Now 9 gets back its own value and knows it is the leader. It can now claim to the server that it is the leader.

---

5 extra sheets

(1) Election

Franklin Algorithm for unidirectional ring:

1. Initially all processes have state = white

2. When state = white
   - a) Send my value (i) to my neighbour
   - b) When received a value \( x \), if \( x > i \) (my value) eliminate myself from being candidate, i.e., State = black.
   - c) If received value = i (my value), then I am the leader.

3. When state = black, forward incoming message to my neighbour.

Initially:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

All states are white. Each process sends its value to its neighbour.
In this case, for the hash, it is a (eq. #1), in each (eq. #2).

Some processes become black as shown. Only white processes send their values now. Black nodes just forward.

Some more nodes become black (eq. #3), 7 sends its value to node #0, but #0 forwards it to #2, which becomes black. Only nodes #9 and #3 forward send their values.

The value #9 goes around and reaches #3, which becomes black. But now #9 still does not know it is a leader. So we need another iteration.
Now 9 gets back its own value and knows it is the leader. It can now proclaim to the others that it is the leader.

3. The worst case for the algorithm is when the nodes are arranged in ascending order (increasing order):

In this case, we need \( n \) rounds where \( n \) = number of processes. (We need \( n-1 \) rounds for all except the highest process to be black. The last round is required for the highest process to get back its value and learn that it is the leader.)

In each round, there are \( n \) messages sent. Total message = \( n^2 \)

So parallel time complexity = \( O(n^2) \)

(The worst case is when the processes are arranged in descending order (decreasing order):)
In this case we need 2 rounds for the highest process to learn that it is a leader.

In each round we have r messages (r = number of processes). Total messages = 2r.
So parallel time time complexity = \( O(r) \) (Since in one iteration, there is only 1 write per process left. We need r-1 more iterations for that process to know it is leader)

4. Both Chang & Roberts' algorithm and Franklin algorithm (for undirectional ring) work on the undirectional token ring.

In Chang & Roberts' algorithm, we get parallel time complexity = \( O(r) \)

The algorithms are similar to each other, just implemented differently. In Chang and Roberts, the highest value propagates in the full ring. In Franklin's algorithm, we use a similar idea, but implemented differently.

Parallel time complexity of both algorithms are the same, \( O(n) \). So we cannot choose one over the other.
In other words, on each link there is a probe + echo or there are 2 probes. So when the value \( x \) reaches the initiator, the sum is the number of links.

2. **Probe Echo**

For each parent-child relation, we have \( x = 1 \). That is, for every child, \( x = 1 \).

For every probe received from other than a parent, we have \( x = x + 0.5 \).

For every echo received from a child, we have \( x = x + \text{child} \cdot x \).

Now, for leaves, \( x = 1 \).

In this example, values of \( x \) at the nodes:

\[
\begin{align*}
E &= 1 \\
F &= 1 \\
D &= 1 \cdot (\text{from Parent B}) + E \cdot x + F \cdot x + 0.5 (\text{from C}) \\
&= 1 + 1 + 1 + 0.5 = 3.5 \\
B &= 1 (\text{from parent A}) + 0.5 (\text{from C}) + D \cdot x \\
&= 1 + 0.5 + 3.5 \\
&= 5 \\
C &= 1 (\text{from parent A}) + 0.5 (\text{from B}) + 0.5 (\text{from D}) \\
&= 2 \\
\end{align*}
\]

\[ A \cdot x = 5 + 2 = 7 \quad \text{OK, perfect} \]

**Note:**

\( x \) actually counts the number of edges in the graph. This is evident by the fact that there is either two probes of 0.5 each in each direction, or there is a parent-child relationship in an edge. If there is a parent-child relationship, the \( x \) counts the number of children (assuming no peer links). The algorithm counts the no. of edges.

[Peer links = links between nodes that are not parent-child]
Group communication

This algorithm does not ensure causal ordering. It only ensures that all processes get messages in some order, this ordering may or may not be causal ordering.

In the above case, message 1 is delivered before message 0 simply because the sequencers got the message 1 before message 0. But \( m_0 \rightarrow m_1 \). Thus causal ordering is not respected.

However, in this case causal ordering does happen (but it is only a coincidence and may not always happen).