Assignment 1

Student Name: Bratin Mondal

Roll Number: 21CS10016

Course Name: Distributed Systems

Course Number: CS60064

1 Problem 1

Let us assume that the client sent the query message at $t = 0$. The message took x milliseconds to reach the server. Since nothing has been specified regarding the server's processing time, we assume it is negligible. The server sends the response message at $t = x$, and the response message took y milliseconds to reach the client. The client receives the response message at $t = x + y$. The client then takes 1 millisecond to process the message and finally sets its clock to the server's clock at $t = x + y + 1$. It is given that $x \leq 2$ and $y \leq 2$. (Here, all time values are with respect to a global reference time.)

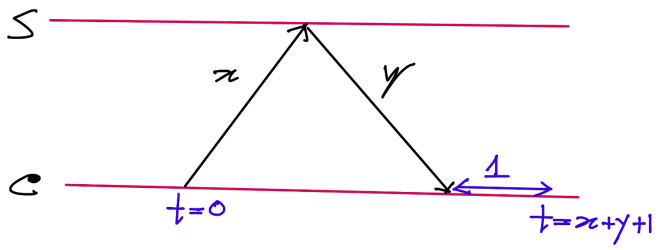

Figure 1: Timeline of Client-Server Communication

The server receives the query message at $t = x$ and immediately sends the response message. The response message to the client contains only this value. Now, the client estimates the round-trip time (RTT) as:

$$\text{RTT} = x + y + 1$$

The client adds $\frac{\text{RTT}}{2}$ to its clock to synchronize with the server's clock. Therefore, at $t = x + y + 1$, the client's clock is set to:

$$\text{Client's Clock} = x + \frac{x + y + 1}{2}$$

Meanwhile, the server's clock has the correct value:

$$\text{Server's Clock} = x + y + 1$$

The difference between the server's clock and the client's clock is given by:

$$\text{Server's Clock} - \text{Client's Clock} = \frac{y + 1 - x}{2}$$

This difference is maximized when $y = 2$ and $x = 0$, and minimized when $y = 0$ and $x = 2$. Therefore, the difference ranges from $-\frac{1}{2}$ milliseconds to $\frac{3}{2}$ milliseconds.

- (a) Just after synchronization, the maximum possible difference between the server's clock and the client's clock is $\frac{3}{2}$ milliseconds (i.e., the server's clock is ahead of the client's clock).

- (b) Between two synchronizations, the client's clock will run slow and drift behind the server's clock by:

$$30 \times 60 \times 10^{-5}\,\text{s} = 0.018\,\text{s} = 18\,\text{ms}$$

In the worst case, the client's clock would have already drifted behind the server's clock by $\frac{3}{2}$ milliseconds. During the synchronization interval, it would additionally drift behind by 18 milliseconds. Thus, the maximum possible difference between the server's clock and the client's clock just before the next synchronization is:

$$\frac{3}{2} + 18 = 19.5\,\text{milliseconds}$$

2 Problem 2

The global snapshot need not be consistent. Below, we provide a counterexample to demonstrate the inconsistency in the global snapshot.

Consider a system with three nodes: A, B, and C, connected by three bidirectional communication links: AB, BC, and AC. Assume the presence of an external reference clock used only for reference purposes, which does not interfere with the system.

Assume the following clock synchronization properties: - Nodes B and C are perfectly synchronized with the reference clock. - Node A runs 1 second slower than the reference clock. Specifically, if the reference clock shows time x, the clocks of B and C also show x, while the clock of A shows $x - 1$.(in seconds)

This assumption aligns with the problem statement, which specifies that the clocks of all three nodes are synchronized to within 1 second of each other.

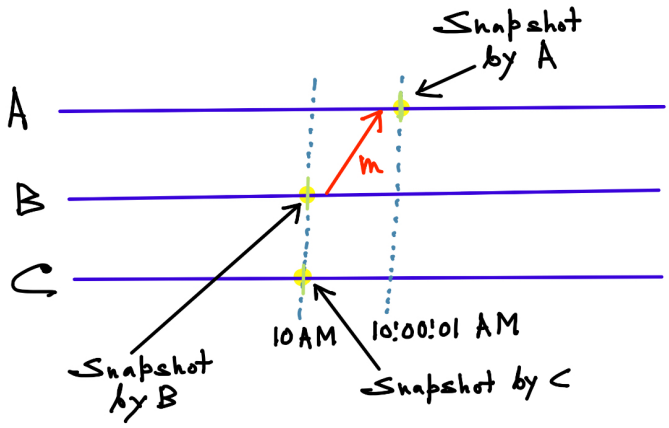

Figure 2: Timeline of Events

Let us analyze the snapshot operation at the reference time 10:00:00:

- At this time, the clock of node A displays 09:59:59.
- The clocks of nodes B and C both display 10:00:00.

Nodes B and C store their snapshots immediately at 10:00:00 (reference clock), whereas node A stores its snapshot when its clock reaches 10:00:00, which corresponds to the reference time 10:00:01.

Now, consider the following message-passing scenario:

- Node B sends a message m to node A just after the reference time 10:00:00.
- Node A receives this message just before its clock reaches 10:00:00 (at reference time 10:00:01).

