Bringing Paxos Consensus in Multi-agent Systems

Article · June 2014

DOI: 10.1145/2611040.2611099

2 authors, including:

Costin Badica
University of Craiova
282 PUBLICATIONS **1,434** CITATIONS

SEE PROFILE

Bringing Paxos Consensus in Multi-agent Systems

Andrei Mocanu
University of Craiova
Software Engineering Department
Bvd. Decebal 107, Craiova, 200440,
Romania
andrei.c.mocanu89@gmail.com

Costin Bădică
University of Craiova
Software Engineering Department
Bvd. Decebal 107, Craiova, 200440,
Romania
badica_costin@software.ucv.ro

ABSTRACT

Reaching consensus has long been regarded as one of the most important problems in distributed systems. Being able to do so under failures is addressed by the Paxos family of algorithms which is able to guarantee safety, while probabilistically satisfying progress. The role-based approach of Paxos makes it an ideal candidate for implementation using autonomous agents which can dynamically assume their part(s). This paper aims to bring the basic Paxos fault-tolerant algorithm into the multi-agent world by describing a system architecture and implementation under the Jade platform, and presents valuable experimental results for convergence in unfavorable cases.

General Terms

Algorithms, Applications

Keywords

Multi-agent System, Paxos, Fault-Tolerance, Consensus, Replication, JADE

1. INTRODUCTION

Multi-agent systems have come to play an important part in various domains ranging from e-commerce[4] to disaster management[14] and argumentation[5]. However, like all intricate parts of a distributed system, agents are prone to failures and special precautions must be taken in order not to affect the functionality of the system as a whole. Moreover, agents as intelligent and autonomous pieces of software must also learn how to deal with failures and pursue their goals despite them.

Consensus[9] is one of the most important problems in distributed computing. Ensuring that a set of processes will reach an agreement on a value proposed by one of them is quite a challenging matter. The famous Fischer-Lynch-Paterson (FLP)[6] result proves the impossibility of solving consensus deterministically in an asynchronous system under the presence of even a single failure. The Paxos family of algorithms [11], [12] is designed to solve consensus in a distributed system of unreliable processes by guaranteeing safety properties. Although Paxos does not guarantee making progress as well, the conditions in which it is stalled are quite improbable. The Paxos algorithm is a prime example of research that takes time to be adopted by the community. Although L. Lamport formulated the algorithm in the 1980s, his paper from 1990 was rejected, only to be published years later in 1998. It has since received increased attention in the following years and also found its way in practice sitting at the heart of Google products Chubby[2] and Megastore[1], Microsoft's Autopilot[8] and Apache Hadoop's Zookeeper[7].

Paxos also has great importance for the topic of replication in distributed systems. Replication is a common method to protect data from various types of failure, regardless if they are software or hardware in nature. However, replication depends on the model that is used for communication, either synchronous or asynchronous. The synchronous model implies that the maximum communication delay is known, so it can be easily detected whenever a replica becomes unresponsive. In the case of the asynchronous model, it is more difficult to detect whether a replica has indeed failed. It is also for replication under the asynchronous model where we must have a total of $2f+1$ processes in order to tolerate f failures [3].

It is very common in distributed systems to model the different processes involved using "state machines"[10] as the abstraction. State machine replication assumes that every replica sees the operations in the same order, which for the deterministic case is analogous to the replicas reaching the same state. A special *primary* process is nominated which chooses the order of operations, sends these operations to other processes and replies to clients. However, when the primary fails a new one must be elected by having a majority decide on it. The primary election requires a "view-change" algorithm[13] which in turn uses Paxos consensus to support decision-making.

In this paper, we propose a model to seamlessly integrate the Paxos family of protocols into the multi-agent world. We use the Jade multi-agent platform to implement the basic Paxos algorithm and provide experimental results for some interesting scenarios using our implementation.

