

# An Implementation of the Paxos Consensus Algorithm

DIPAK BOYED  
MARK IHIMOYAN

## 1. ABSTRACT:

This report summarizes our programming assignment on providing an implementation of the Paxos „synod consensus algorithm based on the „Paxos made simple paper. We implement the Paxos algorithm to solve a simple consensus problem of selecting an alphabet between „a and „z and attempt to verify the correctness of our implementation based on our results and observations.

## 2. INTRODUCTION:

Consensus has been regarded as the fundamental problem that must be solved to implement a fault tolerant distributed system. Given a finite collection of members that can propose values, a consensus algorithm ensures that a single value among the proposed values is chosen. The Paxos consensus algorithm is an efficient and highly fault tolerant algorithm, devised by Lamport, for reaching consensus in a distributed system. This algorithm provides a fault-tolerant implementation of an arbitrary state machine in an asynchronous message passing system.

In this assignment we present an implementation of the Paxos synod consensus algorithm as applied to a finite number of processes running on a single machine. In our problem space, these processes attempt to reach consensus on selecting a letter from the 26 alphabets a – z (all lower case). Each process can propose a value (from letters a – z) and consensus is reached when the majority of the processes agree on a particular value.

The rest of the paper has been divided into the following sections. Section 3 describes the methodology used to implement the project and a detailed explanation of the source code. Section 4 discusses the Paxos algorithm used and lists our assumptions and implementation. Our results and observations from running the application are discussed in Section 5. We also attempt to prove the correctness of our implementation using the results and observations in Section 5.

### 3. METHODOLOGY:

The meat of this assignment was dealing with the implementation of the Paxos assignment. We religiously attempt to follow the 2-Phase proposal-acceptor algorithm and the requirements listed in the Paxos made simple paper. The model is the customary asynchronous, non-Byzantine system that assumes no message corruption.

#### 3.1 Problem Statement

Our application attempts to use Paxos in reaching a consensus on the following problem:

*Given a system of  $n$  nodes (each represented by a process) where each node can asynchronously propose an alphabet (between 'a' and 'z'), select a single one of the proposed values and notify all the nodes only once the value has been chosen.*

Our safety and liveness requirements are the same as those listed in the paper. Safety requirements ask for the following:

Only one of the proposed value is chosen

Only a single value is chosen

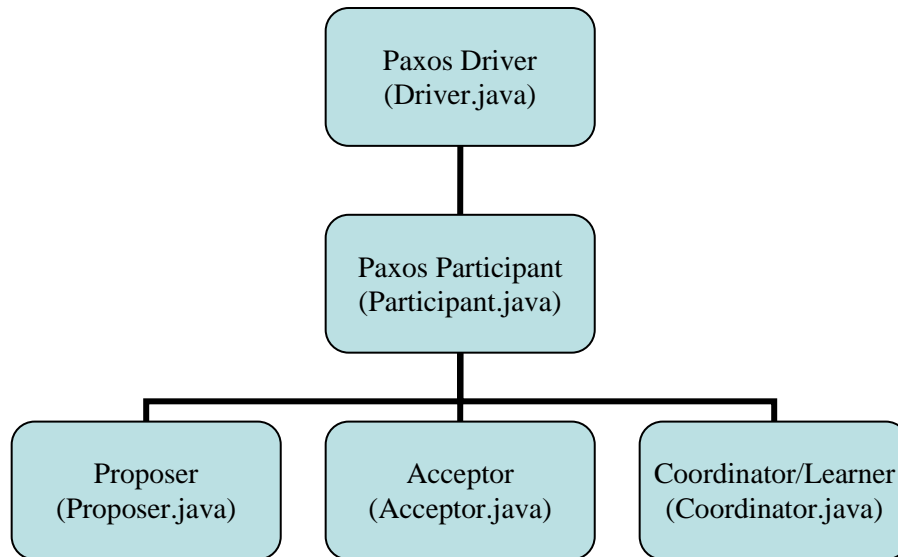
Members do not learn about the chosen value until after it has been chosen

Liveness requirements demand only that a value is eventually chosen and all members can eventually learn about the chosen value.

#### 3.2 Environment

The project was written in Java and compiled on the CS department Linux boxes with the Java 2 Runtime Environment, version 1.5.0\_06-b05. Please note that we do not guarantee for our application to compile and execute successfully on other platforms including Windows. All our implementation and testing work was solely done on the department Linux boxes.