In this case:

- In the snapshot recorded by B, the send event of message m will not be recorded. That is, $\texttt{send}(m) \notin LS_B$.
- In the snapshot recorded by A, the receive event of message m will be recorded. That is, $\texttt{receive}(m) \in LS_A$.

Thus, the global snapshot will contain an inconsistency:

$$\texttt{inconsistent}(LS_B, LS_A) = \{m\}$$

Since this set is not empty, the global snapshot is inconsistent.

3 Problem 3

Each node in the network stores the following information:

- **Parent:** Initialized to `null`. (The parent node of the current node.)
- **Children:** Initialized to an empty list. (The list of child nodes of the current node.)
- **Forward Count:** Initialized to `0`. (The number of neighbors to whom the current node has forwarded a message asking to be its child.)
- **Reply Count:** Initialized to `0`. (The number of neighbors that have replied to the current node's request to be its child.)
- **ACK Count:** Initialized to `0`. (The number of children c of the current node that have informed that the spanning tree construction is complete for the subtree rooted at c.)

Apart from the above information, each node also maintains a list of its neighbors. The neighbors are the nodes to which the current node is directly connected and can communicate with.

There are four types of messages in the protocol: `ASK`, `CHILD`, `REJ`, and `ACK`. Each node follows the rules below for handling each message type:

3.1 Message Handling Rules

On receiving an `ASK` message:

1. If the `Parent` is not `null`:
   - Send a `REJ` message to the sender. (The node is already connected to a parent. So, it rejects the request.)
2. Otherwise:
   - Set the `Parent` to the sender. (The node accepts the sender as its parent.)
   - Send a `CHILD` message to the sender. (The node informs the sender that it has accepted it as its child.)
   - Send `ASK` messages to all other neighbors, excluding the sender. (The node asks all other neighbors to be its children.)
   - Set `Forward Count` to the number of neighbors to whom the `ASK` message was sent. (The node sets the number of neighbors to which it has asked to be its children.)

On receiving a `REJ` message:

1. Increment the `Reply Count` by `1`. (The node received a rejection from a neighbor.)
2. If `Reply Count` = `Forward Count` and `ACK Count` = `Children.size()`: (All neighbors have replied in response to the `ASK` message, and all children have informed that the spanning tree construction is complete for the subtree rooted at them.)
    - Send an `ACK` message to the parent. (Send an acknowledgment to the parent indicating that the spanning tree construction is complete for the subtree rooted at the current node.)

On receiving a `CHILD` message:

1. Add the sender to the `Children` list. (The node accepted a child.)
2. Increment the `Reply Count` by `1`.
3. If `Reply Count` = `Forward Count` and `ACK Count` = `Children.size()`: (All neighbors have replied in response to the `ASK` message, and all children have informed that the spanning tree construction is complete for the subtree rooted at them.)
    - Send an `ACK` message to the parent. (Send an acknowledgment to the parent indicating that the spanning tree construction is complete for the subtree rooted at the current node.)

On receiving an `ACK` message:

1. Increment the `ACK Count` by `1`. (The node received an acknowledgment from a child indicating that the spanning tree construction is complete for the subtree rooted at the child.)
2. If `Reply Count` = `Forward Count` and `ACK Count` = `Children.size()`: (All neighbors have replied in response to the `ASK` message, and all children have informed that the spanning tree construction is complete for the subtree rooted at them.)
    - Send an `ACK` message to the parent. (Send an acknowledgment to the parent indicating that the spanning tree construction is complete for the subtree rooted at the current node.)

3.2 Initialization and Termination Conditions

Initialization: The root node initializes the protocol by performing the following steps:

- Send an `ASK` message to all its neighbors.
- Set its `Parent` to itself. (This ensures the root does not accept any other node as its parent.)
- Set `Forward Count` to the number of neighbors to whom the `ASK` message was sent.

Note: The root node does not add itself to its `Children` list, avoiding unnecessary self-messages.

Termination: The protocol terminates when the root node satisfies the following condition:

`Reply Count` = `Forward Count` and `ACK Count` = `Children.size()`.

This condition ensures that all neighbors have replied in response to the `ASK` message, and all children have informed that the spanning tree construction is complete for the subtree rooted at them.

4 Problem 4

The overall algorithm for the global snapshot consists of three main steps:

1. **Spanning Tree Construction**
2. **Recording Snapshots at Each Node**
3. **Propagating Snapshots to the Root Node through the Spanning Tree**

4.1 Spanning Tree Construction

For constructing the spanning tree, we use the algorithm discussed in question 3, with slight modifications to leverage the synchronous nature of the links and reduce the number of messages. Specifically:

- The `REJ` message is eliminated, and instead, a timeout mechanism is employed to detect cases where a node has already been connected to a parent.
- Further optimization, such as eliminating the `ACK` message, could be achieved if the network's diameter were known. While a bound on the diameter can be indirectly derived as $n - 1$ (given a bound on the number of nodes n), we skip this optimization as it is not explicitly stated in the problem.

The modified algorithm for constructing the spanning tree is as follows:
Each node in the network stores the following information:

- **Parent:** Initialized to `null`. (The parent node in the spanning tree.)
- **Children:** Initialized to an empty list. (The list of children in the spanning tree.)
- **ACK Count:** Initialized to `0`. (The number of acknowledgments received from children c indicating that the spanning tree construction is complete for the subtree rooted at c.)

