Solving Binary Consensus Problem with Synchronous Paxos Algorithm

CS403/534 - Distributed Systems
Course Project, Fall 2021

December 22, 2021

Deadline: January 9, 2022 23:55

1 The Binary Consensus Problem

In your course project, you are asked to solve the binary consensus problem. Binary consensus is a special instance of the consensus problem. As we have seen in lectures, in the consensus problem, distributed nodes aim to agree on a single value. For the binary consensus, the domain of these values has two elements: 0 and 1. You might think of this problem as multiple nodes trying to agree on a value of a bit.

Similar to the consensus problem, a binary consensus algorithm has to satisfy three properties:

- *Agreement:* If multiple rounds result in decisions, decided values must be the same.
- *Validity:* Any decided value must be one of the values provided by the clients.
- *Termination:* All nodes must eventually terminate.

As we have seen in the lectures, ensuring *termination* is impossible in the asynchronous setting due to the FLP result. In this project, you will develop a variant of Paxos for a synchronous system. However, termination will not be guaranteed again since our model will allow any number of nodes to fail at any time. For the details of the system model, please read the following section. In brief, your implementation is expected to satisfy only *ageeement* and *validity* properties.

2 Synchronous Paxos Algorithm

For this project, you will develop and implement a slightly modified version of the Paxos algorithm we have seen in the lectures. The main feature of the synchronous Paxos is that nodes run together in a synchronous or lock-step way.

At any time, all nodes are more or less at the same phase of the same round. The biggest difference between two nodes is one phase i.e., one node can be in Phase 2 of round r whereas another node can be in Phase 1 of round $r+1$ but no node can be in Phase 2 of round $r+1$ or Phase 1 of round r in this case.

Since the system is synchronous, we assume that channels are reliable and bounded i.e., there are no message losses and we can put a finite and fixed upper bound on the message delays. Although having a synchronous system is impossible in general (over internet and WAN), nodes (processes) of your Paxos will run on the same machine. Therefore, we can safely assume fast and reliable communication among nodes.

However, your implementation must tolerate node failures. For this project, we assume that nodes can have *crash-recovery* failures i.e., they can be unavailable for a limited amount of time, but they cannot forge arbitrary messages or send messages on behalf of others. Paxos algorithm works correctly if at most k out of $2k+1$ nodes fail at any time even if the failing sets are different at different points in time. We will relax this requirement and allow a node to fail at any point in time according to a probability. This probability will be one of the inputs to your program and it will be known to all nodes.

In your implementation, you will not really crash Paxos nodes (processes) by terminating or suspending them, but you will *simulate* a crash by sending `CRASH` messages from failing nodes. Before sending a message, each node will randomly decide whether to send the correct message expected by the algorithm or a `CRASH` message simulating a node failure. Probability of sending a `CRASH` message will be determined according to the input probability mentioned before. The `CRASH` messages are independent in the sense that probability of node failures does not depend on previously send or received messages. Please note that round leaders (proposers) might fail and send `CRASH` messages as well.

In order to implement failure semantics described above, you can wrap the original message *send* method with another method *sendFailure (msg, proposer, target, prob)* that sends `"CRASH` *proposer*`"` message to the *target* with probability *prob* and sends *msg* to the *target* with probability $1-prob$. You can also implement your failure-aware broadcast method *broadcastFailure (msg, proposer, N, prob)* that calls *sendFailure(msg, proposer, i, prob)* for all $i \in [0, N-1]$. Here, *proposer* is the node ID of the leader of the current round that the sender node is in. The sender or the receiver do not have to be the proposer. You can use *proposer* field to differentiate `CRASH` messages from different rounds.

As you can imagine, probabilistic failure might cause more than half of nodes to fail at any time. Similar to the asynchronous setting, this only affects the *termination* property and *agreement* and *validity* properties must still hold. We will ensure termination by letting nodes run for a fixed and predefined number of rounds. They will run for these amount whether they reach to multiple decisions or none.

In the rest of this section, we provide details of the initialization and how Paxos nodes behave.

2.1 Main Process

Your program will start by running the main process. The arguments of the program will be input to the main process. You should expect three inputs to your program from the command line in this order: *numProc*, *prob* and *numRounds*. Here, *numProc* is the total number of Paxos nodes (processes), *prob* is the probability of a node failure and *numRounds* is the number of rounds these processes will execute.