The source code was roughly structured in the following way:



The project consists of one application (**paxos.jar**) that executes the `main()` function defined in `Driver.java`. The Driver class is useful for our simulation as it starts the Paxos participant processes and guarantees uniqueness of each proposal number.

### 3.3. List of source files

A short description of each file and its role is described in the table below:

File	Description
Driver.java	Entry point for the application that is responsible for the following: Starting the paxos participant processes Reply to messages from participant proposers requesting unique proposal numbers
Participant.java	Class representing a Paxos participant node. A Participant is responsible for the following duties: Proposer : propose values to coordinator Acceptor : accept values from coordinator Coordinator (maybe): Coordinate proposals and learn chosen value. Only one participant is the coordinator.
Proposer.java	Implements the Proposer role that is responsible for the following: Proposing values to the system via the coordinator Receiving proposal responses and consensus messages from the coordinator

	Notifying the Driver of a consensus result
Acceptor.java	Implements the Acceptor role that is responsible for the following: Receiving new proposal requests from the coordinators Responding and accepting the proposal requests
Coordinator.java	Implements the Coordinator role that is responsible for the following: Receiving new proposals from Proposers Communicating with acceptors about proposals Learning when a consensus has been reached Notifying proposers of consensus and failed proposals
Manifest.txt	Manifest information for the jar file.

### 3.4. Execution:

The Application can be executed on a designated machine by running the following command line:

➤ `“java -jar paxos.jar < n> ”`

*where, <n> is an integer representing the total number of participants to create.*

Assuming the above command is executed from the directory containing the Paxos source files. The application requires that the above command be executed from the source directory.

This will spawn the Driver process that will write to the default output stream and in turn spawn the participant processes. The participant processes will log their outputs to **\*.out** files in the same directory.

The driver application is not guaranteed to terminate due to possible message losses and network port issues. However, all important Paxos issues are logged to the \*.out files. A correct implementation of the Paxos algorithm along with timeouts for liveness will guarantee that the \*.out files will eventually contain consensus information provided that the majority of the nodes are healthy. Viewing the files alone is enough to determine whether consensus was reached or not.

### 3.5 Output

A screenshot of the sample output of running paxos with 1 participant is shown below.

The first figure contains the standard output of the driver (Driver.java) application. In this case the driver is responsible for starting one participant process. It then waits for the participant process to initialize and request for a unique proposal number. Upon receiving a proposal request it returns a new proposal number. Eventually, the participant process reaches a consensus and informs the driver.