There are three types of messages in the protocol: `ASK`, `CHILD`, and `ACK`. Each node follows the rules below for handling each message type:

4.1.1 Message Handling Rules

On receiving an `ASK` message:

1. If the `Parent` is not `null`: (The node is already connected to a parent.)
   - Ignore the message. (No action is taken. The rejection is implicit because of timeout.)
2. Otherwise:
   - Set the `Parent` to the sender. (The node accepts the sender as its parent.)
   - Send a `CHILD` message to the sender. (The sender is accepted as a child.)
   - Send a `ASK` message to all other neighbors, excluding the sender. Start a timer for the `ASK` message with duration $2 \times T$. (The node ask its neighbors to become its children. It also sets a timer with maximum round-trip time. Any node not responding within this time to the `ASK` message does not become a child.)

On receiving a `CHILD` message:

1. Add the sender to the `Children` list. (The node accepted a child.)
2. If `ACK Count` = `Children.size()` and the timer has expired: (All children have informed the node that the spanning tree construction is complete for their subtrees and the timer has expired)
    - Send an `ACK` message to the parent. (All children have acknowledged.)

On receiving an `ACK` message:

1. Increment the `ACK Count` by `1`. (The node received an acknowledgment from a child indicating that the spanning tree construction is complete for the subtree rooted at the child.)
2. If `ACK Count` = `Children.size()` and the timer has expired: (All children have acknowledged and the timer has expired)
    - Send an `ACK` message to the parent. (Send an acknowledgment to the parent indicating that the spanning tree construction is complete for the subtree rooted at the node.)

4.1.2 Timeout Handling

If the timer for the `ASK` message expires, the node proceeds as follows: (Within, this time, the node expects a `CHILD` message from the neighbor which has accepted the node as a parent.)

1. If `ACK Count` = `Children.size()`: (All children have acknowledged that the spanning tree construction is complete for their subtrees.)
    - Send an `ACK` message to the parent. (Send an acknowledgment to the parent indicating that the spanning tree construction is complete for the subtree rooted at the node.)

4.1.3 Initialization and Termination Conditions

A node i begins the spanning tree construction by:

- Sending an `ASK` message to all its neighbors. (The node asks its neighbors to become its children.)
- Setting itself as the root node for its local state. (The node is the root of the spanning tree.)
- Starting a timer for the `ASK` message with a duration of $2 \times T$. (The node sets a timer with maximum round-trip time. Any node not responding within this time to the `ASK` message does not become a child.)

The spanning tree construction is considered complete at node i when the following conditions are met:

- The `ACK Count` equals the size of its `Children` list. (All children have acknowledged that the spanning tree construction is complete for their subtrees.)
- The timer has expired. (The timer for the `ASK` message has expired.)

4.2 Recording Snapshots at Each Node

To record snapshots at each node, we modify the Chandy-Lamport algorithm to handle non-FIFO channels. To do this, we introduce a delay after sending the marker message to ensure that all messages sent before the marker message are received.

Each node in the network maintains the following information:

- **Local State:** Stores the state of the node, initially set to `null`.
- **Channel States:** For each neighbor, stores the state of the channel directed to the node. Specifically, for each neighbor j, the node maintains the state of the channel from j to itself. All channel states are initialized to `null`.

When node x receives a marker message from a neighbor y along the channel $y \to x$, the following steps are executed:

1. **If the node has not recorded its local state:**
   - Upon receiving the marker message, wait for T time units to detect any pending messages. If any pending messages are detected, process them and accordingly update the local state. (This step ensures that all messages sent before the marker message are received and processed.)
   - Record the local state of the node as the state at the end of the waiting period. Record the state of the channel $y \to x$ as empty. (The channel state is recorded as empty since the marker message is the last message sent along the channel.)
   - Send marker messages to all neighbors. (The node sends marker messages to all neighbors asking them to record their local states.)
   - After sending the marker messages, delay sending any further messages for $2 \times T$ time units, i.e., wait for $2 \times T$ time units before placing any new messages in the channels. (This delay ensures that the receiving node can differentiate between messages sent before and after the marker message.)
2. **If the node has already recorded its local state:**
   - Let M_1 denote the set of messages received before the marker message from y and after the local state was recorded by x. (These messages are received after the local state was recorded but before the marker message was received.)
   - Upon receiving the marker message, the node waits for T time units to detect any pending messages. Let M_2 represent the set of messages received during this waiting period from y. (These messages are received after the marker message was received and during the waiting period.)
   - Record the state of the channel $y \to x$ as $M_1 \cup M_2$, the union of the two sets of messages.

In order to see how we use the timeouts value to differentiate between messages sent before and after the marker message, consider the following scenario:(All the time values are with respect to a global reference clock.)

A node x sends a marker message M to a neighbor y at time $t_M = t_1$. The message is received by y at time $t_{rec}^{(M)} \in [t_1, t_1 + T]$. Since our algorithm does not restrict sending messages before the marker, assume that x sent another message msg to y at time $t_{msg} = t_1 - \epsilon$, where ϵ is a small positive value. This message is received by y at time $t_{rec}^{(msg)} \in [t_1 - \epsilon, t_1 + T - \epsilon]$. In the worst case, it is possible that $t_{rec}^{(M)} = t_1$ while $t_{rec}^{(msg)} = t_1 + T - \epsilon$. As a result, after receiving the marker message, y must wait for T time units to detect any pending messages.