After getting the input, the main process will create *numProc* number of *new* Paxos processes. Input of the main should be known by Paxos processes as well. So, you can pass *numProc*, *prob* and *numRounds* to Paxos processes as arguments. In addition, the main process must assign a unique ID (a unique number in $[0, N-1]$) and an initial binary value (0 or 1) to propose to new processes. In brief, signature of the method executed by new Paxos processes must be like this: *PaxosNode (ID, prob, N, val, numRounds)*.

The main process does not act as a Paxos node. After creating Paxos processes, it waits until all Paxos nodes terminate.

Note that synchronous Paxos could not generate all behaviours of asynchronous Paxos. For instance, when the majority of nodes are at some round r and some other node in round $r' < r-1$ can send a vote message and cause a vote quorum and decision to be formed for round r'. However, this case is possible in asynchronous executions.

2.2 Paxos Processes

In the original Paxos algorithm, each node (process) has two roles: *proposer* and *acceptor*. In the asynchronous or partially synchronous setting, these roles are implemented by two distinct threads of the same process. However, in synchronous Paxos, there is a <u>single</u> thread per process since nodes cannot receive unexpected or late messages which will require a separate thread listening to them. Therefore, you can safely assume that each process runs *PaxosNode (ID, prob, N, val, numRounds)* method that <u>does not</u> create any other thread.

Basically, *PaxosNode* consists of a loop that iterates in the range $[0, numRounds-1]$ in an ascending order. Each iteration corresponds to a single round of Paxos. At the beginning of each round, each node first decides its role whether it is the leader of this round (proposer) or not (acceptor). If the current round number is r, leadership can be easily checked with this formula: $r\%N = ID$. As you would remember, nodes of Paxos become round leaders in Round-Robin fashion. They first become the leader of the round with their ID and they become leader after every N rounds.

Before describing the behaviour of the proposer and acceptors, we introduce the local variables kept by the nodes. There are at least 4 local variables: $maxVotedRound$, $maxVotedVal$, $proposeVal$ and $decision$. $maxVotedRound$ is the maximum round number this node has voted for and $maxVotedVal$ is the corresponding value. Initially, $maxVotedRound$ is -1 and $maxVotedVal$ is `NULL`. $proposeVal$ is the value proposed by this node in the latest round in which it is the proposer and it has collected join messages from a majority of nodes. $decision$ is `NULL` initially. If this node can collect a majority of vote messages in a round in which it is the leader, then the $decision$ becomes the $proposeVal$ in this round.

If the node n is the leader of round r, it performs the following in this order:

- It first broadcasts a `START` message using *broadcastFailure* method described above. Since the broadcast is done by *broadcastFailure* method, each acceptor might nondeterministically receive a `START` or `"CRASH` $r\%N$`"`

message representing temporary crashes of the proposer. Since nodes are synchronized, you do not have to attach round number to the `START` message.

- The proposer blocks for receiving N messages (including one from itself as well) and keeps the `JOIN` count for these messages. There are three types of possible incoming messages: They can start with `JOIN`, `CRASH` or `START`. Receiving its own `START` message is interpreted as a successful join to this round. If the proposer does not receive a `START` message, it receives a `CRASH` for its place which will interpreted as failure of this node as an acceptor as well. Note that the proposer always receives N messages in total.

- If the total number of `JOIN` and `START` messages are bigger than $N/2$, then the proposer picks the `JOIN` message with the $maxVotedRound$ field and sets the $proposeVal$ as $maxVotedVal$ of this message. If it has received a `START` message from itself, its own local $maxVotedRound$ and $maxVotedVal$ variables are taken into consideration for the computation of $proposeVal$. If the resulting $maxVotedRound$ is -1 meaning that none of the nodes in the join quorum has voted before, then the proposer n sets $proposeVal$ to val (input of the *PaxosNode* method). Then, it broadcasts `"PROPOSE` $proposeVal$`"` message using *broadcastFailure* method.

- If the total number of `JOIN` and `START` messages are less than or equal to $N/2$, then the proposer broadcasts a `ROUNDCHANGE` message to all nodes and moves directly to the next round. `ROUNDCHANGE` message is sent for establishing the synchronization among nodes. Therefore, it must not be sent using the *broadcastFailure* and directly sent to all nodes (except itself) without any failure possibility.

- If n has previously decided to send a `PROPOSE` message, it blocks for receiving N messages. The possible messages it can get start with `VOTE`, `CRASH` or `PROPOSE`. If it receives its own `PROPOSE` message, it can be thought as a positive `VOTE` response to itself and then it updates the $maxVotedRound$ to r and $maxVotedVal$ to its own $proposeVal$. If total number of `VOTE` and `PROPOSE` messages are bigger than $N/2$, this round reaches to a decision and *decision* becomes $proposeVal$. After receiving N messages, n moves to the next round.