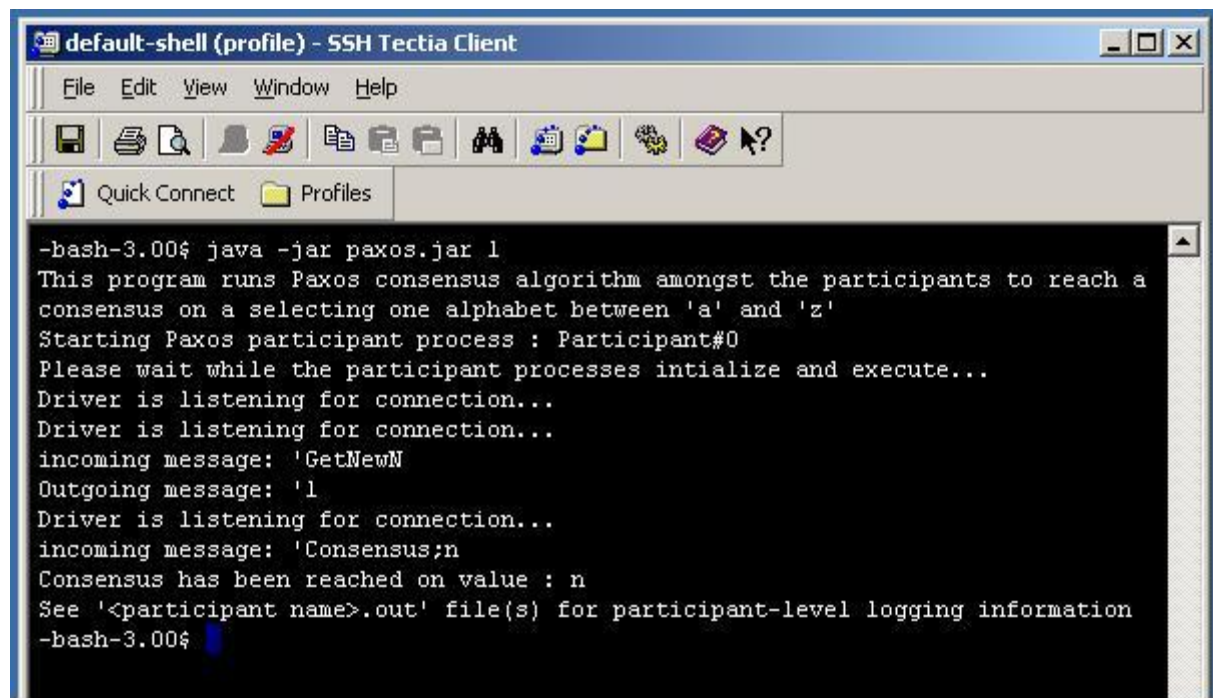


Fig 3.1 : Sample Driver's output when started

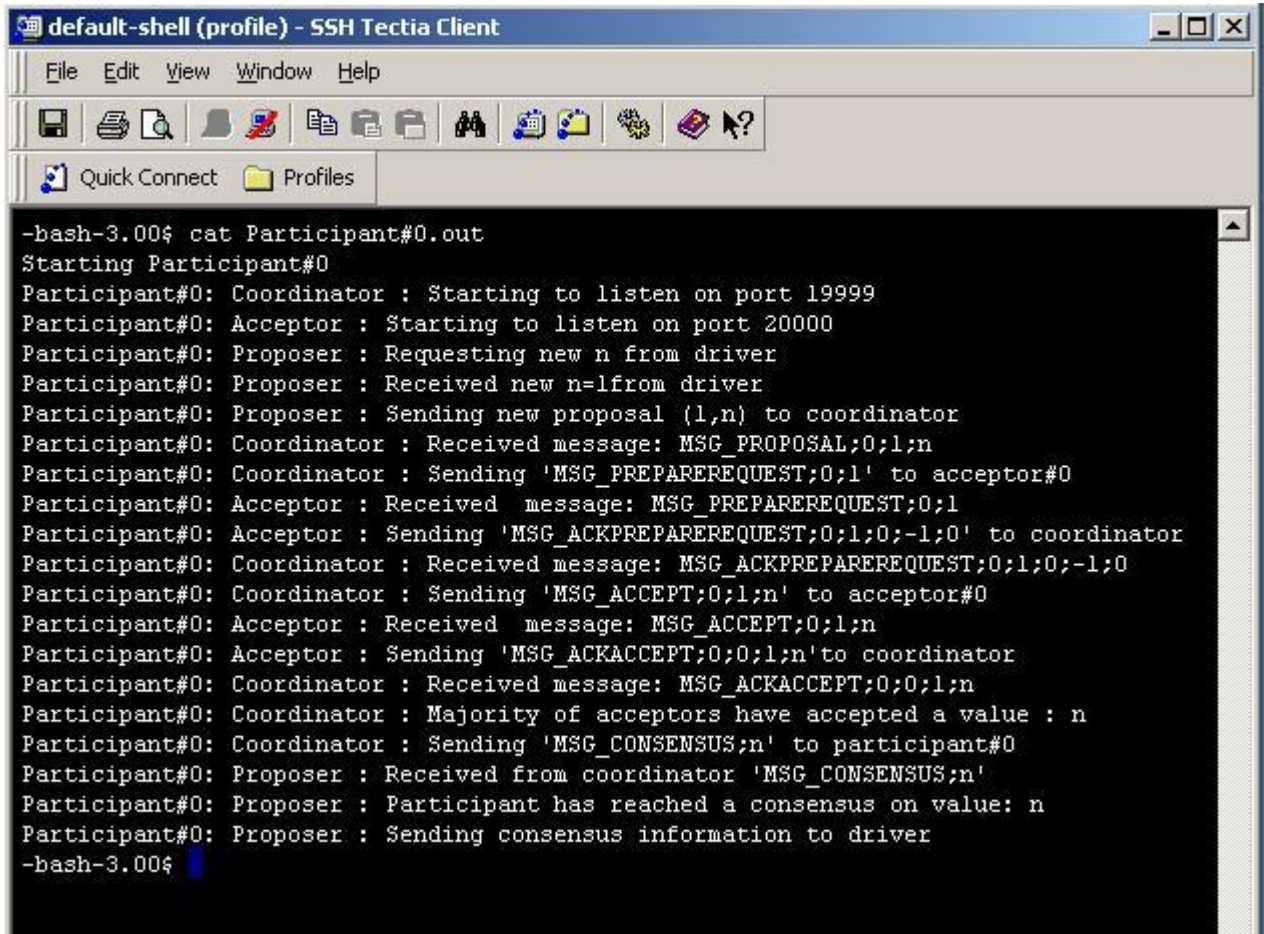
The more interesting logging part of the participant is shown in the second figure below. The single participant in this case plays all the roles of the Coordinator, Proposer and Acceptor. Each line of the participant logging subscribes to the following schema:

**<Name>: <Role> : <Message>**

where,

<Name>	=	Participant name
<Role>	=	The role responsible for logging the current line. Role can only be one of the following values: „Coordinator , „Proposer , or „Acceptor .
<Message>	=	The log message. Message usually contains state information (e.g. consensus reached), or sending/receiving of message

between the roles.



```
-bash-3.00$ cat Participant#0.out
Starting Participant#0
Participant#0: Coordinator : Starting to listen on port 19999
Participant#0: Acceptor : Starting to listen on port 20000
Participant#0: Proposer : Requesting new n from driver
Participant#0: Proposer : Received new n=1 from driver
Participant#0: Proposer : Sending new proposal (1,n) to coordinator
Participant#0: Coordinator : Received message: MSG_PROPOSAL;0;1;n
Participant#0: Coordinator : Sending 'MSG_PREPAREREQUEST;0;1' to acceptor#0
Participant#0: Acceptor : Received message: MSG_PREPAREREQUEST;0;1
Participant#0: Acceptor : Sending 'MSG_ACKPREPAREREQUEST;0;1;0;-1;0' to coordinator
Participant#0: Coordinator : Received message: MSG_ACKPREPAREREQUEST;0;1;0;-1;0
Participant#0: Coordinator : Sending 'MSG_ACCEPT;0;1;n' to acceptor#0
Participant#0: Acceptor : Received message: MSG_ACCEPT;0;1;n
Participant#0: Acceptor : Sending 'MSG_ACKACCEPT;0;0;1;n' to coordinator
Participant#0: Coordinator : Received message: MSG_ACKACCEPT;0;0;1;n
Participant#0: Coordinator : Majority of acceptors have accepted a value : n
Participant#0: Coordinator : Sending 'MSG_CONSENSUS;n' to participant#0
Participant#0: Proposer : Received from coordinator 'MSG_CONSENSUS;n'
Participant#0: Proposer : Participant has reached a consensus on value: n
Participant#0: Proposer : Sending consensus information to driver
-bash-3.00$
```

Fig 3.2: Sample Participant's log file

All the roles communicate with each other using fixed format messages that are assumed to be valid. A detailed description of the possible messages is given in Section 4.

The project submission also has a „SampleOutput“ folder which contains actual output logs from our test runs. The logs are verification tools and are a supplement to our Results and observation section (Section 5). More details on their organization are provided in Section 5.

#### 4. PAXOS ALGORITHM

This section describes the paxos algorithm details and implementation part of the project. We decided to implement Paxos using Lamport's „synod“ algorithm described in the Paxos made simple paper. Our algorithm is directly derived from the description of the two-phase message

communication between the acceptors and the coordinator. To guarantee progress, we route all proposals through a common source, the coordinator. The coordinator is also responsible for learning the chosen value and informing all nodes once a consensus has been reached.

We have implemented a general, working Paxos algorithm that helps us find a solution to our problem statement. Our algorithm works under the same set of assumptions listed in the Lamport paper. The only marked difference being that for simplicity and time constraint reasons, we have not implemented node recovery (persisting data to disk) and coordinator node failure cases described in the paper. Node recovery in our case will not remember any context prior to failure and will simply act as a new uninitialized node joining our system. We do not delve into a new coordinator selection upon the failure of the coordinator node. Coordinator failure in our case will stop further progress of our application.

Another slight deviation from the paper is noted in our implementation of unique proposal numbers. The paper talks about partitioning non-intersecting (distinct) range of numbers between the participants. However we felt this would give undue advantage to the participant receiving higher numbers. (Since the algorithm requires acceptors to accept higher proposal numbers). Instead, we force all proposers to request unique proposal numbers from a common source (the driver) which ensures fairness as well as uniqueness.