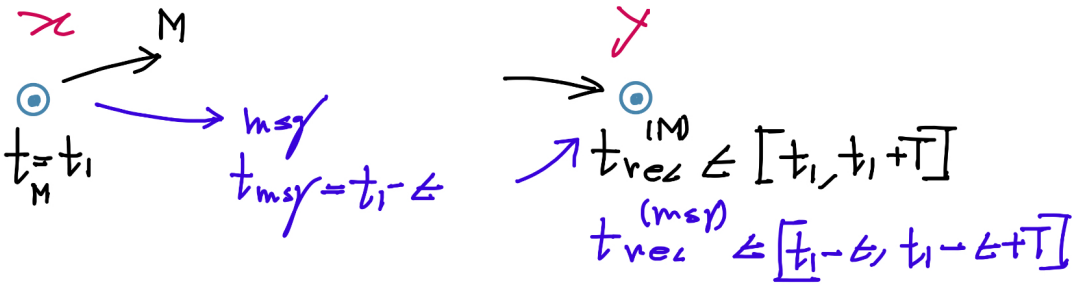

Figure 3: Illustration of the timeout mechanism for recording snapshots

Since y is unaware of the delay for the marker message, it must wait T time units irrespective of the marker's delay to detect any pending messages. If the marker message is delayed by T time units, y will still have to wait for T time units to detect any pending messages after the marker's receive. In the worst case, y may have to wait until $t_1 + 2T$ to identify any pending messages. After this period, any message received by y is considered to have been sent after the marker message. This justifies our algorithm's requirement that x must wait for $2 \times T$ time units before placing any new messages in the channel.

4.2.1 Initialization and Termination Conditions

The global snapshot algorithm is initiated by a node i through the following steps:

- The node first records its local state.
- It then sends marker messages to all its neighbors.
- After sending the marker messages, the node waits for $2 \times T$ time units before placing any new messages in the channel.

A local state recording of a node is deemed complete when both the node's state and all its incoming channel states have been recorded. This condition will eventually be satisfied for all nodes in the network, as the graph is connected.

At this stage, we will not detect the completion of the global snapshot but we note here that each node can independently detect the completion of its local snapshot recording. We will now use the spanning tree constructed in the previous step to propagate the snapshots to the root node.

4.3 Propagating Snapshots to the Root Node through the Spanning Tree

As we noted, every node can independently detect the completion of its local snapshot recording. The next step is to propagate the snapshots to the root node through the spanning tree. The idea is to send the snapshots from the leaf nodes to the root node, merging the snapshots as they propagate upwards. A node can send the snapshot to its parent only when it has received snapshots from all its children and has recorded its local state. The message format can be defined as follows where we assume that the snapshot is a JSON object: (The format of the snapshot can be modified based on the requirements of the application. The only requirement is that the receiver and sender agree on the format.)

```
{
    "node_id": <node_id>,
    "local_state": <local_state>,
    "channel_states": {
```

```
        "link_1": <state>,
        "link_2": <state>,
        ...
    }
}
```

Each node j in the network maintains the following information:

- **Snapshot State:** Initially set to `null`.
- **Received Snapshots:** A list of snapshots received from all children, initially set to an empty list.

When node j receives a snapshot from a child i or detects that its local state and channel states have been recorded, it follows the steps below:

1. If the snapshot is received from a child i, add it to the `Received Snapshots` list. (The snapshot received from child i is added to the list of received snapshots.)
2. If the size of the `Received Snapshots` list equals the number of children of node j, and the local state and channel states of node j have been recorded: (All children have sent snapshots from their subtrees, and the local state and channel states of node j have been recorded.)
   - Merge the local state of node j with the channel states of node j, along with the snapshots received from all children. (The local state and channel states of node j are merged with the snapshots received from all children.)
   - Send the merged snapshot to the parent of node j. (The merged snapshot is sent to the parent of node j.)

The node i that initiated the global snapshot can detect its completion once it receives a snapshot from all its children in the spanning tree. The snapshot received by the root node is the global snapshot of the network.

4.4 Message Complexity

Let n be the number of nodes in the network and m be the number of channels. The message complexity of the global snapshot algorithm can be analyzed as follows:

1. **Spanning Tree Construction:** During the spanning tree construction, each node sends at most one `ASK` message to each of its neighbors. This results in atmost $2 \times m$ messages. Additionally, the `CHILD` and `ACK` messages are sent in response from child to parent, each being sent $n - 1$ times. Therefore, the total number of messages sent during spanning tree construction is:
$$2 \times (m + n - 1)$$
2. **Recording Snapshots at Each Node:** The marker message is sent along each channel at most twice, so the number of messages sent during snapshot recording is at most:
$$2 \times m$$
3. **Propagating Snapshots to the Root Node:** The number of messages sent during the propagation of snapshots to the root node is $n - 1$.

Summing the message complexities of these three steps, the total message complexity of the global snapshot algorithm is:

$$2 \times (m + n - 1) + 2 \times m + (n - 1) = 4 \times m + 3 \times n - 3 = O(m + n)$$

5 Problem 5

In this problem, we modify the Ricart-Agrawala algorithm to design a permission-based distributed mutual exclusion algorithm that requires zero messages in the best case.

