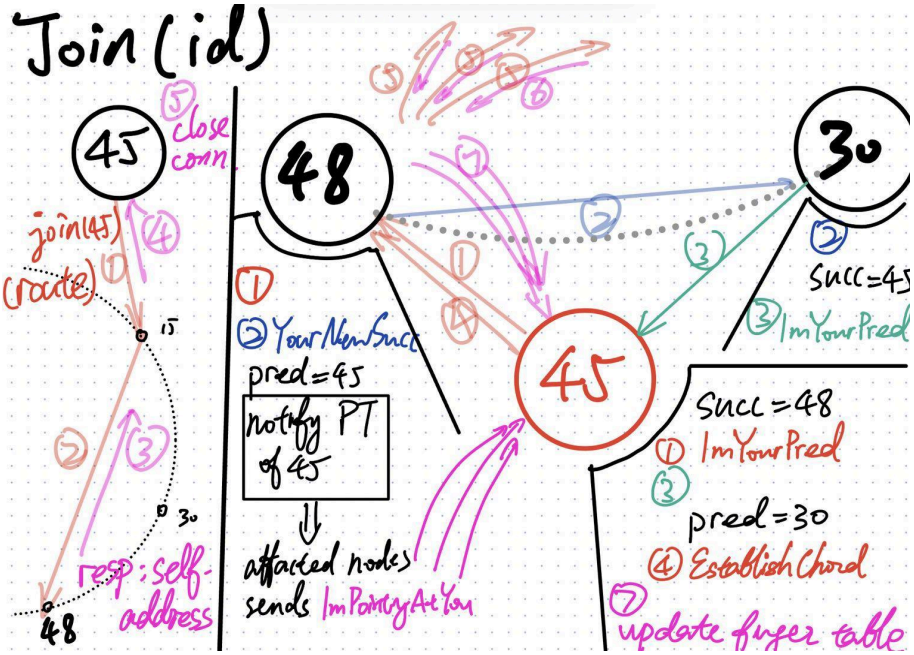
7. D notifies all nodes in its pointed table of C's join and C's information
8. For each notified node N:
    a. If FT[i] = D and N+2^(i-1) <= C:
        i. Update its FT[i] = C, then connect to C
        ii. Inform D that D loses one pointer, inform C that C gains one pointer
9. C and D receives affected nodes' information and update their pointed tables

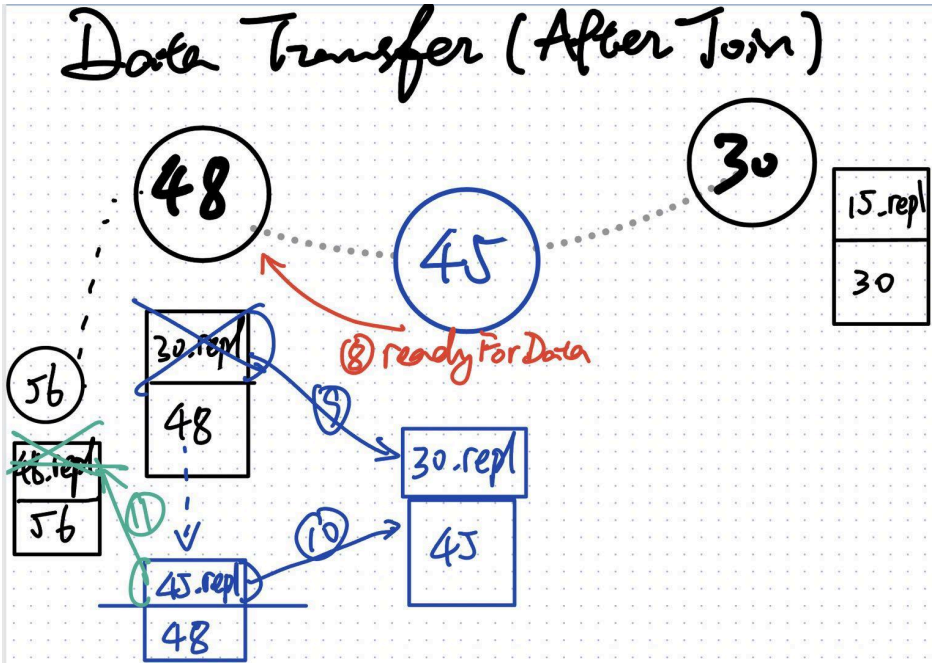
Stage 4: Establish finger table

10. C sends a series of establishChord(key) to D, where key is C's finger table target key
11. D routes the requests, and sends target nodes' addresses back to C
12. C then establishes its finger table by connecting to target nodes and informing them
13. C updates its finger tables, and the target nodes update their pointed tables
14. C informs all it's pointers that chordEstablishmentComplete, and all target nodes inform their successors to update their PPT
15. C informs D about of its FT and PT, and then D updates its PFT and PPT