Our implementation satisfies all the requirements and assertions listed in the paper which include ensuring that „if only one value is proposed , it is selected , ensuring that once „a value  $v$  has been chosen, all future proposals will contain  $v$  , „only a single value is chosen , „participants learn about the consensus only after a value has been chosen by a majority of the acceptors .

#### **4.1 Implementation**

The implementation consists of the source files listed in Section 3. The types of files and their roles have also been listed in Section 3.

The figure below describes a high-level, process diagram of our system. Each participant process represents a node in our system. All nodes play the roles of a proposer and acceptor whereas only one node is the statically designated coordinator. The participant processes

execute independently of each other and hence add asynchrony to our model (because of context switches by the OS).

Each role played by a participant is represented by a separate thread (Proposer, Acceptor or Coordinator Thread). These threads in a process also execute independently of each other and the system thread switching model implies asynchronous execution.

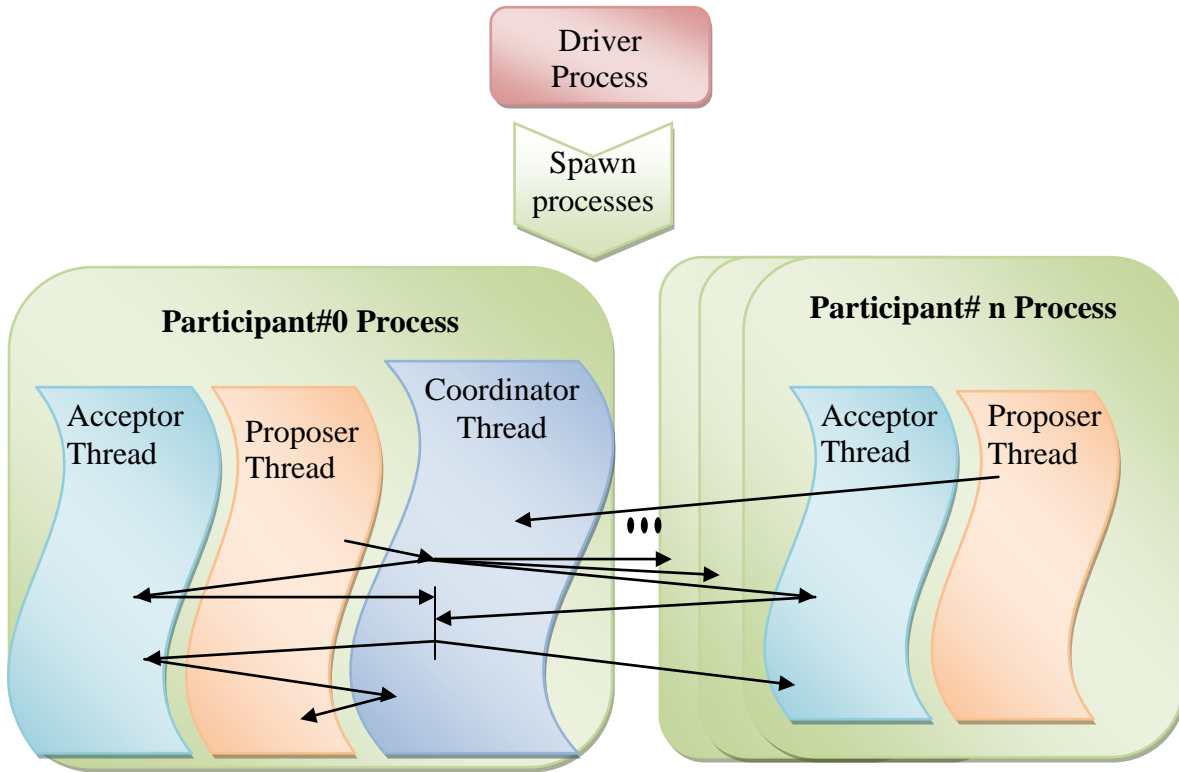


Fig 4.1 Process diagram of our system

We also assume that our distributed system has static membership of nodes and do not handle dynamic membership changes, node failures where the failed nodes recover and automatically want to re-join our system and remember their previous state.

## 4.2. Messaging

Various roles of the participants communicate with each other using messages. We have implemented messaging in our system using sockets and port-level, TCP messages. Each participant process and its corresponding roles were designated fixed port numbers to listen for messages on our local machine. Successful execution of our system requires all those ports to be available.