5.1 Best-Case Scenario

The best case occurs when a single process repeatedly requests access to the critical section while no other process makes a request. The key idea is that once a process p_i has received a `REPLY` message from another process p_j, it does not need to send a `REQUEST` message to p_j again unless p_i has sent a `REPLY` message to p_j in response to a `REQUEST`. Consequently, if only one process requests the critical section, it initially needs to obtain permission from all other processes. However, subsequent requests will not require any additional messages since permission has already been acquired.

Additionally, if a subset of processes requests the critical section, only the processes within the subset need to communicate with each other once all other processes have granted permission at least once to each process in the subset.

5.2 Worst-Case Scenario

The worst case arises when all processes frequently request access to the critical section. In this scenario, any process p_i cannot hold permission indefinitely, as it must send `REPLY` messages to other processes upon receiving their `REQUEST` messages. As a result, each new request necessitates sending a `REQUEST` message to all other processes and waiting for their `REPLY` messages. This scenario requires $2(N-1)$ messages per request in the worst case.

5.3 Process State Management

Each node maintains a logical clock updated following the Lamport clock rules, along with the following data structures:

- **Request Queue:** A queue of requests for the critical section, ordered by logical timestamps.
- **Permission Array:** A boolean array of size N, where the i-th element is `true` if process p_i has granted permission to the current process and `false` otherwise. Initially, all elements are set to `false` except for the current process's index, which is `true`.
- **My Requests Queue:** A list of requests sent by the current process, ordered by logical timestamps.

5.4 Message Types

The system uses two types of messages:

- `REQUEST(i, ts)`: Sent by process p_i to request access to the critical section, where ts is the logical timestamp.
- `REPLY(i)`: Sent by process p_i to grant permission to another process.

5.5 Requesting the Critical Section

To request critical section access, process p_i follows these steps:

1. If the request queue is empty and all elements of the permission array are `true`, enter the critical section (no communication needed).
2. Otherwise, send a `REQUEST(i, ts)` message to all processes for which the corresponding element of the permission array is `false`. Add the request to both the `Request Queue` and `My Requests Queue`. It will eventually enter the critical section when all elements of the permission array are `true`.

5.6 Handling a `REQUEST` Message

Upon receiving a `REQUEST(i, ts)` message from process p_i, process p_j executes the following steps:

1. If p_j is not interested in the critical section (i.e., `My Requests Queue` is empty and it is not currently in the critical section), it:
   - Sends a `REPLY(j)` message to p_i.
   - Sets the i-th element of the permission array to `false`.
2. If p_j is currently in the critical section, it adds the request to the `Request Queue` for later processing.
3. If p_j is interested in the critical section (but not currently executing), it compares the timestamp of the received request with the request at the front of `My Requests Queue`:
   - If the received request has a smaller timestamp, p_j does the following:
     - Sends a `REPLY(j)` message to p_i.
     - Sets the i-th element of the permission array to `false`.
     - If the i-th element of the permission array was previously `true`, it also sends a `REQUEST(j, ts_front)` message to p_i to inform p_i of its own request, where `ts_front` is the timestamp of the request at the front of `My Requests Queue`.

5.7 Handling a `REPLY` Message

Upon receiving a `REPLY(j)` message from process p_j, process p_i:

1. Sets the j-th element of the permission array to `true`.
2. If a request from p_i is present in the `My Requests Queue`, it checks if all elements of the permission array are `true`. If so, it enters the critical section and removes its request from both the `Request Queue` and `My Requests Queue`.
3. Otherwise, it waits for more `REPLY` messages.

5.8 Releasing the Critical Section

To release the critical section, process p_i:

1. Sends a `REPLY(i)` message to all processes in the `Request Queue`.
2. Sets the corresponding elements of the permission array to `false`.
3. Removes the requests from the `Request Queue`.

6 Problem 6

To construct a BFS spanning tree, we use an algorithm similar to the one described in Problem 3, but with modified message types and handling rules to ensure the BFS property rather than an arbitrary spanning tree.

6.1 Node Data Structure

Each node in the network maintains the following information:

- **Parent:** Initially set to `null`. (The node's parent in the BFS spanning tree.)
- **Children:** Initially an empty list. (The node's children in the BFS spanning tree.)
- **Forward Count:** Initially `0`. (Number of neighbors to which the node sent an `ASK/JOIN` message.)
- **Reply Count:** Initially `0`. (Number of replies received by the node.)
- **Level:** Initially `0`. (The node's level in the BFS spanning tree.)
- **New Node Found:** Initially `false`. (Flag indicating if a new node was added to the subtree rooted at the node during a phase.)

6.2 Message Types

The protocol utilizes the following message types:

- `JOIN(LEVEL)`: Sent by the root to initiate the expansion to level `LEVEL`.
- `ASK(LEVEL)`: Sent by a node at level `LEVEL-1` to its neighbors asking them to join level `LEVEL`.
- `ACK`: Acknowledgment message sent by a node to its parent when it agrees to join the tree.
- `NACK`: Negative acknowledgment message sent by a node when it is already part of the tree.
- `DONE(LEVEL,NEW_NODE_FOUND)`: Sent by a node to its parent to indicate completion of level `LEVEL` in its subtree. The `NEW_NODE_FOUND` field is set to `true` if at least one new node was added to the tree.

