Assignment 2 APD – CHORD Protocol in MPI

Student: Ionut Gabriel Mantu
Group: 333CA

General Description

This assignment implements a simplified, static version of the **CHORD** protocol using **MPI** (Message Passing Interface). The main goal is to simulate a Distributed Hash Table (DHT) where nodes are organized in a ring and collaborate to locate the node responsible for a specific key within an identifier space [0, 2^m - 1], where m=4.

The program simulates a fully distributed system, where each MPI process represents a CHORD node.

Implementation Details

The implementation was achieved by completing the provided code skeleton, focusing on routing logic and node interaction. The 5 major components implemented are detailed below:

1. Finger Table Construction (`build_finger_table`)

Each node constructs its routing table (Finger Table) initialization. Since the ring is static (nodes are known from the start and do not change), construction is done by directly iterating through the global sorted list of nodes.

For each entry `i` in the finger table (from 0 to M-1):

1. Calculate the start position: `start = (self.id + 2^i) % RING_SIZE` . This represents the "ideal target" the node wants to reach or skip to.
2. Find the successor of this position in the ring using the `find_successor_simple` function, which traverses the `sorted_ids` list and returns the first existing node that is greater than or equal to `start` .
3. The result is stored in `self.finger[i].node` .

This table is essential for the logarithmic performance of the CHORD algorithm, allowing jumps over large portions of the ring. For example, the first entry is the immediate successor, the second skips 2 positions, the third skips 4, etc.

2. Determining the Next Hop (`closest_preceding_finger`)

This function is the routing "engine". Its purpose is to find the most suitable node in the finger table to forward the request to, such that we get as close as possible to the searched key without overshooting it.

- Iterate through the finger table from the last entry (furthest) to the first.
- Check if the node in the finger table lies strictly within the interval `(self.id, key)` . The `in_interval` function correctly handles wrap-around cases (transition from 15 to 0).
- If a node satisfies the condition, it is returned immediately. The logic is "greedy": jump as far as possible.
- **Fallback:** If no finger is useful (case where the key is very close or the table offers no better jump), `self.successor` is returned. This guarantees progress and avoids infinite loops.

3. Handling Requests (`handle_lookup_request`)

This function receives a lookup request and decides its course:

1. Add the current node (`self.id`) to the path history (`msg->path`).
2. **Responsibility Check:** Check if the searched key belongs to the immediate successor (`in_interval(key, self.id, self.successor)`). In CHORD, a node is responsible for keys between its predecessor and itself, so during routing we assume we are looking for the key's predecessor.
   - If YES, it means the successor is responsible. Add the successor to the path and send the response message (`TAG_LOOKUP_REP`) directly to the request initiator (`msg->initiator_id`).
3. **Forwarding:** If the successor is not responsible, the request must be forwarded.
   - Call `closest_preceding_finger(key)` to determine the next node (`next_node`).
   - Send the `TAG_LOOKUP_REQ` message to the corresponding MPI rank of that node.

4. Sequential Lookup Initiation and Execution

To comply with the requirement of displaying results exactly in the order of the input file, I implemented a **sequential (chained)** execution logic instead of a parallel one.

- **Initiation (TODO 4):** In the `main` function, after the synchronization barrier, a lookup request is sent **only for the first key** in the input list (`lookups[0]`).
- **Chaining (TODO 5):**
  - When a node receives the response (`TAG_LOOKUP_REP`) for the current request, it displays the complete path.
  - Immediately after printing, it checks if there are more keys to search (`my_lookups_done < nr_lookups`).
  - If so, it extracts the next key from the array and automatically sends a new `TAG_LOOKUP_REQ` .
  - Thus, request `i+1` is launched only after request `i` has fully completed.

This approach eliminates the risk of messages arriving out of order from the network and guarantees deterministic output, as discussed on the forum.

5. Distributed Service Loop and Termination

The node enters an infinite `while` loop to process messages, which stops only when all nodes in the system have finished their tasks.

- **Message Handling:** Uses `MPI_Recv` with `MPI_ANY_SOURCE` and `MPI_ANY_TAG` to react to any event.
- **Termination Protocol:**
    - If a node has received responses for all its keys (has nothing left to send in the sequential chain), it sends a `TAG_DONE` message to **all** other nodes (manual broadcast).
    - Special case: a node with 0 keys to search sends `TAG_DONE` immediately at the start to avoid blocking the system.
    - The main loop counts the `TAG_DONE` messages received from others (`finished_nodes`).
    - The process stops only when `finished_nodes == world_size` (all participants have signaled completion).