CO21BTECH11002
Aayush Kumar
Theory Assignment 1

1)

Each process maintains 4 arrays of size N each :-
int msgReceived[N];
int sendCount[N];
bool replyValue[N];
int replyCount[N];

msgReceived is initialized with -1, sendCount is initialized with 0, replyValue is initialized with true.

Suppose that node x and y are neighbors, and an election message is initiated by node i and it has reached node x.

msgReceived[i] at y denotes whether y has received the election msg that was started by node i and which node was the sender. If msgReceived[i] is -1 then x will send the election message to y and set msgReceived[i] at y to be x.

sendCount[i] at x denotes to how many neighboring nodes x has sent the message.

replyValue[i] at x denotes whether node i can be a leader based on the replies received by x.

replyCount[i] at x denotes from how many nodes, x has received a reply for the election message of i.

The modified algorithm goes as follows:

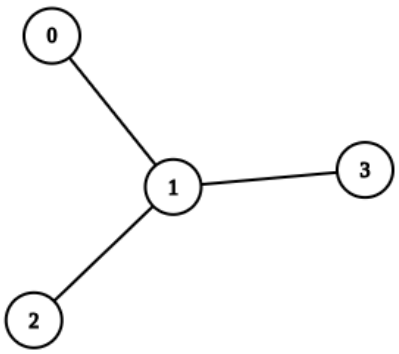
define failed : Boolean {set if the failure of the leader is detected}
    L: process {identifies the leader}
    m: message {election | leader | reply}
    state : idle | wait for reply | wait for leader

initially $\forall i \in V$ state = idle, failed = false

```
while( failure of L(i) detected && failed ) {
        int i = current process id;
        // start the election process
        ∀j such that there exists an edge between i and j {
                if(!msgReceived[ j ]) {
                        send an election message to j;
                        // denotes that i is a parent of j.
                        msgReceived[ j ] = i;
                        // i has sent a message, so increase the count.
                        sendCount[ i ]++;
                }
        }
        // message is received
        int p = process id of process that this message corresponds to
        if(message is for “election”) {
                // propagate the message forward
                ∀j such that there exists an edge between i and j {
                        if(!msgReceived[ j ]) {
                                send an election message to j;
                                // denotes that i is a parent of j.
                                msgReceived[ j ] = i;
                                // i has sent a message, so increase the count.
                                sendCount[ i ]++;
                        }
                }
        }
        if(message is a “reply” || sendCount[p]==0) {
                // received a reply from a node
                if(sendCount[p] != 0) replyCount[ p ]++;
                // if reply value from even one node is false, set it to false
                if(sendCount[p] != 0) replyValue[ p ] &= reply value received;
                // reply received from all the nodes
                if(replyCount[ p ] == sendCount[ p ] ) {
                        // if p is smaller than current process i, set replyValue to false
                        replyValue[ p ] &= (p > i);
                        // send replyValue back to parent
                        send replyValue[ p ] to msgReceived[ p ];
                }
        }
```

```
        if(status == idle && timeout) {
                failed = true;
        }
        if(replyCount[ i ] == sendCount[ i ] && replyValue[ i ]) {
                L(i) = i;
                send leader message to all;
        }
}
```

Consider the following example:



For the election message that is initiated by 2:
2 sends “election” to 1 and sets msgReceived[2] = 2 for node 1.
sendCount[2] = 1 at node 2.
Now 1 sends this “election” to 0 and 3 and sets msgReceived[2] = 1 at both 0 and 3.
sendCount[2] = 2 at node 1.
For 0, sendCount[2] is 0. So it calculates the reply value as replyValue[2] &= (2>0). So, replyValue[2] remains 1. 0 now sends this value back to 1. For 3, sendCount[2] is 0. So it calculates the reply value as replyValue[2] &= (2>3). So, replyValue[2] becomes 0. 3 now sends this value back to 1.
1 receives 0 from 0 and 1 from 3. On doing replyValue[2] &= reply value received, it becomes 0 at node 1. 1 finally calculates the replyValue[2] as replyValue[2] &= (2>1), but it remains 0.

Now, 1 sends this value back to msgReceived[2] which at node 1 is 2 itself.
2 finally calculates the replyValue[2] as replyValue[2] &= reply value received. Thus at node 2, replyValue[2] = 0. Therefore 2, cannot elect itself as leader.

Similarly, for the election message started by 3, since it has the highest ID, the final value of replyValue[3] at node 3 will remain to be 1. Thus 3 will elect itself as the leader and will send the leader message forward.

For each election message, the message is forwarded to further nodes in a dfs manner, thus each node takes O(N) complexity for determining whether it can elect itself as the leader. And there are N nodes, so each round takes O(N^2) complexity. In case the leader fails, there will be N rounds at worst, so final complexity is O(N^3).

2)
For an asynchronous system, each process can maintain 2 arrays of size D (diameter of graph).

int maxID[D];
int receiveCount[D];

Where maxID[i] at node x represents the leader for x at the i'th round. Initially, maxID[0] = x for all processes $x \in V$.

receiveCount[i] at x represents the number of messages received by node x for i'th round.

The messages passed contain two values, {round number, leader id}.

Algorithm:

define r: integer {round number}, L: process id {identifies the leader}, N - neighbors of i,
Δ - degree of node i
initially r = 1
send { r, maxID[0] } to each $j \in N(i)$;
while (r < D) {
receive message from each $j \in N(i)$;
int roundId = message.roundID, leaderID = message.leaderID;
maxID[roundID] = max(maxID[roundID], leaderID);
receiveCount[roundID]++;

```
        // received message for r'th round from all the neighbors
        if( receiveCount[ r ] == Δ) {
                r++;
                send { r, maxID[ r ] } to each j ∈ N(i);
        }
}

// after all rounds have completed
L(i) = maxID[ r ];
```

Each node waits until it has received messages from all its neighbors for the r'th round before sending a message for the r+1'th round.
Also, if a node is at the r'th round and receives a message from one of its neighbors for r+1'th round, the values will be updated in different cells of the array because of the round ID, and hence, will not interfere with each other.
Hence it will be consistent with the synchronous implementation.