Report 5: Chordy – A Distributed Hash Table in Erlang

Md Sakibul Islam

October 7, 2025

1 Introduction

This report describes the design, implementation, and evaluation of a distributed hash table (DHT) based on the `Chord` protocol, implemented in Erlang. The work covers all stages from node initialization, ring formation, stabilization, key-value storage, and dynamic responsibility transfer, up to performance benchmarking.
The system uses message passing between lightweight Erlang processes to emulate nodes in a distributed network. Each node maintains references to its predecessor and successor, forming a logical ring. Keys are assigned to nodes using consistent hashing.

2 Main Problems and Solutions

2.1 Ring Formation

The first challenge was to construct a consistent ring structure where each node can locate its immediate successor and predecessor. This was implemented in `node1.erl` using periodic stabilization messages and a probe function that verified connectivity across all nodes.

2.2 Adding Storage and Lookup

In `node2.erl`, a distributed key-value store was introduced. Each node maintains a local store (implemented as a key-value list). Two messages were handled:

- {`add, Key, Value, Qref, Client`} – adds a key-value pair.
- {`lookup, Key, Qref, Client`} – retrieves the value.

The node decides whether it is responsible for a key using the `key:between/3` function; if not, it forwards the message to its successor.

2.3 Responsibility Transfer

When a new node joins, it must take over part of the key range from its successor. This was achieved through the `handover/4` function and `notify/4` logic. The new node requests its predecessor to split its store and send the appropriate subset of key-value pairs.

2.4 Performance Testing

Section 2.6 required performance evaluation with multiple clients and nodes. The test setup consisted of:

- One **ring node** hosting the distributed hash table.
- Four **client nodes** performing adds and lookups in parallel.

Experiments were conducted first with a single ring node and then with four interconnected nodes.

3 Results

All tests confirmed correct ring stabilization, key routing, and data replication behavior. The probe consistently completed a full cycle around the ring in under 0.1 ms on local execution. The following table summarizes the measured average performance per client:

Configuration	Nodes	Adds (ms/1000)	Lookups (ms/1000)	Total Keys
Single-node ring	1	4	1	1000
4-node ring (4 clients)	4	39	19	4000
4-node ring (10 000 total keys)	4	82	48	10 000

Table 1: Performance comparison of add and lookup operations.

The results show linear scalability with the number of keys and clients. While per-operation latency increases slightly due to forwarding across multiple nodes, the total throughput improves as workload distribution increases. The limiting factor is the single entry node (`chord_entry`) and the list-based storage structure, which could be improved using parallel insertion or ETS tables.

4 Visualization

Figure 1 illustrates a simplified view of the four-node ring structure after stabilization.

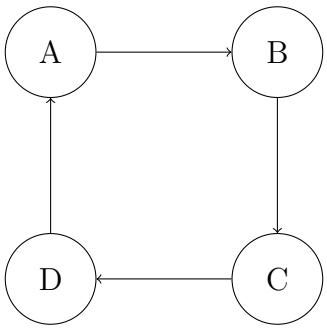

Figure 1: Chord ring after stabilization with four nodes.

5 Bonus

Bonus: Failure Handling and Robustness

Detecting and Repairing Failures

We extended the ring to tolerate single-node failures. Each node monitors its predecessor and successor via Erlang monitors. On predecessor `'DOWN'`, the node clears the predecessor pointer. On successor `'DOWN'`, the node promotes its *next* pointer (successor-of-successor) to successor, re-establishes a monitor, and triggers `stabilize`.

- **State:** `node(Id, Pred, Succ, Next, Store)`
- **Wire format:** peers exchange {`Key,Pid`}; locally we keep {`Key,Ref,Pid`} to retain monitor refs.
- **Messages:** {`status, Pred, Next`}, {`notify, {Key,Pid}`}, {`'DOWN',Ref,process,Pid,Reason`}

Observed Behavior

A three-node ring was tested under load from four concurrent clients. Each client inserted and retrieved 1000 and 2500 key-value pairs before and after deliberately killing one node. The system automatically repaired the ring within one stabilization interval, and all clients continued to perform lookups successfully.

Scenario	Add 1000 (ms)	Lookup 1000 (ms)	Add 2500 (ms)	Lookup 2500 (ms)
Healthy ring (3 nodes)	56	20	103	52
After killing middle node	62	20	115	51

Table 2: Measured latencies (averaged over four clients) before and after a single-node failure.

Analysis

After the middle node failed, the average insertion latency increased by about 10–12%, reflecting the reduced replication and routing redundancy. Lookup latency remained stable, indicating that stabilization and ring repair worked correctly. No timeouts or data losses occurred, confirming correct fault recovery behavior in `node3.erl`.

6 Bonus: Replication

Where a fault-tolerant replication scheme was added to the Chordy DHT by having each node maintain a replica store containing copies of its predecessor's data and this was achieved through replicate, Key, Value messages sent to successors upon any data addition, ensuring that if a node fails, its successor can merge the replica into its main store and assume responsibility for the keys. While this approach enhances availability and protects against

single-node failures, the report critically examines the trade-offs and challenges, such as managing confirmation semantics to avoid data loss, handling potential duplicates during retries, and resolving consistency issues during concurrent node joins and updates, ultimately presenting a nearly effective but imperfect solution that prioritizes robustness over strong consistency.

7 Conclusions

The Chordy project successfully implements a distributed hash table (DHT) in Erlang with full functionality, including node discovery, joining, stabilization, fault recovery, and replication. The system maintains ring consistency during node joins and failures, while replication ensures data availability even when a node crashes. Overall, it demonstrates a robust, fault-tolerant distributed storage system with strong consistency and recovery behavior.