INF-3200: Distributed System Fundamentals
Assignment 2: Dynamic Membership

Ilya Taksis & Asbjørn Aarekol
UiT id: ita022@uit.no
UiT id: aaa040@uit.no November 2, 2023

1 Introduction

This assignment extends the key-value store network developed in Assignment 1 by introducing dynamic node participation. In this phase, we implement the capability for the nodes to join and leave the network dynamically. Nodes are organized based on a hash of their respective IDs, and each node maintains a pointer to its successor. To enable the dynamism of node operations, we implement additional HTTP API functionalities that support node ingress, egress, and crash simulations. This report will cover the design, implementation, testing, evaluation, and measurement of the system's throughput.

2 Technical background

2.1 Chord protocol

The Chord protocol is a distributed lookup with the capability to dynamically manage nodes—allowing them to join and leave—with finger-table stabilization[1]. It relies upon a few mechanisms to execute its functions. It uses consistent hashing to designate identifiers. While key identifiers are determined using an m-bit identifier via the SHA-1 hash function, node identifiers are made by hashing their IP addresses. In Chord's architecture, the node that succeeds a key is typically responsible for that key, giving its key location. Important to its design is the finger table, every node upholds a table with a maximum of m entries. Each entry denotes the ID of the first node that surpasses the key $n+2^{i-1}$ on the identifier circle. This structured approach ensures that Chord, with dynamic features, maintains an average logarithmic lookup time that remains efficient as the network grows.

3 Design and Implementation

This section covers the design and implementation of the Chord protocol. This section is also split up into two subsections, where the first covers the initialization of the node and the other subsection covers the different APIs.

3.1 Node initialization

Before a node commences its server operations, it undergoes several initialization procedures. This process is as follows:

1. The node retrieves its local IP address by getting a socket. Using the consistent SHA-1 hashing function on its IP and modulating with the cluster size, the node's ID is made, ensuring the Distributed Hash Table (DHT) retains a ring structure.

2. The node creates a finger table using its node ID, the total number of nodes in the Chord ring, a node data from its surrounding nodes. Employing the formula $n + 2^{i-1}$, where n is the node ID and i is the finger table index, succeeding keys are determined.

3. For dynamic joining, new nodes communicate with a designated prime node. This prime node assists the new node in positioning itself correctly within the ring, updating its successor lists, and initiating finger-table stabilization.
4. Finger-table stabilization ensures accurate and updated network configurations. Periodically, nodes validate their finger table entries.
5. The node's successor lists—a set of nodes immediately following it—help maintain the system's stability. This list is crucial, especially when nodes leave or fail, as it aids in making the ring whole again.