2. BACKGROUND

The Paxos family of algorithms satisfies three key safety properties[12] :

- the value that is chosen must be from the set of values that were proposed
- there is only one value that is chosen
- the only circumstance in which a value is learned by a process is if that value has been chosen

Additionally, there are some characteristics that the processes involved must satisfy:

- the processes involved operate at variable speed, may fail by halting, or they may restart
- there is no upper bound for message delay and messages may be lost, reordered or duplicated, but they are not corrupted

The Paxos algorithm assigns roles to processes, and each role contributes towards the correctness of the protocol. It is also possible for processes to play multiple roles during the lifetime of the protocol, but the actual processes to roles mapping in the implementation was not covered in Lamport's paper.

1. **Client** - is the process that makes the request in the distributed system (by contacting Proposer processes) and then waits to be informed of the result (by Learner processes).
2. **Proposer** - acts on behalf of the Client, trying to persuade Acceptors to agree to a Client request. They are also important for ensuring that progress is made in the protocol despite conflicts.
3. **Acceptors** - these processes behave as the fault-tolerant "memory" of the protocol. They organize themselves in collections called Quorums and each message sent to an Acceptor (regardless of its nature) must also be sent to a Quorum of Acceptors. Subsequently, a message received from an Acceptor is discarded, unless it is received from a Quorum of Acceptors. Since these Quorums represent absolute majorities of Acceptors, it means that two Quorums share at least one Acceptor.
4. **Learners** - they ensure that the request is carried out in the distributed system and a response is sent to the Client by each informed Learner, as soon as they receive the agreement notice from the Acceptors. Multiple Learners may be added to increase availability.
5. **Leader** - it is a special Proposer process, needed by the protocol in order to make progress. While safety is guaranteed, if multiple Proposers believe themselves to be Leaders it may result in conflicting proposals, which in turn delays progress.

The basic Paxos algorithm[12] runs in two phases which are outlined below:

Phase 1.

a **Prepare** - A proposal with identifier n is sent by one of the Proposer processes to the Acceptor processes attempting to assemble an absolute majority.
b **Promise** - Whenever an Acceptor receives a Prepare message, numbered with n, that is greater than the identifiers of any proposals it had received thus far, then it replies with a promise not to accept any proposal numbered less than n and sends the highest numbered accepted proposal if there exists one. Otherwise, if the proposal identifier is not greater than the one it holds, the Acceptor could simply disregard that message, though it is best practice to inform the Proposer by sending a `NACK` message.

Phase 2.

a **Accept Request** - If the Proposer gathers replies from an absolute majority of Acceptors to the proposals with number n it has made, then it issues an *Accept Request* to those Acceptors for a proposal with number n and value v (the maximum proposal number among the responses, or an arbitrary value should there be no other proposals).
b **Accepted** - Should an Acceptor receive an *Accept Request* for a proposal with number n, it will accept that proposal unless it had responded to a different *Prepare* message numbered greater than n. If it had, for the purpose of optimization, it may send a `NACK` message to the Proposer but it is not required to.

In order to aid understanding, we present a simple example of a successful basic Paxos protocol in Figure 1.