If the node n is an acceptor in round r, it performs the following in this order:

- It first blocks for receiving a message. The message is either `START` or starts with `CRASH`. In the former case, it responds to the proposer by sending the message `"JOIN` $maxVotedRound$ $maxVotedVal$" by calling the *sendFailure* method. As it is the case for regular Paxos, $maxVotedRound$ is the maximum round number n has voted for and $maxVotedVal$ is the corresponding value. Since the message is send via *sendFailure* method, the proposer might receive a `CRASH` message instead. In the latter case (if the acceptor receives a `CRASH` message from the proposer), it just sends a `"CRASH` $r\%N$`"`. This message can be sent by regular *send* method instead of using the *sendFailure*.

- Acceptor n again blocks for receiving a message. This time, it can receive a message starting with `PROPOSE`, `CRASH` or `ROUNDCHANGE`. In the first case, n responds to the proposer of round r by sending a `VOTE` message using *sendFailure* method. In this case n updates its $maxVotedRound$ variable to r and $maxVotedVal$ as the second field of the `PROPOSE` message. Note that the positive `VOTE` response of n might not reach to the proposer since *sendFailure* can alter it to a `CRASH` message. However, this does not incur any soundness problem on the algorithm. After sending the `VOTE` message, n moves to the next round. If n receives a `CRASH` message, it directly response the proposer back with a `"CRASH` $r\%N$`"` message. This message is not sent with *sendFailure* but, it is sent directly. After this message, n again moves to the next round. Lastly, if the received message is `ROUNDCHANGE`, n directly proceeds to the next round.

3 Synchronization

The Paxos algorithm explained in the previous section assumes synchronized execution of the nodes such that all nodes must be in the same phase of the same round. More precisely, when a node finishes a phase, it should wait for the other nodes to finish the same phase, they must all know the next phase and proceed to the new phase together.

However, this synchronization is not immediately guaranteed by the algorithm described above. Assume that initially all nodes are at Phase 2 of round r and the proposer of r broadcasts a `PROPOSE` message. The proposer of $r + 1$ might receive this `PROPOSE` message before another acceptor n. Even, the proposer of $r + 1$ might send a `VOTE` response for round r, move to round $r + 1$ and broadcast `START` message before n receives the `PROPOSE` message of r. In this case, behaviour of n becomes complicated. It might receive a `START` message of round $r + 1$ while it was expecting a `PROPOSE` or `ROUNDCHANGE` message from round r and when it moves to Phase 1 of round $r + 1$, it might receive a `ROUNDCHANGE` or `PROPOSE` message from round r while it was expecting a `START` message from $r + 1$.

A similar problem occurs when the proposer broadcasts `ROUNDCHANGE` message at the end of Phase 1. A fast going proposer of the next round might send a `START` message before the `ROUNDCHANGE` message of the current round arrives to an acceptor.

In order to prevent these kind of problems you have to ensure synchronization between phases. The first and the easiest solution is to implement a barrier at the end of each phase. You can find and use an off-the-shelf distributed barrier implementation from a library or you might implement one yourself.

Assume that you have such a *barrier* method at your disposal. Then, when the first phase of a round ends and the proposer of the round decides to broadcast a `ROUNDCHANGE` message, it can call the *barrier* method after the broadcast and acceptors can call the *barrier* method after they receive the $ROUNDCHANGE$ message. Similarly, *barrier* can be used at the end of Phase 2. After the proposer collects N `VOTE` responses and decides on the fate of the $decision$ variable, it can call *barrier* method. Similarly, acceptors call barrier after they send their `VOTE` responses. In all cases, nodes proceed to the next round after the *barrier* method returns.

3.1 BONUS: Barrier-Free Synchronization

Another way to ensure synchronization between phases is to figure out all possible messages that might arrive in the synchronization-free algorithm and take an action accordingly. However, this way is difficult and requires great care and effort. If you miss a small case, it might ruin all the execution causing deadlocks and incorrect computations. Therefore, it is strongly advised that you first ensure synchronization via barriers and obtain and store a stable implementation. Then, if you have time left, you might try developing a barrier-free implementation.

In order to simplify your job, we list all possible messages that can arrive at any time. We start with proposers:

- When the proposer collects N messages from the acceptors after broadcasting `START` message for round r, it can only get `START`, `CRASH` or `JOIN` messages from r. These are the expected messages in the synchronous execution.