The following default port numbers were used:

Port Number	Usage
19998	Driver listens on this port for new proposal number requests and consensus information
19999	Coordinator listens on this port for messages from acceptors and proposers.
20000, (20000 + n -1)	One port for each of the n acceptor where they listen for messages from the coordinator.

The various types of asynchronous messages allowed in our system were specified in the following fixed format:

Message Type	Message Parameters	Source	Destination
MSG_PROPOSAL	pId ; n ; v	Any Proposer	Coordinator
MSG_PREPAREREQUEST	pId ; n	Coordinator	All Acceptors
MSG_ACKPREPAREREQUEST	pId ; n ; aId ; nA ; vA	Any Acceptor	Coordinator
MSG_ACCEPT	pId ; n ; v	Coordinator	All Acceptors
MSG_ACKACCEPT	pId ; aId ; n ; v	Any Acceptor	Coordinator
MSG_REPLYPROPOSAL	pId ; n ; FAIL ; vA	Coordinator	Proposer
MSG_CONSENSUS	V	Coordinator	All Proposers

where,

pId	=	Proposer Id
n	=	Proposal number
v	=	Proposed value
aId	=	Acceptor Id
nA	=	H ighest proposal num ber (if any) acknow ledged by an acceptor, „-1 otherwise.
vA	=	V alue accepted (if any) by an acceptor, „0 o therw ise.

For simplicity, the format is fixed and all messages are assumed to be valid. We did not incorporate error checking, validation and message corruption handling.

As an example, the sequence diagram below shows the possible message flow and state transition for a given proposal number  $n$  with proposed value  $v$ . This diagram represents the lifetime of a single proposal. Note that at a given time there are multiple such proposals alive in the system independent of each other. The message types are simply ordered by the states but otherwise are completely asynchronous. The message parameters used in this diagram have been defined in the table above.