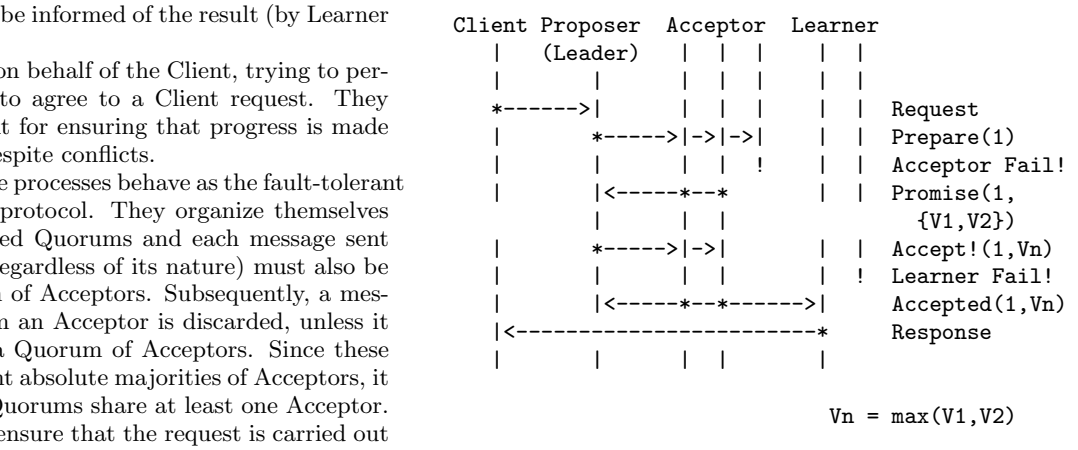

Figure 1: Paxos Successful Example

There is only one Proposer in this scenario (thus it is also the Leader), three Acceptors and two Learners. We may note that the number of Acceptors needed to reach an absolute majority in this case is two. Even though one Acceptor process fails before replying to the Proposer the protocol succeeds because two Acceptors accept the proposal. Furthermore, even though one of the two Learner processes fails during this run, there is still one alive at the very end to inform the Client of the decision.

3. SYSTEM DESIGN

Before we discuss our system architecture, we must look ahead to what it is we are actually striving to build. Our goal is to produce a distributed multi-agent platform that supports the Paxos protocol through its intelligent agents. This is equivalent to having a subset or all agents in the system play one or more roles in the Paxos algorithm, similar

to processes in the previous section, but it does not prevent them from doing additional tasks or cohabitate with non-Paxos agents as well. Our system must enforce that Paxos agents conform to their role-specific responsibilities, it should give them the possibility of discovering one another and by being distributed, it should be flexible to various deployment schemes.

Since agents according to their nature follow their own agenda, integrating them into the consensus algorithm means programming each agent to behave correctly in respect to the role(s) they are performing. The correctness of our distributed application follows from that of the individual entities. We could design our autonomous agents in two manners: map an agent type to exactly one role or conceive a more complex agent that is able to handle one or more roles at the same time.

The first approach is outlined in Figure 2. The agents involved have a single clear role they have to fulfill, and they are instantiated specifically for this purpose using the four available agent types. Although lightweight and straightforward this approach suffers from some disadvantages such as the large number of agents needed and their rigidity.

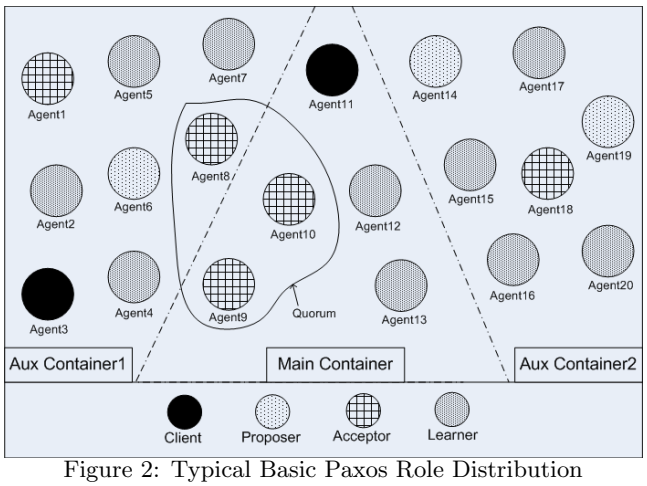

Figure 2: Typical Basic Paxos Role Distribution

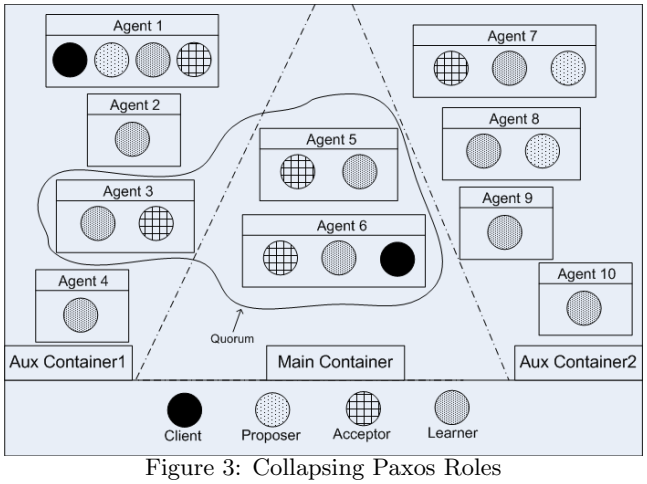

Figure 3: Collapsing Paxos Roles

In the second approach described in Figure 3 we cannot discuss about fixed agent types, but rather about agents that are able to play one or more roles simultaneously. We can achieve this by incorporating each role into an agent behaviour and then adding or removing a certain behaviour for a specific agent. This is the approach we feel is most natural to multi-agent systems and it also reduces the required number of agents (halves it for our example).