- When the proposer collects N messages from acceptors after broadcasting `PROPOSE` message for round r, it can receive `VOTE`, `CRASH` or `PROPOSE` message from the current round (expected) or `START` or `CRASH` message from the proposer of $r + 1$. If the received message is a `CRASH` message from the next round, this should be noted and a `CRASH` answer must be returned in the next round. Also, messages of $r + 1$ must be discarded and not counted as part of expected N messages. The proposer must have received N messages from r.In this case, the proposer should not wait a Phase 1 message in the next round and directly send a Phase 1 response to the proposer of $r + 1$ when it moves to the round $r + 1$.

For acceptors, here are the possible messages and corresponding actions:

- When the acceptor is expecting a `START` message from the proposer of round r, it might receive a `START` or `CRASH` message from round r (expected) or `ROUNDCHANGE`, `PROPOSE` or `CRASH` message from $r - 1$. In any case, it must respond to the leader of r. If the received message is unexpected (one of the last three) the response depends on the previous message. If the message is `PROPOSE` or `CRASH` of $r - 1$, this node should respond to the proposer of $r - 1$ with an appropriate answer and update its $maxVotedRound$ and $maxVotedVal$ fields.

- When the acceptor is expecting a `PROPOSE` or `ROUNDCHANGE` message from the proposer of round r, it might receive a `PROPOSE`, `CRASH` or `ROUNDCHANGE` message from round r or `START` or `CRASH` message from round $r + 1$. If the received message is a `CRASH` message from the next round, this should be noted and a `CRASH` answer must be returned in the next round. If it receives a message from $r + 1$, for sure this node is an acceptor in $r + 1$ as well and it receives one of the first three messages from r in the first phase of $r + 1$. In this case, it can remember the decision here and send an appropriate response to the leader of $r + 1$.

4 Implementation Details

4.1 Network Topology

Communication between nodes will be established using ZeroMQ. In this section, we will describe an example network topology for Paxos. You are free to use this but we also encourage you to come up with your own design if you find something more suitable.

Our system will use the pipeline pattern of ZeroMQ. We will use a total of N ports. Port number base+i will belong to the i-th node (where base is a constant like 5550). Every node will create a total of N+1 (N=number of nodes) sockets. 1 of these sockets will be a PULL socket. The other N sockets will be PUSH sockets. The PULL socket will bind to the port number associated with this node. The PUSH sockets will each connect to a different port.

Then sending a message to a node can be achieved by using the PUSH socket connected to the port which belongs to that node. These PUSH sockets can be kept in a list or dictionary to easily find the socket connected to a particular node. Broadcasting a message can be done by sending the message through all PUSH sockets one by one. And receiving a message is simply done by using the single PULL socket.

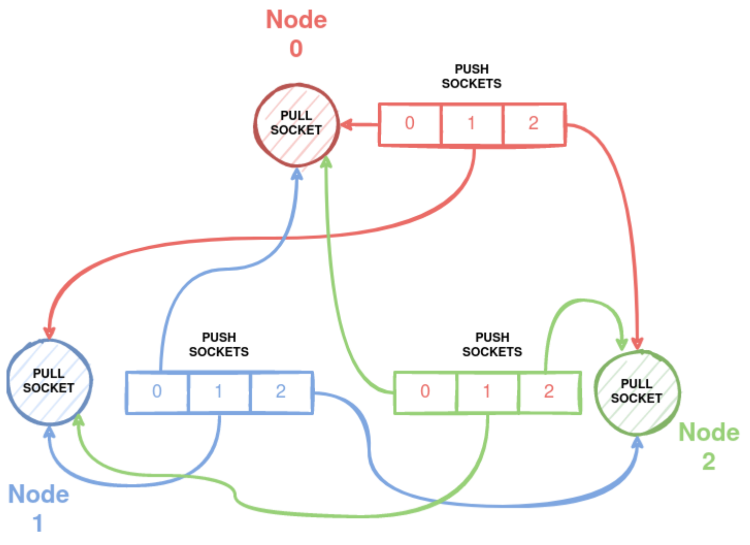

Figure 1: The diagram of the topology for 3 nodes. Note that each PUSH socket is connected to exactly 1 PULL socket. And a total of N (3) PUSH sockets connect to each PULL socket.

4.2 Output

Your programs should only be printing the information listed below. You can print other messages during development of course but please comment these out before submitting.