6.3 Algorithm Description

Let's say at any phase i, the BFS spanning tree has been constructed up to level i, and termination has been detected by the root for this level. We now describe the process of expanding the tree to level $i + 1$.

6.3.1 Step 1: Sending JOIN Messages

The root node initiates the process by sending a `JOIN(i+1)` message to all its `Children`. Any node u at level j ($j < i$) forwards this message to all its `Children`.

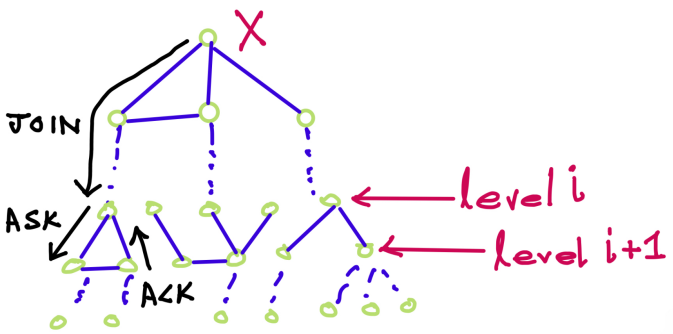

Figure 4: Expansion of the BFS Spanning Tree to Level $i+1$.

6.3.2 Step 2: ASK and ACK/NACK Exchange

When a node at level i receives `JOIN(i+1)`, it sends an `ASK(i+1)` message to all directly connected nodes except its `Parent`. It also sets:

- `Forward Count` to the number of nodes to which it sent `ASK`.
- `Reply Count` to `0`.

Nodes that receive `ASK(i+1)` respond as follows:

- If already part of the spanning tree, they send a `NACK`.
- If not part of the tree and receiving `ASK` for the first time:
  - They send an `ACK`.
  - They mark the sender as their `Parent` and set their `Level` to $i+1$.

Nodes at level i collect `ACK` and `NACK` responses, updating:

- **Children:** Nodes from which they receive `ACK`.
- **Reply Count:** Increments for every received `ACK` or `NACK`.

6.3.3 Step 3: Propagating DONE Messages

Once a node at level i has received replies from all nodes it sent `ASK` to (`Forward Count` = `Reply Count`), it sends a `DONE(LEVEL,NEW_NODE_FOUND)` message to its `Parent`. Here, `NEW_NODE_FOUND` is:

- `true` if at least one `ACK` was received.
- `false` otherwise.

Nodes at top levels j ($j < i$) wait for `DONE` messages from all their `Children` before forwarding the `DONE` message to their own `Parent`. The `NEW_NODE_FOUND` field is set to `true` if at least one child reported `NEW_NODE_FOUND = true`.

6.3.4 Step 4: Root Detection of Termination

The root node X receives `DONE` messages from all its `Children`. If any contain `NEW_NODE_FOUND = true`, it proceeds to phase $i+2$. Otherwise, the algorithm terminates, completing the BFS spanning tree construction.

This process can be repeated until the root detects that no new nodes were added to the tree in the last phase, signaling the completion of the BFS spanning tree.

6.4 Message Handling Rules

6.4.1 Handling a `JOIN` Message

When a node receives a `JOIN(i)` message:

- If the node is at level $i-1$, it sends an `ASK(i)` message to all its neighbors except the `Parent` and sets `New Node Found` to `false`. It also set the `Forward Count` to the number of neighbors to which it sent the message and `Reply Count` to `0`. (Nodes that are leaves in the current treen send `ASK` messages to all their neighbors asking them to join level i.)
- If the node is at level $j < i-1$, it forwards the message to all its `Children` and sets `Reply Count` to `0`. (Nodes that are not leaves in the current tree forward the `JOIN` message to their children.)

6.4.2 Handling an `ASK` Message

Upon receiving an `ASK(i)` message:

- If the `Parent` field is not `null`, reply with a `NACK` message. (The node is already part of the BFS spanning tree.)
- Otherwise, reply with an `ACK` message, set the `Parent` field to the sender, and `Level` to i. (The node joins the BFS spanning tree at level i.)

6.4.3 Handling an `ACK` Message

Upon receiving an `ACK` message:

- Add the sender to the `Children` list. (The sender is now a child of the node.)
- Increment the `Reply Count`. (The node has received a reply.)
- If `Reply Count` = `Forward Count`, send a `DONE(Level,true)` message to the `Parent`. (All replies have been received from the neighbors.)

6.4.4 Handling a `NACK` Message

Upon receiving a `NACK` message:

- Increment the `Reply Count`. (The node has received a reply.)
- If `Reply Count` = `Forward Count`, check if the `Children` list is empty. If so, send a `DONE(level,false)` message to the `Parent`. Otherwise, send a `DONE(level,true)` message to the `Parent`. (All replies have been received from the neighbors. If no new nodes were explored, the `NEW_NODE_FOUND` field is set to `false`.)

6.4.5 Handling a `DONE` Message

Upon receiving a `DONE(i,NEW_NODE_FOUND)` message:

- Increment the `Reply Count`. (The node has received a `DONE` message from a child indicating BFS tree construction up to level i in the child's subtree.)
- Set `New Node Found` to `New Node Found` $\vee$ `NEW_NODE_FOUND`. (Update the `New Node Found` flag based on the child's message.)
- If `Reply Count` = `Forward Count`, send a `DONE(i,New Node Found)` message to the `Parent`. (All replies have been received from the children. Send a message to the parent indicating completion of level i in the subtree.)