For both approaches we observe that the agents exist within agent containers which can be deployed at various locations. We also notice that for our example there is no Proposer role in the *Main Container* and no Client role in *Auxiliary Container 2*. This potentially means that individual sites need not know how to generate all agent types in the first approach or agent behaviours in the second, but a limited subset of these.

Furthermore, we must also consider a way in which agents are able to find one another. A *Yellow Pages* type directory may be used so that agents are able to post information about themselves and also retrieve information about other agents that have previously registered in the directory.

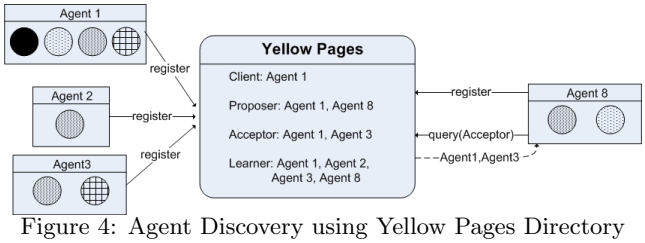

Figure 4: Agent Discovery using Yellow Pages Directory

As can be seen from Figure 4, the agents start by registering the services they provide to the *Yellow Pages* directory. This will keep track of which services are offered by the agents in our system. For example, if Agent 8 wants to know which are the Acceptors it can contact, it queries the directory and retrieves a list of agents providing that service. Querying can be done periodically to keep the agents updated.

4. IMPLEMENTATION

Commencing from the system design, we have built our custom implementation that benefits from the Jade platform facilities. The classes are structured in two different packages: *ro.ucv.agents* and *ro.ucv.bootstrap* outlined in Figure 5. The former focuses on the agent roles, their behaviours and interactions, while the latter addresses the interaction with the platform and provides important helper functions.

The *JadeHandler* class is responsible for invoking the platform, creating the main container and any auxiliary containers required. The *MessageManager* simplifies the manner in which messages are sent between agents. Both classes are invoked by the *ExperimentRunner* class which creates the required agents according to the command line parameters passed to the application. Furthermore, for simulation purposes, it is in this class that we trigger the Client to start the Paxos protocol by passing an invocation message through the *JadeGateway*. Normally, in Paxos the Client would initiate the request autonomously, but here we use the *JadeGateway* to control when this happens in order to simplify our experiment setup.

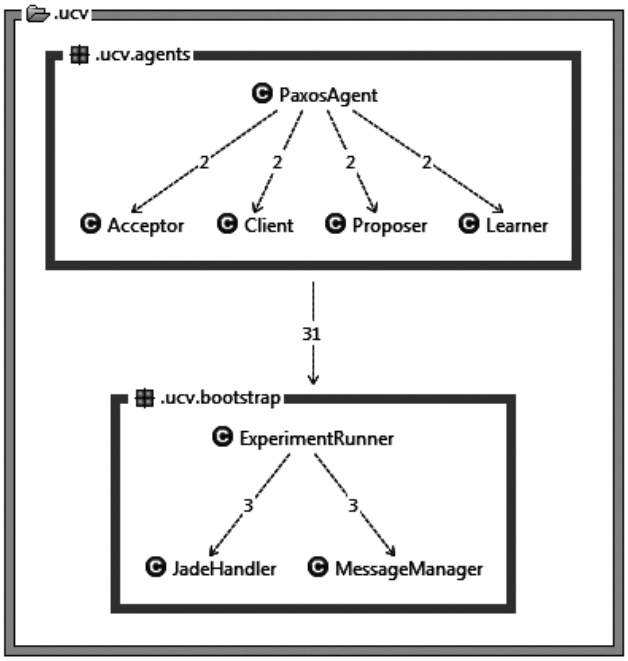

Figure 5: System Class Structure and Dependency Analysis

The four behaviours have specific implementations to match the role they are designed for in the Paxos algorithm. Our *PaxosAgent* can incorporate one or more of these behaviours according to how it is instantiated.