Below is a list of all the information that needs to printed and the structure of the print. Lowercase terms are variables.

- At the very start of the program number of nodes, crash probability and number of rounds should be printed as "NUM_NODES: num_nodes, CRASH PROB: crash_prob, NUM_ROUNDS: num_rounds"
- Messages received by the leader should be printed as "LEADER OF round_number RECEIVED IN phase: msg".
- Messages received by acceptors should be printed as "ACCEPTOR id RECEIVED IN phase: msg".
- Leader should print "ROUND round_number STARTED WITH INITIAL VALUE init_value" at the start of a round.
- When a value is decided the leader should print "LEADER OF round_number DECIDED ON VALUE value".
- If a ROUNDCHANGE occurs the leader that initiated it should print "LEADER OF ROUND round_number CHANGED ROUND".

4.3 Command-line Arguments

You are expected to take three arguments from the command-line. In order these should be number of nodes (or processes), probability of a crash (float between 0 and 1), number of rounds. Please, see the examples in the example runs section.

We might test the submissions using automated scripts so make sure to implement these arguments in the correct order.

Please check `https://www.tutorialspoint.com/python/python_command_line_arguments.htm` for more information on Python command line arguments.

5 Example Runs

In this section, two example runs are provided for your convenience. Note that your runs will not be identical to these due to randomness and the arrival order of messages.

In the example below notice that all rounds ended in a decision due to the low probability of crashes despite the number of nodes being small. Note that all the decisions are on the same value (agreement).

$ python paxos.py 4 0.1 3

```
NUM_NODES: 4, CRASH PROB: 0.1, NUM_ROUNDS: 3
ROUND 0 STARTED WITH INITIAL VALUE: 1
LEADER OF 0 RECEIVED IN JOIN PHASE: START
ACCEPTOR 2 RECEIVED IN JOIN PHASE: START
LEADER OF 0 RECEIVED IN JOIN PHASE: JOIN -1 None
ACCEPTOR 1 RECEIVED IN JOIN PHASE: START
LEADER OF 0 RECEIVED IN JOIN PHASE: JOIN -1 None
ACCEPTOR 3 RECEIVED IN JOIN PHASE: CRASH 0
LEADER OF 0 RECEIVED IN JOIN PHASE: CRASH 0
LEADER OF 0 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 1 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 2 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 3 RECEIVED IN VOTE PHASE: PROPOSE 1
LEADER OF 0 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 0 RECEIVED IN VOTE PHASE: CRASH 0
ROUND 1 STARTED WITH INITIAL VALUE: 1
LEADER OF 0 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 0 DECIDED ON VALUE: 1
ACCEPTOR 3 RECEIVED IN JOIN PHASE: START
ACCEPTOR 0 RECEIVED IN JOIN PHASE: START
ACCEPTOR 2 RECEIVED IN JOIN PHASE: START
LEADER OF 1 RECEIVED IN JOIN PHASE: START
LEADER OF 1 RECEIVED IN JOIN PHASE: JOIN 0 1
LEADER OF 1 RECEIVED IN JOIN PHASE: JOIN 0 1
LEADER OF 1 RECEIVED IN JOIN PHASE: JOIN 0 1
ACCEPTOR 0 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 2 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 3 RECEIVED IN VOTE PHASE: PROPOSE 1
ROUND 2 STARTED WITH INITIAL VALUE: 1
ACCEPTOR 0 RECEIVED IN JOIN PHASE: START
ACCEPTOR 3 RECEIVED IN JOIN PHASE: START
LEADER OF 2 RECEIVED IN JOIN PHASE: START
LEADER OF 1 RECEIVED IN VOTE PHASE: PROPOSE 1
LEADER OF 2 RECEIVED IN JOIN PHASE: JOIN 1 1
LEADER OF 2 RECEIVED IN JOIN PHASE: JOIN 1 1
LEADER OF 1 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 1 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 1 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 1 DECIDED ON VALUE: 1
ACCEPTOR 1 RECEIVED IN JOIN PHASE: START
LEADER OF 2 RECEIVED IN JOIN PHASE: JOIN 1 1
ACCEPTOR 1 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 3 RECEIVED IN VOTE PHASE: PROPOSE 1
ACCEPTOR 0 RECEIVED IN VOTE PHASE: PROPOSE 1
LEADER OF 2 RECEIVED IN VOTE PHASE: PROPOSE 1
LEADER OF 2 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 2 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 2 RECEIVED IN VOTE PHASE: VOTE
LEADER OF 2 DECIDED ON VALUE: 1
```

In the example below due to the relatively low number of nodes and relatively high probability of crashes none of the rounds ended in a decision.

