INF-3200: Distributed System Fundamentals

Assignment 3: Distributed Hash Table - Dynamic Membership

October 31st, 2025

August S. Hindenes
Department of Computer Science
UiT The Arctic University of Norway
Tromsø, Norway
ahi025@uit.no

Karoline W. Benjaminsen
Department of Computer Science
UiT The Arctic University of Norway
Tromsø, Norway
kbe173@uit.no

I. Introduction

This report presents the solution for Assignment 3 in INF-3200. In the previous assignment, we implemented a distributed hash table (DBT) based on the Chord protocol, where the system was initialised with a fixed number of nodes. In this assignment, the goal was to extend that implementation to support dynamic membership, allowing nodes to join and leave the network during runtime. The network is still structured according to the Chord protocol, in which nodes are organised by the hash of their identifiers and maintain pointers to their successors.

To evaluate the performance and robustness of our implementation, we conducted a series of experiments measuring (1) the time required to grow the network to 32 nodes, (2) the time required to shrink the network from 32 nodes to 16 nodes, and (3) the network's tolerance to bursts of node crashes.

The rest of this report is organised as follows. Section II introduces key concepts and background information relevant to this assignment. Section III describes the overall design and mechanism enabling dynamic membership in the Chord protocol, while Section IV details the specifics of our implementation. Section V outlines the methodology used to test and evaluate our implementation, and Section VI presents the results of these experiments. And finally, Section VII analyses the results and discusses our findings, and Section VIII concludes our report.

II. Technical Background

In this assignment, the focus shifts from static to dynamic network management in a Chord-based distributed hash table. While the previous assignment assumed a fixed number of nodes, real-world distributed systems must handle nodes frequently joining or leaving the network, either intentionally or due to failures. Supporting dynamic membership is important for scalability and fault tolerance, as it enables the system to adapt to changes with minimal disruption [1].

Dynamic membership introduces several challenges not present in a static network. As nodes continuously join and leave, each node must update its routing information to maintain correct lookups. Outdated successor, predecessor, or finger table entries can lead to failed or incorrect lookups. Frequent changes create inconsistencies in the ring, so the system must implement mechanisms to restore consistency and ensure that the network remains reliable over time [1].

A. Dynamic Membership in Chord

To handle dynamic changes. Chord relies on a series of periodic stabilisation procedures. These allow nodes to discover newly joined nodes, update their successor and predecessor, and ensure that finger tables remain consistent over time. Through these mechanisms, Chord guarantees that even with frequent joins and leaves, the network will eventually converge to a stable state and preserve its logarithmic lookup performance [2].

III. Design

The general design of our implementation follows the Chord protocol described by Stoica *et al.* in [2].

A. Create and Join

When a new node is created, it is initialised as a single-node network. The node's predecessor is set to NULL, and its successor is defined as the node itself, as shown in Figure 1 [2]. This establishes the minimal consistent state of the network, where the node maintains a valid ring structure of size one.

```
1: procedure CREATE
2:     predecessor = NULL
3:     successor = n
```

Fig. 1. Pseudocode for creating a new single-node Chord ring for node n [2]

A node joins an existing Chord ring through a known seed node. It requests the seed node to locate its immediate successor and updates its successor pointer accordingly, as shown in Figure 2. However, this operation alone does not inform the rest of the network about the new node's presence [2].

```
1: procedure JOIN(n')
2:     predecessor = NULL
3:     successor = n'.find_successor(n)
```

Fig. 2. Pseudocode for n joining a Chord ring containing node n' [2]

B. Stabilization Protocol

As the set of participating nodes in the system changes, Chord maintains a consistent ring structure through a set of *stabilisation* functions executed by each node periodically. These background tasks ensure that the finger tables are updated and that the successors and predecessor pointers are correct [2].

The core of this process, shown in Figure 3, allows a node to verify whether its current successor is still correct. The node queries its successor for its predecessor and updates its own successor if a closer node has joined the ring. It then calls the notify function to inform the successor of its existence [2].

```
1: procedure STABILIZE
2:     x = successor.predecessor
3:     if x ∈ (n, successor) then
4:         successor = x
5:     successor.notify(n)
```

Fig. 3. Pseudocode for a periodic stabilization function for node n [2]

The notify procedure, illustrated in Figure 4, allows a node to update its predecessor if necessary. When it is invoked by the stabilize function, it informs a node that another node might be its predecessor. If the current predecessor is NULL, or the notifying node's identifier lies between the node's existing predecessor and itself on the ring, the node updates its predecessor to the notifying node [2].

```
1: procedure NOTIFY(n')
2:     if predecessor is NULL or n' ∈ (predecessor, n)
       then
3:         predecessor = n'
```

Fig. 4. Pseudocode for node n' notifying that it might node n predecessor [2]