Another important aspect we have discussed in the system design is agent discovery. Over time, agents may terminate, configurations may change, so it is important for them to stay up to date and be aware of their peers. The Directory Facilitator in Jade centralizes the agent information and can be used both for registering a certain service an agent provides and for searching for agents providing a certain service.

5. EXPERIMENTAL ANALYSIS

5.1 Preliminaries

In order to test our implementation we decided to reproduce a special case in which multiple Proposers believe themselves to be Leaders and try to assemble a Quorum of Acceptors. We believe that this scenario is relevant for the Paxos algorithm because it sets our expectations for efficiency under "worst-case" conditions and acts as a baseline for further development.

The "Dueling Proposer" scenario is detailed in Figure 6 which illustrates the interactions between the agents involved. As can be observed, as a result of a Proposer failure a different Proposer assumes the role of the Leader. Upon the first agent's recovery it attempts to continue from the point where it failed. Unfortunately this carries on until one Proposer is allowed sufficient time to complete its run, i.e. no other Proposer intervenes until the protocol has finished. One way in which this could be achieved is to have the dueling Proposers sleep for a random amount of time whenever they receive a `NACK` message, but we decided to omit this for our experiments in order to observe the performance of basic Paxos.

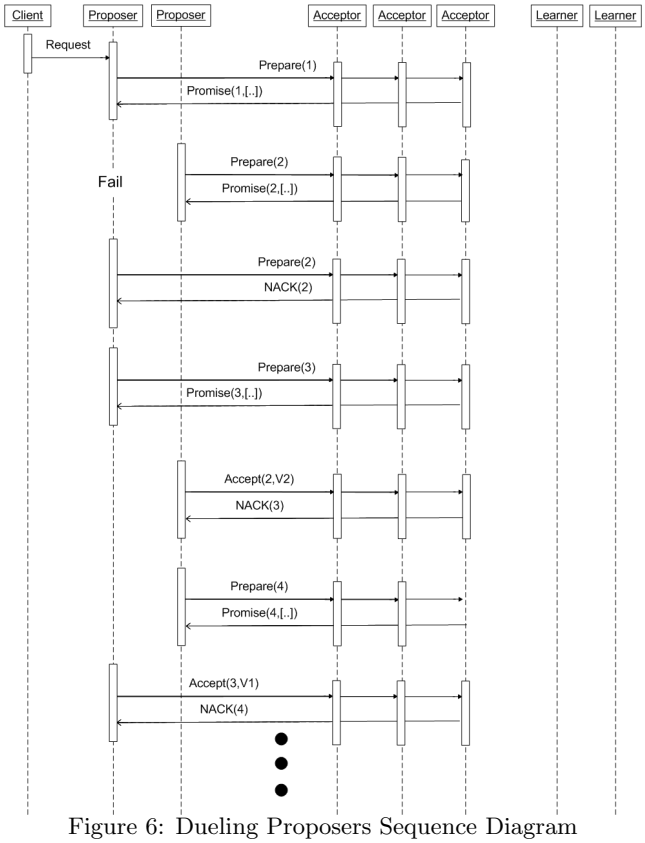

Figure 6: Dueling Proposers Sequence Diagram

Before we discuss the results we must clarify the object of our measurement, the number of rounds until convergence. We say that a new round is started whenever a Proposer sends a message belonging to Phase 1, as described in the protocol description. A successful round is one in which all the phases and sub-phases of the protocol are carried out.

Our implementation supports message loss simulation by having agents ignore certain messages proportional to the rate that we choose. For this experiment we decided to focus on unreliable channels, but faulty agents could be introduced just as easily by simply ignoring all communication from a particular agent or by terminating it.