Stage 5: Data Transfer

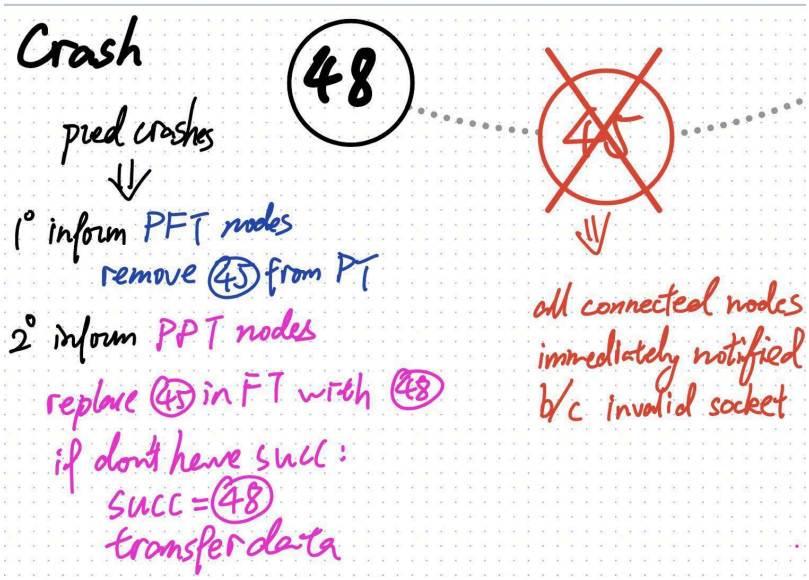
16. C informs D that it's ready for data transfer
17. D transfers all data with key <= C (C's responsible data and B's replication data) to C
18. D informs succ(D) to delete all data with key <= B (B's replication data)
19. D then deletes B's replication data locally

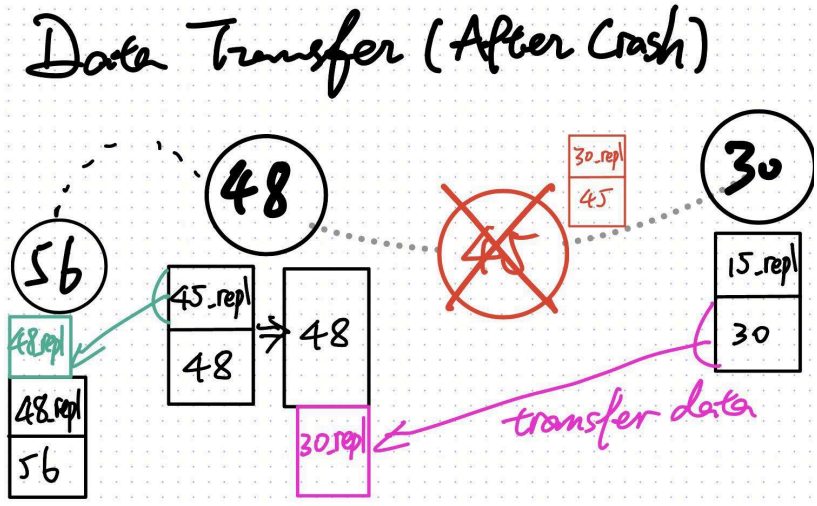




2.4. Node Crash / Leave

1. Node C crashes
2. All nodes connected to C (nodes pointing at, pointed by, and succ(C), pred(C)) notice the absence of C immediately because of the socket becoming invalid. They then remove the invalid sockets. D (succ(C)) sets pred(D) = None; B (pred(C)) sets succ(B) = None
3. D (succ(C)) contacts all nodes in PFT, the notified nodes remove C from their PT
4. D contacts all nodes in PPT, the notified nodes then replace all C rows with D in their FTs, establish connection with D, and inform their successors to update PFTs
    a. One contacted node B has None succ, so sets its succ(B) = D, and connects to D
5. D therefore updates its PT to include new nodes, and inform succ(D) to update PPT
6. D then sets pred(D) = B, and updates its PPT and PFT
7. B transfer its data to D as replica, D transfers its data (C's replication) to succ(D)