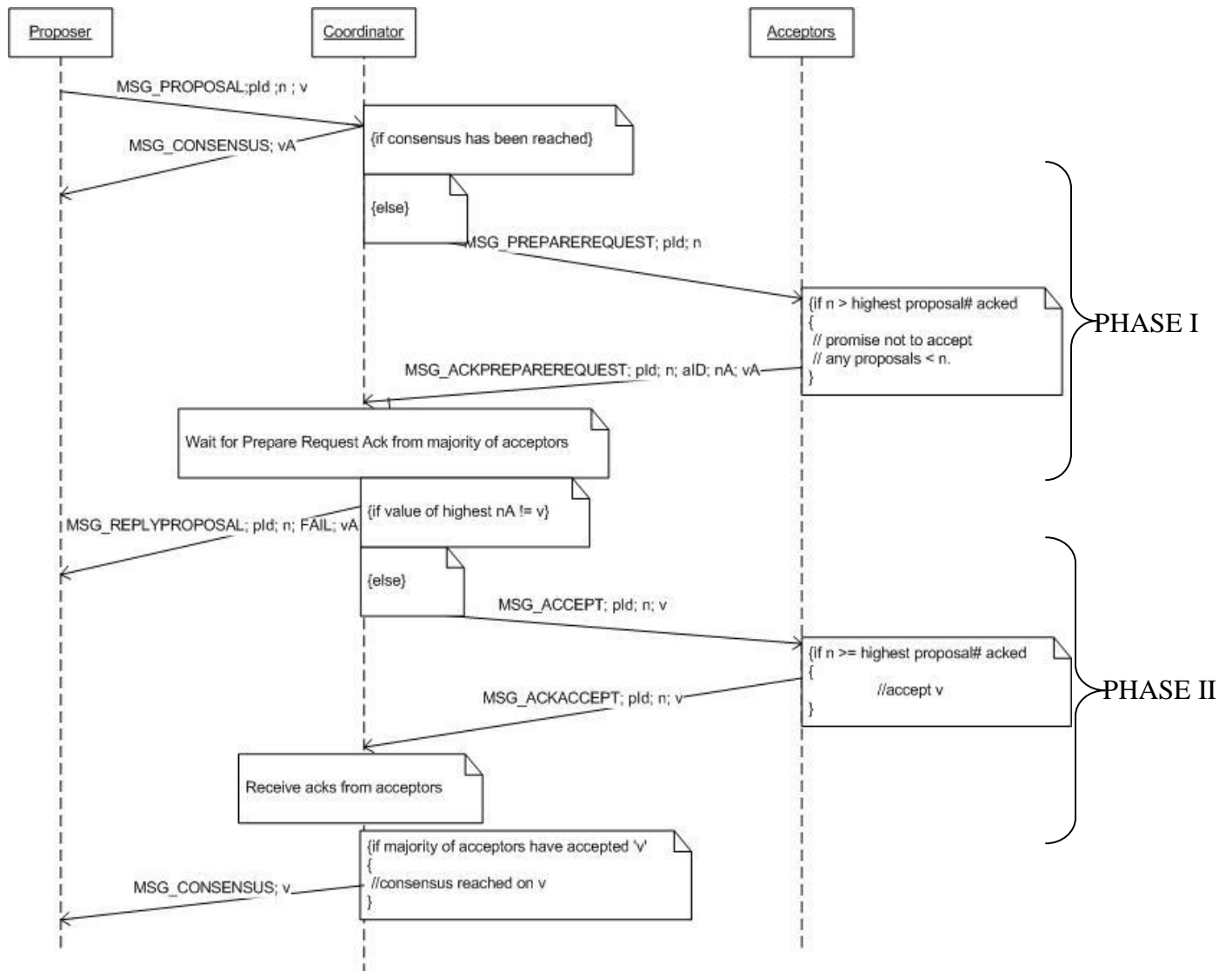


Fig 4.2 Message flow diagram

All the message types in our system represent arrows going from one state to another and hence can also be illustrated using a state transition diagram. The message types follows a certain order and can only be issued from states that need the particular message type to go to the succeeding state. For example, a MSG\_ACCEPT message for given proposal number  $n$  cannot be issued unless a MSG\_PROPOSAL for  $n$  has already been issued.

Messages that do not receive acknowledgements and successive state transitions are considered dropped. This optimization follows from the paper in order to reduce communication overheads. For example once an acceptor has acknowledgement a higher number proposal in Phase I (MSG\_ACKPREPAREREQUEST), it does not need to respond to new prepare request messages from older proposal numbers. Liveness of our system is the responsibility of the Proposer. The diagram above does not discuss liveness however our implementation assumes all proposals to receive some kind of response (successful or failed or consensus) within a statically fixed period of 1 minute. If no response is received within the given time frame, the proposal is dropped a new proposal is initiated. All new proposals also conform to the requirement that once a value has been chosen, all new proposals must select that same value. This timeout period also ensures that our system does not hang.

We also incorporate the rule that a given proposer can only propose one value at a time, as soon as a coordinator receives a new proposal from a given proposer; it ignores any previous pending proposal from that proposer. This optimization works because of the fact that all new proposal numbers  $n$  are greater than old proposal numbers and once a value  $v$  has been chosen all new proposals contain  $v$ .

## **5. RESULTS AND OBSERVATIONS:**

The goal of this section is to present data from our tests runs and verify our results. We also want to be able to prove the correctness of our Paxos algorithm to a logical extent. In addition to the above questions, we also attempt to look at scalability and performance issues. To supplement our results and help us prove our success, the submission also contains the „*SampleOutput*“ folder. This folder consists of actual participant log files from multiple test runs with varying number of participants. The „*SampleOutput*“ folder has been organized into folders containing logs from runs with different participant numbers. The log files contain

logging information about state changes and sending and receiving of messages. Each log message is individually synchronized and atomic. However please note that the logging of a message and the actual sending of a message is not atomic, hence the log messages in one file do not conform to strict temporal order.

We attempt to discuss our results and show the correctness of our implementation by answering the following questions:

*Was consensus reached?*

Yes. Our algorithm works on reaching a consensus as long as a majority of the participants including the coordinator are healthy and can communicate over their ports. A correct implementation of Paxos along with the timeout for liveness in the Proposer (discussed in Section 4) ensures that a consensus is eventually reached for healthy participants. We ran our application with as few as 1 participant and as many as 10 participant process and as long as the processes were created and executed successfully, were able to reach consensus.

*Did we satisfy the P(1) condition listed in the Paxos paper?*

“P1. An acceptor must accept the first proposal that it receives”

Yes. Running our application with 1 participant verifies that the acceptor always accepts the first proposal. (For 1 participant, only one proposal is initiated and accepted).