5.2 Experiment Setup

We started from a configuration of 3 Proposers, 5 Acceptors and 5 Learners, each mapped on separate agents by default on the platform. Since it is important that message delays are kept uniform, it is better to provide a one to one mapping between agents and roles. Hence, we are able to avoid situations where one role would interact with another inside the same agent producing negligible message delays. We then altered the initial configuration in two different directions: on one hand we increased the number of Proposers present in our distributed system, and on the other hand we increased the number of Acceptors and Learners. We simulated message losses of 10%, 30% and 50% and measured the number of rounds, that were necessary in order to reach consensus in each case. The tests were carried out on a Microsoft Windows 7 machine, quad-core CPU i5@2.67Ghz, 4GB RAM, JDK7, JADE version 4.3.1.

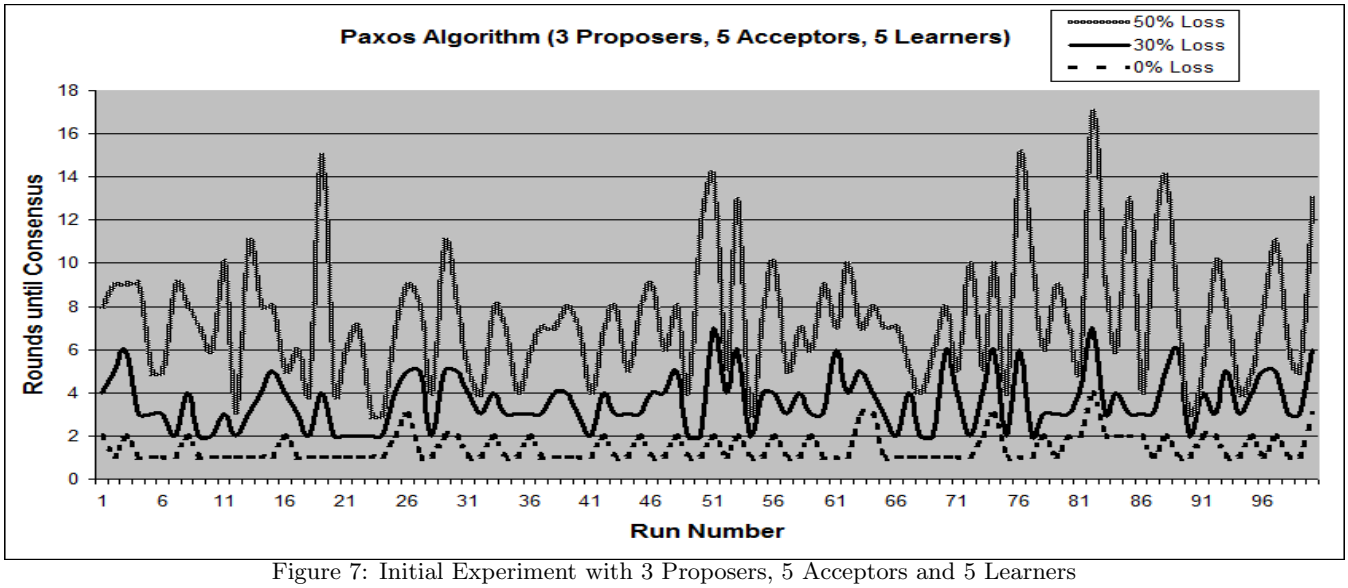

Figure 7: Initial Experiment with 3 Proposers, 5 Acceptors and 5 Learners

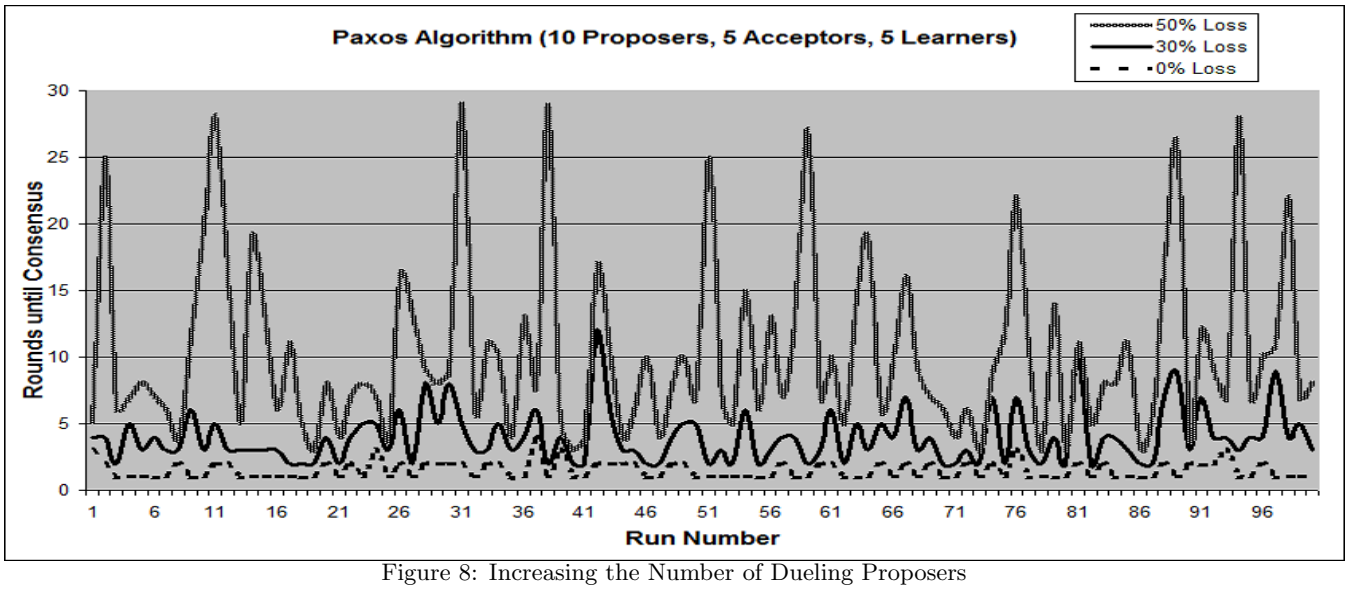

Figure 8: Increasing the Number of Dueling Proposers

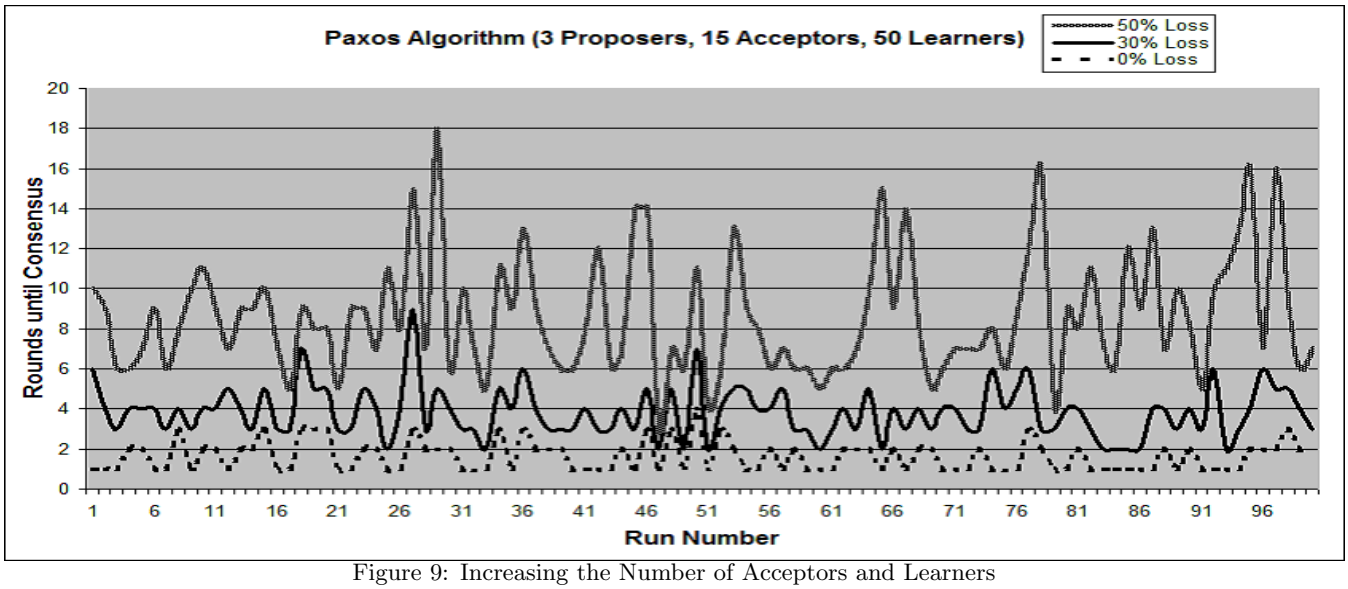

Figure 9: Increasing the Number of Acceptors and Learners

5.3 Results and Discussion

In table 1 we present our results from one hundred runs for each of the given configurations.

Configuration / Message Loss	10%	30%	50%
3 Proposer ,5 Acceptor ,5 Learner	1.42	2.18	3.83
10 Proposer, 5 Acceptor, 5 Learner	1.47	2.48	6.36
3 Proposer, 15 Acceptor, 50 Learner	1.63	2.2	4.78

Table 1: Average Number of Rounds until Consensus

The iterations of our experiments are detailed in Figures 7, 8 and 9. Confirming the initial expectations and intuition, by increasing the message loss rate, the number of rounds necessary to reach consensus also increased in all three configurations.