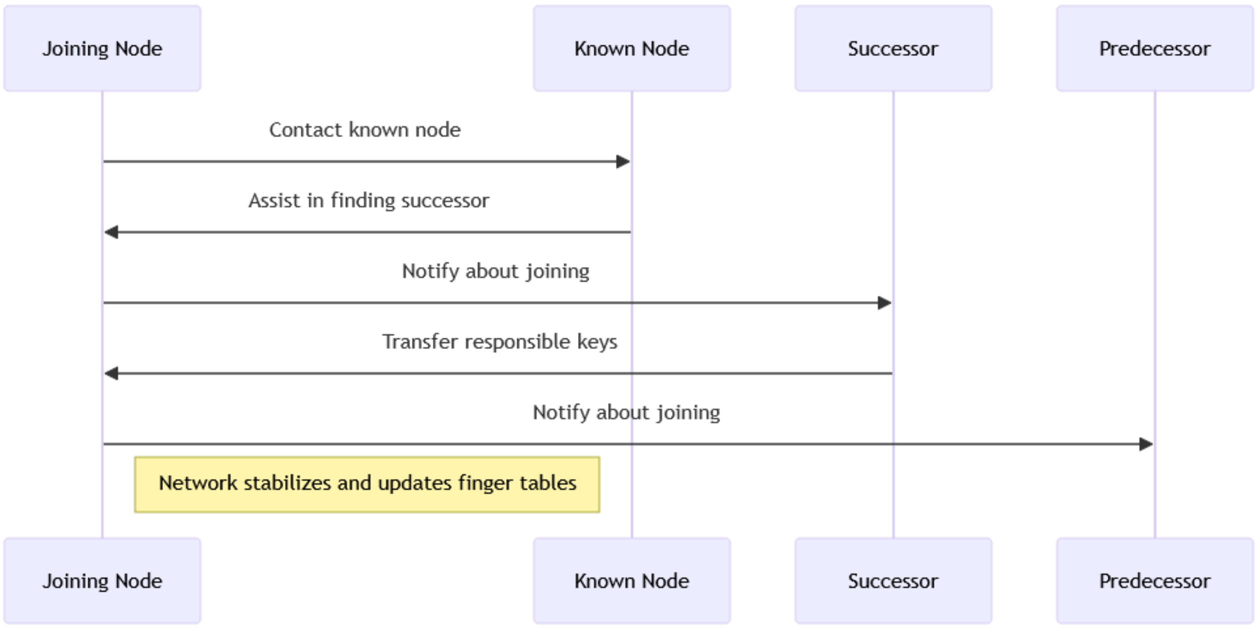


Figure 1: Chord Node Joining Process

With the necessary configurations, finger table stabilization, and provisions for dynamic operations in place, the node can initiate the HTTP server.

3.2 APIs

Every node in the system can respond to multiple API calls:

- **GET /node-info**: This API call provides detailed information about a node. Calling it, it returns a JSON object that includes the node's hashed key or ID (`node_hash`), its successor (`successor`), and other known nodes (`others`). The `others` field shows the node's predecessor, finger-table entries, and any other nodes known to the requester.
- **POST /join?nprime=HOST:PORT**: Upon receiving this API call, the node is instructed to join a network specified by the `nprime`'s HOST:PORT. Essentially, the node tries to join the given network to integrate into it.
- **POST /leave**: This call mandates the node to return to its initial standalone state. The departing node notifies its neighbors of its impending departure.
- **POST /sim-crash**: Simulating a node

crash, this call directs a node to stop processing external requests. Inter-node interactions, either regular operational messages or direct requests, are responded to with an error without further processing. During this simulated crash, the only processed request for the node is a `/sim-recover` call.

- **POST /sim-recover**: Representing a recovery from a simulated crash, this call tells the previously "crashed" node to start processing API calls again. The node tries to join the network again by looking at its successor list and tries to join the first responding one.

These API calls facilitate nodes to obtain node information, dynamically join or leave the network, simulate crashes, and recover from such simulated states.

4 Experiment and Analysis

In this section, we present the results of our experiments conducted to evaluate the performance of the Chord distributed hash table system under varying network node configurations. We conducted 10 runs for each of the following node configurations: 10, 20, 30, 40, and 50 nodes. Our primary metric for evaluation was time to stabilize, measured in `ms`.

4.1 Join Analysis

Figure 2, shows the relationship between the number of nodes and the average time required to join the Chord ring. Initially, the join time remains relatively stable with a minimal number of nodes, suggesting a low overhead for join operations. As the number of nodes escalates, a strong increase in join time is apparent, indicating that the complexity and overhead might increase in a non-linear fashion. The error bars, representing standard deviation, widen as the network expands, reflecting a higher variability in joining times.

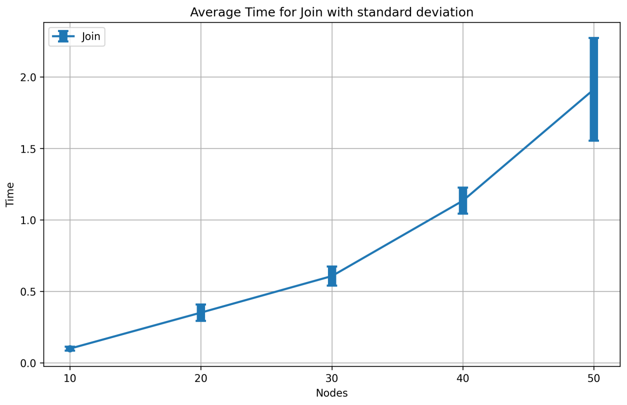


Figure 2: Average Time for Leave with SD

This variability could come from diverse factors such as network conditions, and the specific position a node occupies within the Chord ring. Overall, while the Chord ring exhibits the capacity to accommodate an increasing number of nodes, the joining becomes increasingly time-consuming, potentially due to increased routing information updates or network traffic.