Note: Socket Management

Every time a node intends to establish a new socket connection with a new node, it always searches in its socket pool for a socket with the same node_id, and reuses the socket whenever possible. Therefore, our system avoids establishing multiple TCP connections between each pair of nodes.

2.5. Client Operations

The client sends requests to the DHT for operations: lookup, insert, and remove. All client operations experience the routing process (as described in 2.2). The client contacts a random node in the ring, which is responsible for routing all its requests to the responsible node and receiving all responses and sending them to the client.

In the write process, when a write(key) operation is initiated, the request is routed to the node responsible for the key. The responsible node processes the write request by updating its own data and concurrently propagates the request to its successor, and the successor performs the update on its predecessor's replication.

For the read process, when a read(key) operation is initiated, the request is first routed to the node responsible for the key, and then redirected to its successor. The successor processes the read request and sends back the response to the client.

This replication strategy combines chain replication principles with Chord's architecture. This sequential order of updates and reading from the last ensures sequential consistency and monotonic reads, even in the event of node crashes.

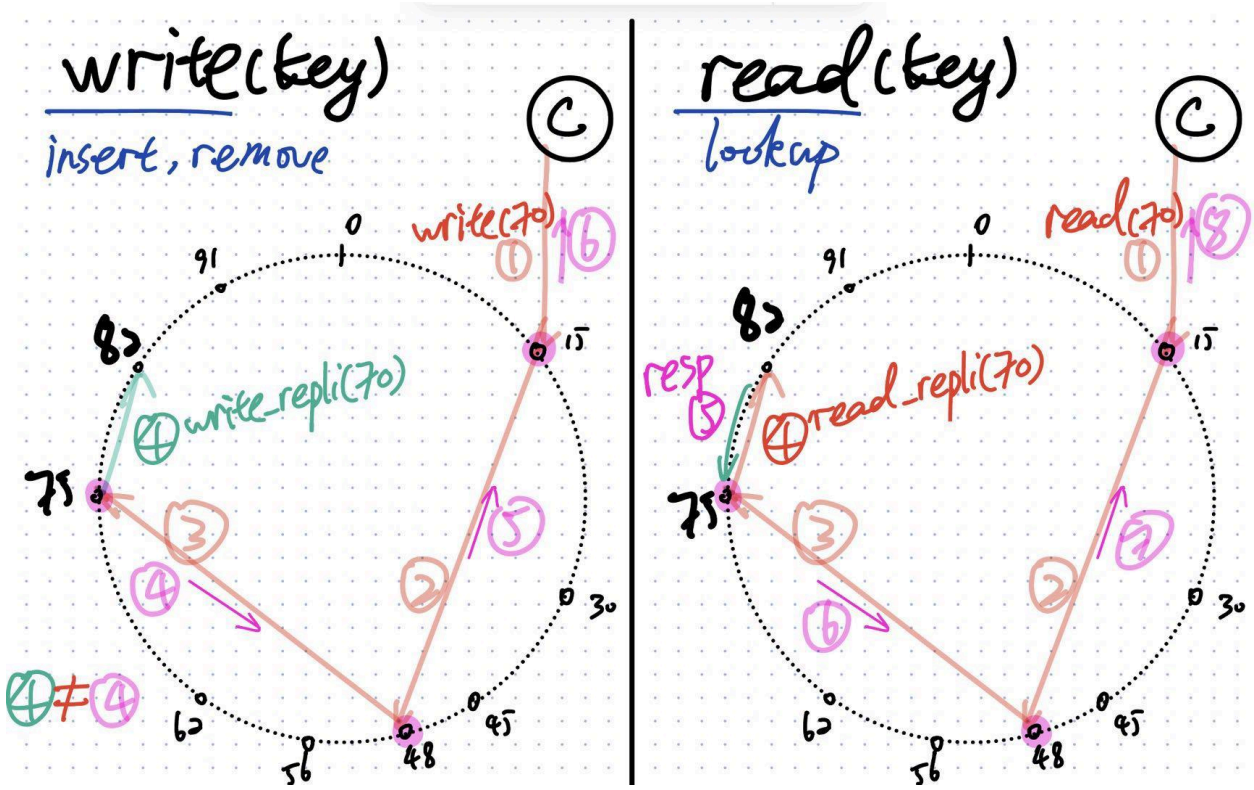
Read Operations (Lookup):