6.5 Initialization and Termination

The root node X initiates the BFS spanning tree construction by sending a `JOIN(1)` message to all its `Children`. It also sets itself as the root of the tree by setting its `Parent` field to X and `Level` to 0.

For each phase i, the root sends a `JOIN(i)` message to all its `Children` and waits for `DONE(i, NEW_NODE_FOUND)` messages from all of them. If at least one `DONE` message contains `NEW_NODE_FOUND = true`, the process continues to phase $i + 1$. Otherwise, the algorithm terminates, signaling the completion of the BFS spanning tree.

6.6 Message Complexity

Let N represent the number of nodes in the network, M the number of edges, and D the diameter.

The number of rounds is bounded by the network's diameter. For each edge (u, v), the `ASK` and `ACK/NACK` messages are exchanged at most twice, resulting in a maximum of 4 messages per edge, leading to $4 \times M$ messages in total.

In each round, `JOIN` and `DONE` messages are exchanged across the tree edges, with a total of at most $2 \times (N - 1)$ messages per round. Hence, the overall message complexity is:

$$O(N \times D + M).$$

7 Problem 7

This section describes the protocol for constructing a Depth-First Search (DFS) spanning tree in a network. The protocol uses a DFS traversal approach, with tie-breaking based on root node IDs in cases of concurrent initiation. Specifically, if multiple nodes initiate the process simultaneously, a node joins the tree rooted at the highest ID node in case of a conflict. The protocol employs the following message types and rules.

7.1 Node Data Structure

Each node in the network maintains the following data:

- **Parent**: Initially set to `null`. (Represents the parent node in the DFS tree.)
- **MyRoot**: Initially set to `null`. (Represents the root node of the DFS tree.)
- **Children**: An initially empty list to store child nodes. (These are nodes that have accepted the tree edge from the current node.)
- **Neighbors**: A list of adjacent nodes. (Represents the set of nodes directly connected to the current node.)
- **Unknown**: Initially set to the list of neighbors. (Represents nodes whose status is unknown i.e not yet visited in the DFS traversal.)

7.2 Message Types

The protocol uses the following message types:

- `QUERY(id)`: Sent by a node to its neighbors to invite them to join the DFS tree rooted at the node with ID `id`.

- `ACCEPT(id)`: Sent by a node to its parent to confirm the acceptance of the tree edge. The `id` field contains the root node's ID.
- `REJECT(id)`: Sent by a node to its parent to decline the tree edge. The `id` field contains the root node's ID.

The `QUERY` messages facilitate the continuation of the DFS traversal from the current node. The `REJECT` message serves to detect cycles during the tree construction process. The `ACCEPT` message confirms the acceptance of a tree edge and notifies the parent node that all its neighbors have been visited.

Additionally, if a node receives a `QUERY` message from a node with a higher root ID, it updates its tree membership to join the tree rooted at the higher ID node.

7.3 Message Handling Rules

7.3.1 Handling a `QUERY` Message

When a node i receives a `QUERY(id)` message from a neighbor j, it performs the following steps:

1. If `MyRoot` is `null` or `id` is greater than `MyRoot`: (The current node should join the tree rooted at `id`.)
   - Set `MyRoot` to `id`. (Update the root node.)
   - Set `Parent` to j. (Set the parent node.)
   - Set `Unknown` to the list of neighbors excluding j. (Initialize the unknown list.)
   - If `Unknown` is empty, send an `ACCEPT(id)` message to j. (No unknown neighbors remain. So, accept the tree edge and notify the parent that DFS in the subtree is complete.)
   - Otherwise, send a `QUERY(id)` message to the first node in `Unknown` and remove it from `Unknown`. (Continue DFS traversal by querying the next unknown neighbor.)
2. If `id` equals `MyRoot`, send a `REJECT(id)` message to j. (Reject the tree edge if the root ID matches the current root. The node is already part of the same tree.)
3. If `id` is less than `MyRoot`, ignore the message. (Reject the tree edge if the root ID is lower than the current root; the node is already part of a tree rooted at a higher ID.)

7.3.2 Handling an `ACCEPT` Message

When a node i receives an `ACCEPT(id)` message from a neighbor j, it performs the following steps:

1. If `MyRoot` is `id`:
   - Add j to the `Children` list. (Update the list of children.)
   - If `Unknown` is empty:
     - If `Parent` is i, designate i as the root and terminate. (The current node is the root of the tree and all neighbors have been visited.)
     - Otherwise, send an `ACCEPT(id)` message to `Parent`. (Notify the parent that all neighbors have been visited in the subtree.)
   - Otherwise, send a `QUERY(id)` message to the first node in `Unknown` and remove it from `Unknown`. (Continue DFS traversal by querying the next unknown neighbor.)
2. If `MyRoot` is not `id`, ignore the message. (This node has changed tree membership after sending the `QUERY` message.)