4.2 Leave Analysis

Figure 3, shows the correlation between the number of nodes in the Chord ring and the average time taken for a node to leave. Evidently, as the number of nodes increases, the departure time shows a consistent increase, suggesting a linear increase in complexity. Moreover, the expanding standard deviation, depicted by the error bars, points towards an augmenting variability in leave times as the node count increases.

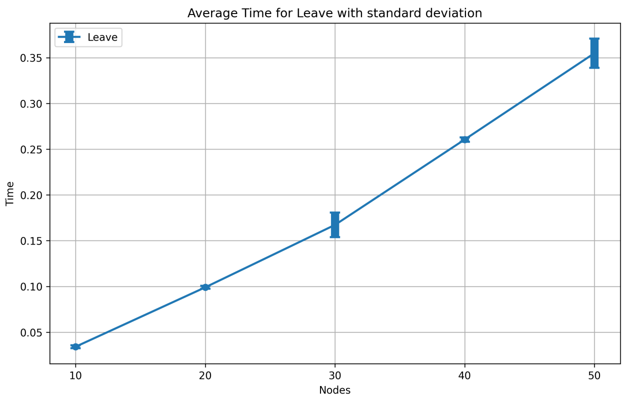
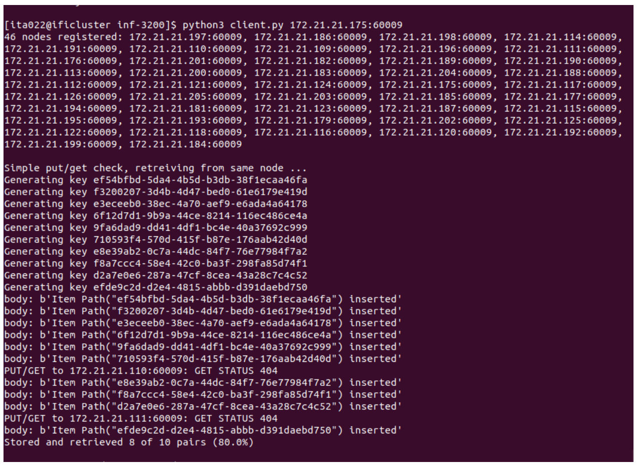


Figure 3: Average Time for Leave with SD

This increment in variability might be attributed to factors such as network conditions. In summary, the Chord ring's efficiency in handling node departures diminishes slightly with its expansion, potentially due to increased overheads associated with maintaining ring integrity.