1. Client connects to a random Node R
2. Client sends a request lookup(k) to R, R saves it in R's Request Record
3. R checks if it's responsible for k. If yes, goto 6 with S=R
4. If no, check its FT and forward the request to the server N with max key <= k, goto 4 with R=N
5. S forwards the request to succ(s), succ(s) performs the lookup, and send back the response
6. The response flows back to the client in the same route

Write Operations (Insert and Remove):

1. Client connects to a random Node R
2. Client sends a request insert/remove(k) to R
3. R saves it in R's Request Record

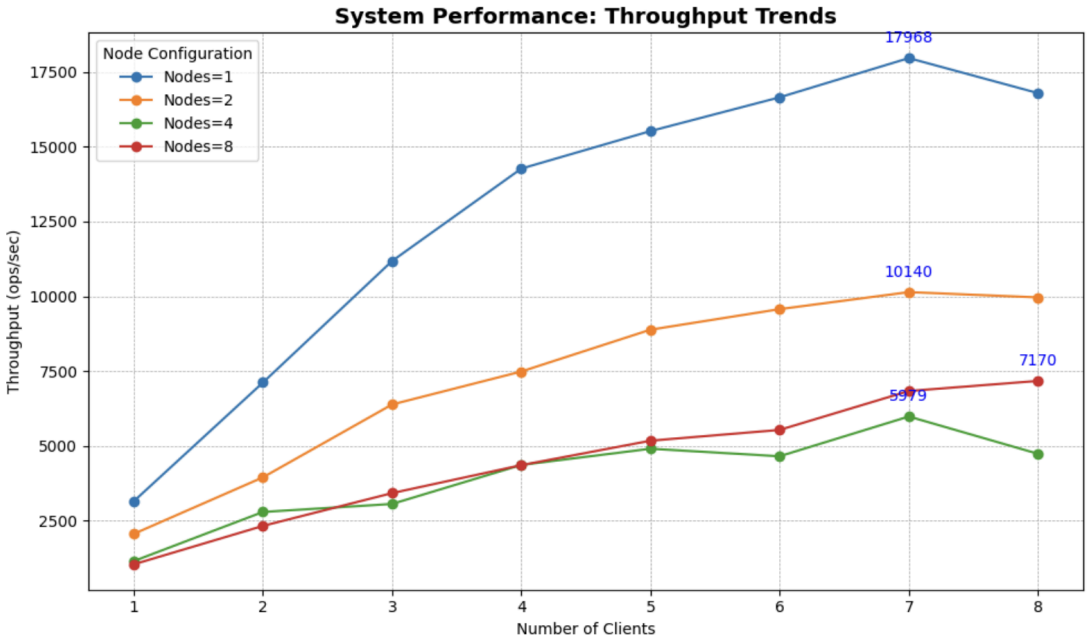
4. R checks if it's responsible for k. If yes, goto 6 with S=R
5. If no, check its FT and forward the request to the server with max key smaller than k, goto 4
6. S performs the insert/remove, and then forwards the request to succ(s)
7. Concurrently, succ(s) performs the insert/remove, and S send back the response
8. The response flows back to the client in the same route



