Exam 1

CS 420 Fall 2019

You need to submit this to Exam1 folder in D2L.

1. We talked about leader election being an important problem in distributed systems. In this assignment, we will implement a simplified version of the Bully Algorithm for the **leader election problem** in distributed systems. In our version of the Bully algorithm, we will also simulate node/link failure. The algorithm that you need to implement will be as follows: **[50]**

UPDATE: As discussed in class, you may ignore/remove any redundant/unnecessary if()...else() blocks.

- Consider a distributed system of 'n' nodes **arranged in the form of a ring**. One of the nodes is designated as the current_leader at the time of MPI initialization. Your code should have the provision for changing the initial value of the current_leader by checking for argv[1].
- Step 2: The current_leader can do one of the two: i) send out a HELLO message to its successor node in the ring, ii) OR decide to transmit a message to its successor in order to initiate a leader election process.
- The decision to initiate leader election has to be done in a probabilistic way. The current_leader generates a random number between [0, 1] and if this random number is above a certain THRESHOLD value, then initiate leader election. Leader election process involves first generating a random token (integer between 0 and MAX_TOKEN_VALUE) and then transmitting a LEADER_ELECTION_MSG to successor. The LEADER_ELECTION_MESSAGE consists of 2 integers – msg[0] = rank of the token generator, msg[1] = token value.
- If however, the random number is less than the THRESHOLD, then the leader would send out the HELLO message. The HELLO message consists of a single integer = HELLO_MESSAGE.
- When a non-leader node 'i' receives a leader election message from its predecessor, it too first has to take a probabilistic decision as to whether it will participate in the leader election process (in exactly the same way as the current_leader). If yes, then it too generates a random token value (mytoken) and compares with the value of the token it received. if (mytoken > msg[1]) OR if (mytoken == msg[1]) AND (myrank > msg[0]) then the node 'i' will update the msg[0] and msg[1] with its rank and mytoken values respectively. If no, then it keeps the LEADER_ELECTION_MSG unchanged. Regardless of the outcome, node 'i' will transmit the LEADER_ELECTION_MSG to its successor. It will then issue a blocking MPI_Recv() call to receive the results of the leader election process by waiting for a message with tag LEADER_ELECTION_RESULT_MSG_TAG.
- If a node receives a HELLO message, it first generates a random number and tests whether it is larger than a given TX_PROB. TX_PROB may be modified by the user at runtime by specifying argv[2]. If larger than TX_PROB, then it will simply send the HELLO message to its successor node. Otherwise, it will not transmit the HELLO message to its successor. This is a common way of simulating node/link failure in distributed systems design.
- Each non-leader node in the system, waits to receive either a HELLO or LEADER_ELECTION_MSG up to a TIME_OUT_INTERVAL period (in secs) using the **MPI_Iprobe()** function. If no message is received within that time, then it gives up and goes for the next round → hits and **MPI_Barrier()** statement and waits for all the other nodes in the system to also reach the barrier. This is a standard practice of implementing a round-based protocol wherein the nodes are synchronized

at the beginning/end of every round. The number of rounds the algorithm will run for may be changed through user input (using argv[2]).

- The current_leader will wait to receive back the HELLO or LEADER_ELECTION_MSG using a combination of **MPI_Irecv()** and **MPI_Test()** functions. If the current_leader times out (due to not receiving a HELLO message), then it will cancel the MPI_IRecv() and then free up the receive call Request resources using MPI_Cancel() and MPI_Request_free() functions.
- NOTE: only the HELLO message may be lost in the network!
- When the current_leader receives back the LEADER_ELECTION_MSG, it will update its current_leader = msg[0] and then send out LEADER_ELECTION_RESULT_MSG to its successor which is a 2-element integer array consisting of: msg[0] = new leader ID, msg[1] = new leader's token value. It will then also issue a blocking MPI_Recv() call to receive back the message with the LEADER_ELECTION_RESULT_MSG_TAG.
- When a node receives a message with the LEADER_ELECTION_RESULT_MSG_TAG, it updates its current_leader = msg[0] of the received message. It then prints out its new leader.
- At the end of the above steps, each node issues a MPI_Barrier() to allow for synchronization before starting the next round of iteration.
- To give a sense of how your program is executing, you should print out appropriate messages for each node in every round whenever it receives/sends a message; decision to participate in leader election; decision to transmit or not a HELLO message; time-out occurs; result of leader election process.

I will be providing you with skeletal programs (.c and .h files) to help you get started as well as maintain standard notations. **Submit your solution as a single zipped file midsem.zip containing 2 files – simplebully.h, simplebully.c**.