$ python paxos.py 5 0.6 3

```
NUM_NODES: 5, CRASH PROB: 0.6, NUM_ROUNDS: 3
ROUND 0 STARTED WITH INITIAL VALUE: 1
LEADER OF 0 RECEIVED IN JOIN PHASE: CRASH 0
ACCEPTOR 1 RECEIVED IN JOIN PHASE: START
```

```
LEADER OF 0 RECEIVED IN JOIN PHASE: CRASH 0
ACCEPTOR 2 RECEIVED IN JOIN PHASE: CRASH 0
LEADER OF 0 RECEIVED IN JOIN PHASE: CRASH 0
ACCEPTOR 3 RECEIVED IN JOIN PHASE: START
ACCEPTOR 4 RECEIVED IN JOIN PHASE: CRASH 0
LEADER OF 0 RECEIVED IN JOIN PHASE: CRASH 0
LEADER OF 0 RECEIVED IN JOIN PHASE: CRASH 0
LEADER OF ROUND 0 CHANGED ROUND
ACCEPTOR 3 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 4 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 1 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 2 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ROUND 1 STARTED WITH INITIAL VALUE: 1
ACCEPTOR 0 RECEIVED IN JOIN PHASE: CRASH 1
ACCEPTOR 3 RECEIVED IN JOIN PHASE: CRASH 1
ACCEPTOR 2 RECEIVED IN JOIN PHASE: CRASH 1
LEADER OF 1 RECEIVED IN JOIN PHASE: START
ACCEPTOR 4 RECEIVED IN JOIN PHASE: START
LEADER OF 1 RECEIVED IN JOIN PHASE: CRASH 1
LEADER OF 1 RECEIVED IN JOIN PHASE: CRASH 1
LEADER OF 1 RECEIVED IN JOIN PHASE: CRASH 1
LEADER OF 1 RECEIVED IN JOIN PHASE: CRASH 1
LEADER OF ROUND 1 CHANGED ROUND
ACCEPTOR 2 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 0 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 3 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 4 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ROUND 2 STARTED WITH INITIAL VALUE: 0
ACCEPTOR 0 RECEIVED IN JOIN PHASE: START
ACCEPTOR 1 RECEIVED IN JOIN PHASE: START
LEADER OF 2 RECEIVED IN JOIN PHASE: CRASH 2
ACCEPTOR 3 RECEIVED IN JOIN PHASE: CRASH 2
ACCEPTOR 4 RECEIVED IN JOIN PHASE: CRASH 2
LEADER OF 2 RECEIVED IN JOIN PHASE: CRASH 2
LEADER OF 2 RECEIVED IN JOIN PHASE: CRASH 2
LEADER OF 2 RECEIVED IN JOIN PHASE: CRASH 2
LEADER OF 2 RECEIVED IN JOIN PHASE: CRASH 2
LEADER OF ROUND 2 CHANGED ROUND
ACCEPTOR 0 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 1 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 3 RECEIVED IN VOTE PHASE: ROUNDCHANGE
ACCEPTOR 4 RECEIVED IN VOTE PHASE: ROUNDCHANGE
```

6 Submission Guidelines

This project must be implemented in Python3 using ZMQ sockets. You need to submit the following files:

- `paxos.py`: This should be callable with the arguments defined in section 4.3.
- `readme.txt`: This file should contain the names of the group members. If you are doing the project individually then this file could be omitted.

Required file explained above should be put in a single zip file named as `CS_403-534_Project_`*name_surname*`.zip` and submitted to Project under assignments in SUCourse+. It is enough for one group member to submit this. Submission will be open until January 9, 2022 23:55 Turkish time. Late submission will be allowed for 1 day with a 10 point deduction for both group members.

7 Grading Criteria

- Execution (10 pts): Program runs without any major errors (apart from the termination step).
- Multi-Processing & Networking (10 pts): N processes are successfully created and can communicate with each other using ZeroMQ sockets.
- Termination (20 pts): All processes terminate.
- Correctness of the implementation (30 pts): We will execute the programs with different inputs to detect problems with the implementation.
- Demo (30 pts): Groups with sufficiently correct implementations will be asked for a demo where each member will be expected to answer some questions about the project. Groups with incorrect implementations will not be asked for a demo and will be unable get the 30 points from this section.
- BONUS (20 pts): If you implement the barrier-free implementation described in Section 3.1, you get 20 extra points.