4.3 Crash Recovery Analysis

The system is built on pinging the nodes current successor every 5 seconds and checking if they are alive. So after given a few seconds, the network in theory should have been stabilized after crashing N amount of nodes.



```
[ita022@ificluster inf-3200]$ python3 client.py 172.21.21.175:60009
46 nodes registered: 172.21.21.197:60009, 172.21.21.186:60009, 172.21.21.198:60009, 172.21.21.114:60009,
172.21.21.191:60009, 172.21.21.110:60009, 172.21.21.109:60009, 172.21.21.196:60009, 172.21.21.111:60009,
172.21.21.176:60009, 172.21.21.201:60009, 172.21.21.182:60009, 172.21.21.189:60009, 172.21.21.190:60009,
172.21.21.113:60009, 172.21.21.200:60009, 172.21.21.183:60009, 172.21.21.204:60009, 172.21.21.188:60009,
172.21.21.112:60009, 172.21.21.121:60009, 172.21.21.124:60009, 172.21.21.175:60009, 172.21.21.117:60009,
172.21.21.126:60009, 172.21.21.205:60009, 172.21.21.203:60009, 172.21.21.185:60009, 172.21.21.177:60009,
172.21.21.194:60009, 172.21.21.181:60009, 172.21.21.123:60009, 172.21.21.187:60009, 172.21.21.115:60009,
172.21.21.195:60009, 172.21.21.193:60009, 172.21.21.179:60009, 172.21.21.202:60009, 172.21.21.125:60009,
172.21.21.122:60009, 172.21.21.118:60009, 172.21.21.116:60009, 172.21.21.120:60009, 172.21.21.192:60009,
172.21.21.199:60009, 172.21.21.184:60009

Simple put/get check, retreiving from same node ...
Generating key ef54bfbd-5da4-4b5d-b3db-38f1ecaa46fa
Generating key f3200207-3d4b-4d47-bed0-61e6179e419d
Generating key e3eceeb0-38ec-4a70-aef9-e6ada4a64178
Generating key 6f12d7d1-9b9a-44ce-8214-116ec486ce4a
Generating key 9fa6dad9-dd41-4df1-bc4e-40a37692c999
Generating key 710593f4-570d-415f-b87e-176aab42d40d
Generating key e8e39ab2-0c7a-44dc-84f7-76e77984f7a2
Generating key f8a7ccc4-58e4-42c0-ba3f-298fa85d74f1
Generating key d2a7e0e6-287a-47cf-8cea-43a28c7c4c52
Generating key efde9c2d-d2e4-4815-abbb-d391daebd750
body: b'Item Path("ef54bfbd-5da4-4b5d-b3db-38f1ecaa46fa") inserted'
body: b'Item Path("f3200207-3d4b-4d47-bed0-61e6179e419d") inserted'
body: b'Item Path("e3eceeb0-38ec-4a70-aef9-e6ada4a64178") inserted'
body: b'Item Path("6f12d7d1-9b9a-44ce-8214-116ec486ce4a") inserted'
body: b'Item Path("9fa6dad9-dd41-4df1-bc4e-40a37692c999") inserted'
body: b'Item Path("710593f4-570d-415f-b87e-176aab42d40d") inserted'
PUT/GET to 172.21.21.110:60009: GET STATUS 404
body: b'Item Path("e8e39ab2-0c7a-44dc-84f7-76e77984f7a2") inserted'
body: b'Item Path("f8a7ccc4-58e4-42c0-ba3f-298fa85d74f1") inserted'
body: b'Item Path("d2a7e0e6-287a-47cf-8cea-43a28c7c4c52") inserted'
PUT/GET to 172.21.21.111:60009: GET STATUS 404
body: b'Item Path("efde9c2d-d2e4-4815-abbb-d391daebd750") inserted'
Stored and retrieved 8 of 10 pairs (80.0%)
```

Figure 4: Recovery result after crash

Through testing by crashing 1, 2 or 3 nodes in a burst, the network was still stable and could find all the keys and still functioned as a ring. But when we came to 4 nodes, the chord ring still functioned as a interconnected ring and could find all their right neighbors, but the ring started to loss keys when adding and trying to find them as we see in Figure 4. Even though we can crash more and more nodes in burst, the ring can still stabilize and create a ring but the finger tables gets more scramble up. Because of that, we will start to loss more and more data because the nodes tries to PUT/GET keys into fingers that has not been properly updated.

Recovery of the crashed node is dependent on its successor list. So the Node can join back to the network as long as one of the nodes inside of the successor list is still active. But if every node in the successor list is crashed, then the crashed node cannot be recovered.

4.4 Discussion

Our analyses provide some insights into how the Chord protocol behaves under join and leave. The increase in join time with more nodes suggests that as the network grows, it becomes more challenging for a node to integrate. This could be due to factors such as additional routing information updates or increased network traffic.

Similarly, the consistent growth in departure times as more nodes are added indicates that leaving the network isn't a simple operation. It emphasizes the care taken by the Chord protocol to maintain network integrity, especially with more nodes in play.

In conclusion, our findings give a clearer understanding of how the Chord system behaves under different node configurations. It's evident that while the system is robust, its performance can change as the network grows, which is important to consider in real-world applications.

5 Conclusion

In conclusion, this implementation has explored the dynamic joining and leaving of the chord protocol. By leveraging finger tables and successor lists, Chord offers an efficient and scalable solution for distributed key-value storage. As the network scales, certain challenges become apparent, such as increased join times, highlighting the difficulties of node integration in larger environments. Still, its logarithmic lookup time and robustness make it a valuable tool in the realm of distributed systems despite its challenges with dynamic membership.

References

[1] Stoica, I. et al. (2003) ‘Chord: A scalable peer-to-peer lookup protocol for internet applications’, IEEE/ACM Transactions on Networking, 11(1).