Furthermore, we may notice that all subsequent increases tend to be superlinear to previous ones, although this phenomenon seems less obvious for the initial configuration in Figure 7. By adding more Proposers in our second setup detailed in Figure 8, the average number of rounds grew slightly for lower message loss rates and increased by 66% when half of the messages are not delivered. Finally, increasing the number of Acceptors and Learners in comparison with our initial setup is detailed in Figure 9. We notice that the average number of rounds increases once again, but less dramatically than the previous case.

6. CONCLUSIONS & FUTURE WORK

This paper has demonstrated how Paxos consensus can be successfully implemented in multi-agent systems. We have produced a distributed application based on Jade for which we have leveraged agent discovery capabilities present in the platform through the Directory Facilitator agent. We have also conducted experiments for certain edge cases in Paxos using different agent configurations which our application allowed us to effortlessly change.

Moreover, the experiments we have conducted for dueling Proposer scenarios showed a superlinear increase in the number of rounds to convergence, as the message loss rate also increased.

We plan to introduce other implementations of the Paxos algorithms, such as Cheap Paxos or Fast Paxos and analyze how these behave in a multi-agent environment, and also use the results presented in this paper as a benchmark for future developments.

Furthermore, our aim is to integrate these algorithms into critical multi-agent system applications in which our agents could benefit from the dynamic collective decision making scheme proposed in Paxos. A further direction is to explore server management and availability by proactive, intelligent agents.

Finally, since multi-agent platforms are being adopted on mobile operating systems, another possibility is to investigate a Paxos crowd-based fault-tolerance layout in which users lend their hardware for running the agents that form the system.

7. REFERENCES

[1] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[2] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[3] Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+ 1 processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 475–480. ACM, 2010.

[4] Adriana Dobriceanu, Laurentiu Biscu, Amelia Badica, and Costin Badica. The design and implementation of an agent-based auction service. *IJAOSE*, 3(2/3):116–134, 2009.

[5] Xiuyi Fan and Francesca Toni. Assumption-based argumentation dialogues. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 198–203. AAAI Press, 2011.

[6] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[7] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[8] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[9] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*, chapter Consensus and agreement algorithms. Cambridge University Press, Cambridge, United Kingdom, 2008.

[10] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

[11] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[12] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[13] David Mazieres. Paxos made practical. Available at `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, 2007.

[14] Andrei Mocanu, Sorin Ilie, and Costin Badica. Ubiquitous multi-agent environmental hazard management. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 513–521. IEEE, 2012.