Each node also periodically executes the fix fingers function, shown in Figure 5, to keep its finger table entries correct. This is how existing nodes incorporate new nodes into their finger table. Each call to fix fingers updates one entry in the finger table [2].

```
1: procedure FIX FINGERS(n')
2:     next = next + 1
3:     if next > m then
4:         next = 1
5:     finger[next] = find_successor(n + 2^(next−1))
```

Fig. 5. Pseudocode for a periodic function to refresh finger table entries for node n [2]

Finally, the check predecessor function, shown in Figure 6, detects if a node has failed by periodically verifying that a node's predecessor is still responsive. If it has failed, the predecessor pointer is reset [2].

```
1: procedure CHECK PREDECESSOR
2:     if predecessor has failed then
3:         predecessor = NULL
```

Fig. 6. Pseudocode for a periodic function to check whether predecessor for node n has failed [2]

C. Leave

During a voluntary departure, a node performs a *graceful* leave. Before leaving the network, the node notifies both its successor and predecessor. The successor then updates its predecessor pointer to reference the leaving node's predecessor, while the predecessor updates its successor pointer to the leaving node's successor. In effect, the leaving node closes the gap in the ring before leaving, enabling the network to stabilise more quickly than if the exit is treated as an unexpected failure [2].

IV. Implementation

The implementation follows the design described closely for the most part, but there are some key differences. On join, the node tries to populate index 2, 4, and 8 of the finger table instead of solely relying on the `fix_fingers` maintenance routine. This should reduce the time required for a newly joined node to fully integrate into the network. We also set *predecessor* to the current node itself instead of null, as specified in the chord paper, this is mostly due to practical reasons in our maintenance routines.

The nature of multithreaded Rust applications also forces us to make some changes to the flow of functions to prevent holding locks unnecessarily while making RPC calls. Holding a lock during these slow parts of the program can cause a buildup of queued network requests, leading to network timeouts or, in the worst case, deadlocks of the entire Chord network.

V. Experiments

To evaluate the performance and robustness of our implementation, three benchmark experiments were performed: network growth, network shrinking, and crash tolerance. All tests were performed using a benchmark script that interacts

with each node through an HTTP API, allowing for controlled joins, leaves, crashes and recoveries.

In each experiment, a network stability checker determined when the network had converged. Stability was defined as the point when all reachable nodes formed a closed ring of the expected size, verified through repeated successor traversals. The network was considered stable after several consecutive consistency checks. However, some minor inconsistencies in finger table entries were tolerated, as these were expected to be corrected gradually in time.

Each experiment was repeated three times to reduce variance, and the average stabilisation time and standard deviation were recorded.

A. Grow Network

In the network growth experiment, the benchmark measured the time for the network to grow from a single node to various target sizes: 2, 4, 8, 16, and 32 nodes. At the start of each test, all nodes were reset to ensure a clean state. The first node initialised a single-node Chord ring, and the remaining nodes joined sequentially. Once all join requests had been issued, the benchmark monitored the network until it was considered stable. The stabilisation time was defined as the period between the first join operation and the moment the network was verified as stable.

B. Shrink Network

The network shrinking experiment evaluated how quickly the network stabilised after several nodes left voluntarily. Starting from a stable 32-node network, a set of nodes was instructed to leave. The benchmark then measured how long it took for the remaining nodes to stabilise the network. This procedure was repeated for configurations $32 \rightarrow 16$, $16 \rightarrow 8$, $8 \rightarrow 4$, and $4 \rightarrow 2$. The stabilisation time was defined from the first leave call till the network had stabilised.

C. Network Tolerance

The crash tolerance experiment assessed the network's resistance to simultaneous node failures. From a stable 32-node network, a random subset of $1 - 16$ nodes was crashed at the same time. The benchmark then measured whether or not the remaining nodes could stabilise into a consistent ring and how long this recovery took.

VI. Results

From Figure 7 we observe that network growth causes an increasing stabilisation cost. As the network size rises from 2 to 32 nodes, convergence time climbs from around 2 seconds to roughly 20 seconds.

In Figure 8, we observe that during shrinking, the network converges quickly. Graceful leaving allows transitions to half the network size in around 2 seconds for all sizes with low variance.

Crash tolerance is overall very strong. In Figure 9, we see that a 32-node network handles up to 15 simultaneous crashes perfectly. For 16 simultaneous crashes, one of three experiments fails.