7.3.3 Handling a `REJECT` Message

When a node i receives a `REJECT(id)` message from a child j, it performs the following steps:

1. If `MyRoot` is `id`:
   - If `Unknown` is empty:
     - If `Parent` is i, designate i as the root and terminate. (The current node is the root of the tree and all neighbors have been visited.)
     - Otherwise, send an `ACCEPT(id)` message to `Parent`. (Notify the parent that all neighbors have been visited in the subtree.)
   - Otherwise, send a `QUERY(id)` message to the first node in `Unknown` and remove it from `Unknown`. (Continue DFS traversal by querying the next unknown neighbor.)
2. If `MyRoot` is not `id`, ignore the message. (The node has changed tree membership after sending the `QUERY` message.)

7.4 Initialization and Termination

To initiate the tree construction, a node i first checks if `Parent` is `null`. If so, it sends a `QUERY(i)` message to itself. (This is an internal action and does not involve actual communication.)

The tree construction terminates when the root node confirms that all its neighbors have been visited. This occurs when the `Unknown` list becomes empty. At this point, the root node designates itself as the root of the tree and finalizes the tree construction. To disseminate this information, the root node can propagate a completion signal throughout the spanning tree.

8 Problem 8

For this problem, we will utilize the concept that a request made by a node will be forwarded along the path directed by the `to_token` value. Each node will make requests to the its parent, either for itself or on behalf of its subtree. The root is the node that holds the token. To allow access to the critical section, the token will be forwarded to the node with the earliest request in the queue.

Additionally, depending on the forwarding of the token, the edges of the spanning tree will be reversed to point to the token holder. A node that holds the token does not enter the critical section if it has a request in `request_q` with an earlier timestamp from its subtree. In such a case, the node forwards the token to the appropriate node and sends a new request to the current root.

Each node maintains the following queue and variables:

- `request_q`: A queue storing requests for the critical section, ordered by timestamps, ensuring a total ordering (e.g., using Lamport's logical clock).
- `is_requested`: A boolean variable set to `true` if the node has requested access to the critical section for itself or its subtree to is parent. (To avoid multiple requests to the parent.)
- `has_token`: A boolean variable indicating if the node holds the token. This can be omitted since the node with the token is the one where `to_token` equals itself.

8.1 Message Types

The algorithm uses two types of messages:

- `REQUEST(ts,i)`: Sent to `to_token` to request access to the critical section for itself or its subtree with timestamp `ts` and node ID `i`.
- `TOKEN`: Sent to another node to pass the token.

8.2 Handling `REQUEST` Messages

When a node i receives a `REQUEST(ts,j)` message from its child j:

1. If i is the root:
   - If i is not executing the critical section and its `request_q` is empty, it sends a `TOKEN` message to j and sets `to_token` to j. (Pass the token to the only requester and reverse the edge.)
   - If i is executing the critical section, it adds j's request to `request_q`. (Put the request in the queue for later processing.)
2. Otherwise (if i is not the root):
   - i adds j's request to its `request_q`. (Put the request in the queue)
   - If `is_requested` is `false`, i sends a `REQUEST(ts,i)` message to its `to_token` and sets `is_requested` to `true`. (Make a request to the parent if no request has been made yet.)

Every node sends a `REQUEST` message to its parent when requesting access to the critical section, either for itself or its subtree. This mechanism helps merge multiple requests from the same subtree into a single request.

8.3 Handling `TOKEN` Messages

When a node i receives a `TOKEN` message from node j:

1. It removes the first request from `request_q`, if it is from i itself, it sets `to_token` to i, and enters the critical section. (If the first request is from itself, enter the critical section.) For releasing the token, follow the steps mentioned in Section 8.5. Otherwise, if the first request is from a child k, send a `TOKEN` message to k and set `to_token` to k. (Forward the token to the first requester in the queue.)
2. Now, if `request_q` is empty, it sets `is_requested` to `false`. (No more requests are pending.)
3. Otherwise, it sends a `REQUEST` message to its `to_token` (To whom the token was forwarded) and sets `is_requested` to `true`. (If any requests are pending, submit a new request to the new parent.)

8.4 Entering the Critical Section

To enter the critical section, a node:

1. Checks if it holds the token and the `request_q` is empty.
   - If both conditions are met, it enters the critical section.
   - Otherwise, it adds its request to `request_q`. If the flag `is_requested` is `false`, it sends a `REQUEST` message to its `to_token` and sets `is_requested` to `true`. It will enter the critical section when the token is received and its own request is at the front of the `request_q`.

8.5 Releasing the Critical Section

To release the critical section:

1. If `request_q` is non-empty: (Pending requests are present)
    - The node sends a `TOKEN` message to the first node in `request_q` and updates `to_token` to that corresponding node and removes the first request from `request_q`. (Forward the token to the first requester in the queue.)
2. After the previous step, if `request_q` is still non-empty: (More requests are pending)
    - The node sends a `REQUEST` message to its new parent and sets `is_requested` to `true`. (Make a request to the parent if any requests are still pending.)

Note: The algorithm assumes that each node processes different types of messages in a FIFO order, ensuring that no two messages are interleaved during processing. For instance, if a process is currently handling a `TOKEN` message and receives a `REQUEST` message simultaneously, it will queue the `REQUEST` message and process it only after completing the `TOKEN` message.

8.6 Message Complexity

In the best-case scenario, a node may already possess the token and can enter the critical section without exchanging any messages. However, in the general case, the average distance between any two nodes in a tree is $O(\log n)$, where n represents the number of nodes. For each request, the request must first propagate to the root, and then the token must travel back to the requester. Consequently, the average message complexity for each request is $O(\log n)$.