*Did we satisfy the P(2) condition listed in the Paxos paper?*

“P2<sup>b</sup>. If a proposal with value  $v$  is chosen, then every higher-numbered proposal issued by any proposer has value  $v$ ”

Yes. The state diagram (Fig 4.2) verifies that once a consensus has been reached (i.e. a value  $v$  has been chosen), all new proposals are failed and all proposers are notified of the value  $v$ . The proposers maintain an array of allowableValues and on receiving information about the chosen value update the allowableValues data structure to only contain  $v$ .

*Did we satisfy our safety requirements listed in Section 3?*

Yes. The logs for all our test runs confirm that only a single value is chosen and the chosen value is always one of the proposed values. We also ensure that participant processes learn about the consensus and the chosen value only after consensus has been reached. This is

facilitated by the coordinator which also acts as the learner. Consensus is reached only when the coordinator confirms that a majority of the acceptors have accepted the same value. Learning about consensus is facilitated using the MSG\_CONSENSUS message and that message type is only sent by the coordinator after a consensus has actually been reached.

*Did we satisfy our liveness requirements listed in Section 3?*

Yes. The liveness timer in the proposer ensures that there is no hang. This ensures that our application eventually reaches a consensus as long as the majorities of the participants including the coordinator are healthy and can communicate over their ports. We saw no problems when running our algorithms with as many as 10 participant processes. However a note must be made about the system context switching with 10 processes. For 10 participants we often saw that consensus was reached before all the participant processes got a fair share of system execution time. Also we noticed that the java process creation logic in our driver was not guaranteed to succeed and silently failed sometimes due to lack of memory.

*Did we meet the Phase 1 conditions listed in the Paxos paper?*

Yes. This was achieved using the MSG\_PREPAREREQUEST message sent by the coordinator to all acceptors and the MSG\_ACKPREPAREREQUEST message sent by the acceptors to the coordinator.

*Did we meet the Phase 2 conditions listed in the Paxos paper?*

Yes. This was achieved using the MSG\_ACCEPT message sent by the coordinator to all acceptors and the MSG\_ACKACCEPT message sent by the acceptors to the coordinator. More details on the message types can be found in Section 4.

*Is our implementation asynchronous?*

Yes. The independent execution of the participant processes ensures that all participant processes could propose their value asynchronously. We also implemented separate message ports for the coordinator and acceptors that listened for new connections and took action upon receiving new messages. Each role (coordinator, acceptor and proposer) was implemented using separate threads that executed asynchronously.

*Does our implementation successfully handle multiple node failure?*

Yes. As some of our test runs with 8 and 10 participants indicate that when a minority of the participant processes were failed (terminated or not scheduled for execution or had communication failure), we were still able to achieve consensus. Our algorithm is non-Byzantine, fault-tolerant as long as the majority of the participants including the coordinator are healthy.

*Do we successfully handle coordinator failure?*

No. As mentioned before we do not incorporate node recovery and data retrieval after a failure. A coordinator failure would mean running another consensus among the participant processes to choose a new coordinator. A poor man's solution could have been to let the driver ping the coordinator periodically and upon not receiving a response within a given time period assume the coordinator dead and designate a new participant as coordinator. However this solution would be susceptible to driver failure and would have also required the driver to be able to externally change the state of an already running process. We decided not to venture into these optimizations due to time and complexity constraints.

*Is our implementation scalable?*

Partly yes. We were able to successfully run our consensus algorithm with 10 participant processes. However creating more participants often led to OutOfMemory exceptions and some of the participant processes not being created and scheduled for execution in a timely fashion. Also, since we actively listen on many ports. Active scaling would require many more ports to be continuously available. Our implementation works correctly but is in no way a practical and ready to be publicly consumed product.

We also speculate that running the participant processes on different machines would greatly solve the scalability issues since the participant processes will not compete for system resources and the JVM heap size would be ample for the process on each machine. Extending our port-level communication layer to work with different machines would require minimal work.

Answering the above questions with the log files gives us ample confidence that for our practical purposes, the Paxos algorithm was implemented successfully.

## **7. CONCLUSIONS:**

We implemented a simplified version of the Paxos synod consensus algorithm to allow multiple participants to reach a consensus on selecting an alphabet between ,a and ,z . Our distributed system was executed for multiple topologies (minimum of 1 participant, maximum of 10 participants). We listed our assumptions made during the implementation and verified that the Paxos algorithm was correctly implemented for all practical purposes.