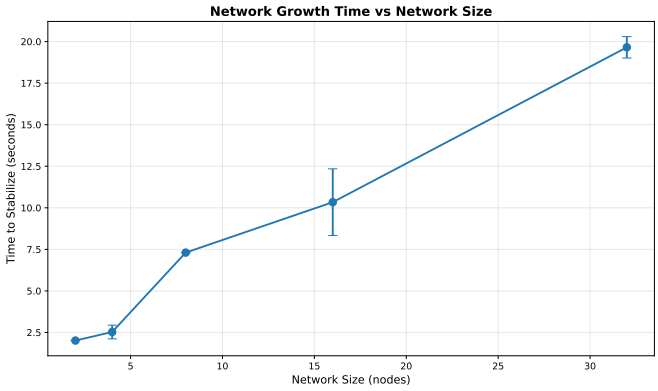

Fig. 7. Time to reach a stable ring for different network sizes

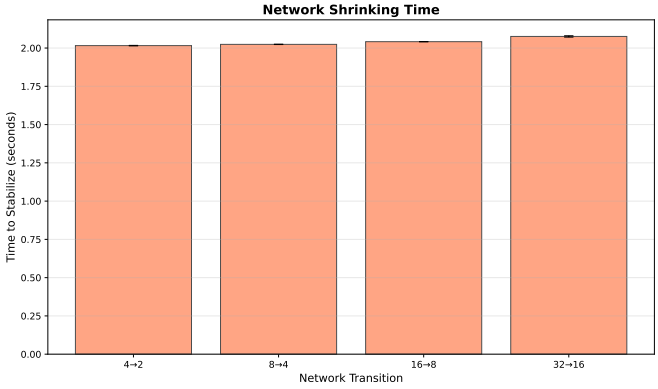

Fig. 8. Time to reach a stable ring when shrinking network size

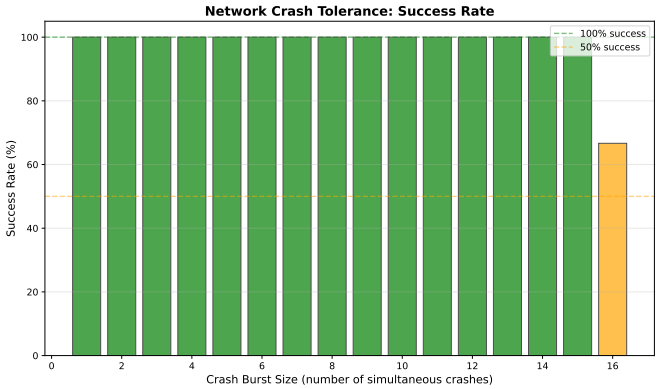

Fig. 9. Burst crash tolerance in a 32-node network

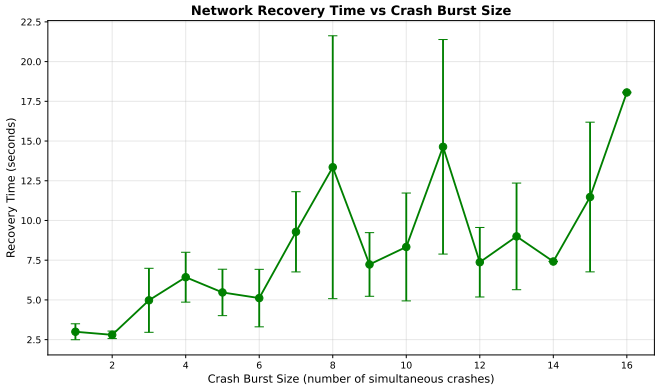

Fig. 10. Time to reach stable ring after burst crash in a 32-node network

From Figure 10 we can observe that there is quite a bit more variance in recovery times compared to the other times measurements. But in general, the convergence time trends upwards for larger bursts, from around 2.5-3 seconds at the lowest to around 18 seconds at the highest (mean values).

VII. Discussion

The results show that the Chord implementation performs as expected.

In the network growth experiment, stabilisation time increased roughly linearly with the number of nodes. This indicates that the implementation maintains a predictable performance as the network expands.

When shrinking the network, the stabilisation time remained nearly constant. This is likely due to the graceful leave mechanism, where departing nodes update their neighbours before leaving, reducing the need for recovery.

In the crash tolerance experiment, the network demonstrated strong resilience, successfully recovering from up to 15 simultaneous node crashes. A burst of 16 nodes, however, caused a timeout, indicating that the remaining nodes could not restore a consistent ring within the allowed time frame. Recovery time generally increased with the number of crashed nodes, but varied between runs, likely due to random crash locations affecting the ring differently.

Finally, while lookups (`PUTs` and `GETs`) worked correctly in newly stabilised networks, they often failed immediately after a shrink or crash. This is likely because the finger tables remained temporarily inconsistent even after the ring itself had reformed.

Overall, the results indicate good scalability and fault tolerance, with stabilisation mechanisms functioning as intended.

VIII. Conclusion

This assignment discussed the Chord implementation to support dynamic membership, allowing nodes to leave and join the network at runtime. The test results show that the network scales predictably, recovers efficiently from voluntary leaves, and is resistant to node crashes. Overall, the implementation meets the assignment requirements of a robust and dynamic distributed hash table.

References

[1] A. T. Joseph. Distributed hash tables (dhts). Accessed on October 31st, 2025. [Online]. Available: https://medium.com/@aleenat.csa2024/distributed-hash-tables-dhts-8546eef61f07

[2] I. Stoica, R. Morris, L.-N. David, D. M. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on networking*, vol. 11, no. 1, pp. 17–32, 2003.

Appendix A
Use of Generative A.I.

Generative artificial intelligence tools were used to enhance the language and clarity of the text.