3. Test and Measure Performances (LookUps)

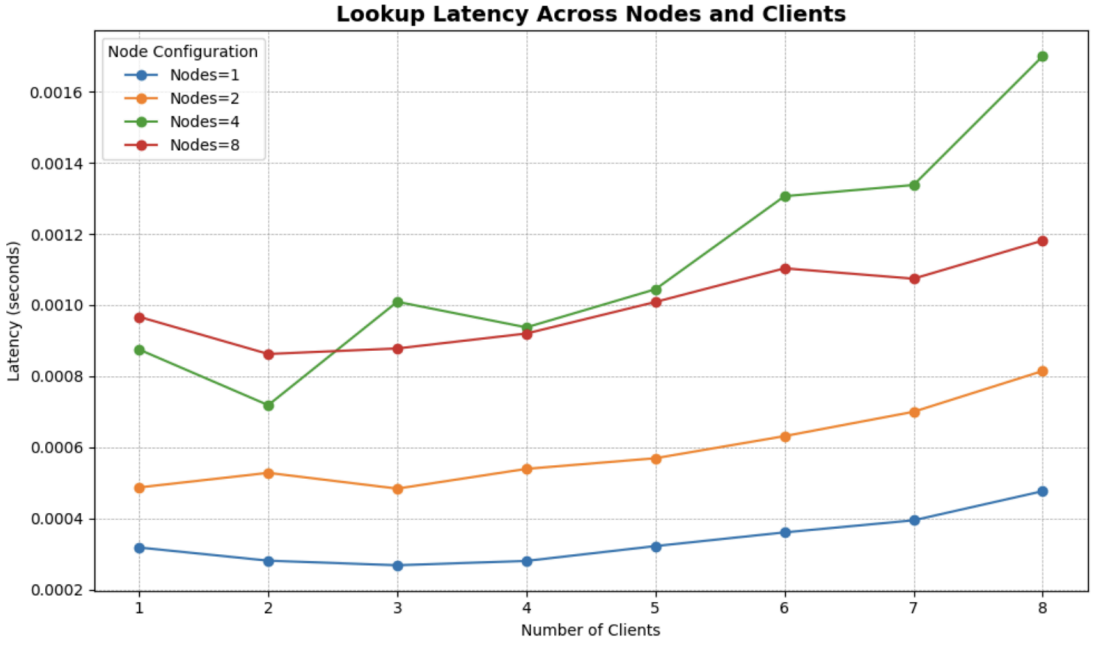
3.1. Throughput (Server's Perspective)

- The throughput increases steadily as client number increases until it reaches 7. This indicates a critical point where the system reaches its capacity. As client numbers continue to increase, some client operations need to wait for another operation to finish before being performed.
- As node number increases to 8, the critical point shifts right, because the increase in node number achieves better load balancing, indicating that each node reaches its maximum capacity when serving a larger number of clients. We may deduce that as client number continues to increase, the system with the most nodes can achieve the highest throughput.



3.2. Latency (Client's Perspective)

- As client number increases, the latencies increase because each node receives more requests and each request has more chances of waiting for another request to complete.
- As the node number N increases, the clients experience larger latency, because the Chord's routing takes O(logN) hops for each request.
- N=8 outperforms N=4 a little bit as client number increases, because although N=8 takes more hops for routing, each node is responsible for less requests because of load balancing. As client number increases, the benefit of load balancing becomes more evident.



4. Assumptions and Limitations

Our DHT system relies on the following assumptions for it to function properly and achieve better performance.

For our system to perform properly, we assume the following:

1. There should always exist at least three alive nodes. Because the data persistence is based entirely on replication, the system would suffer from entire data loss when the last node dies. Therefore, while the system supports relatively frequent node joins and leaves, the node number should always be greater than two (having only two live nodes can be extremely vulnerable to data loss as one node crashes).
2. We assume a moderate churn (join and crash) rate. Our system cannot tolerate two nodes crashing or joining within a very short time (~3 seconds). A new node join or crash during the last stabilization process would cause unexpected behaviors.
3. Client operations should be idempotent. As a node crashes, all clients directly connected to it are disconnected to the system. Moreover, all requests that are routed through the crashed node cannot send their responses back to the client. For example, the client may insert into the hash table, and the request reaches the responsible node and operation is carried out successfully. However, one of the nodes the request went through crashed, so the response can't be sent back to the client. The client would experience a time-out for this request, and it cannot tell if the insertion is successful. Therefore, the client may want to perform the update again.
4. Bounded network latency is necessary because we use time-out for client operations. Client gives up when not receiving the result after 5 seconds.
5. We assume that there is no malice. A malicious process pretending to be a server can mess up the chord structure easily because we haven't implemented authorization for API calls.

We also make reasonable and useful assumptions for our system to perform well:

1. We assume the number of nodes is large and node_ids are evenly distributed in the key range for Chord's routing efficiency (load balancing).