Paxos Algorithm Use in Smart Meter Technologies

Wasfi Momen   
CSC 6220  
Department of Computer Science – Georgia State UniversityAtlanta, Georgia   
[wmomen1@student.gsu.edu](mailto:wmomen1@student.gsu.edu)

# Introduction

As one of the oldest distributed systems algorithms, Paxos is rarely implemented in production software yet still leads a healthy presence in academic research. Unfortunately, Paxos holds a reputation for being complex and hard to understand since it was made during a time of experimentation. However, the future of cloud computing requires candidates for new problems and Paxos is one algorithm that deserves to be on the list. In this paper, we will explain Paxos at the complexity of an undergraduate networks class and also produce a Paxos sample from the research provided.

## **History of Paxos**

A researcher in distributed systems and the creator of LaTeX, Leslie Lamport first came up with the idea of Paxos in 1990 in his paper “The Part-Time Parliament” [Parttime Parliament]. Lamport had seen a presentation on another built fault-tolerant system named Echo, but the system required many states to handle any potential errors for consensus. Instead, Lamport’s paper tried to generalize the problem of consensus and improve fault-tolerance by having previously connected nodes in the system reconnect with the consensus protocol.

According the synopsis of Lamport’s paper, Paxos was the first time a “clearly stated correctness condition and a proof of correctness” [Parttime Parliament]. By comparison, the Echo system [1] did not provide any mathematical proof to guarantee its consensus protocol. It did provide a stated setup that takes a new look beyond primary and secondary databases and identify key terms that continue to be prevalent in material about distributed systems today.

Unfortunately, the key idea of the Paxos algorithm is lost upon the reception and critic of Lamport’s original paper. While it is sound mathematically in its ideas and principles, Lamport chose a more creative approach to the paper by enclosing it in a story about the Paxon society where a quorum of part-time parliaments, faulty politicians, must concede to pass ballots for the governing of society. The paper also contains mathematics that engineers at the time tended to stay away from. Lamport tried to explain his proof more clearly in “Paxos Made Simple” [Paxos Made Simple], but Paxos would not see a resurgence till the 21st century.

## **Research and Development**

Since the initial presentation of Paxos, there have been many papers published on modifications to the Paxos algorithm. The Paxos specification itself is not defined explicitly, so researchers mix the concepts with other algorithms. As with any algorithm, optimization was the first step towards adoption, so Lamport wrote another paper with “Fast Paxos” [Fast Paxos]. Paxos by itself has to have two round trips to establish a single ballot proposal (as we will explain in Section II), so by decreasing these messages to only a part of the network will make progress in consensus faster for the majority of nodes deemed non-faulty and call for other nodes to make consensus or proposals later.

Other implementations of Paxos include Cheap Paxos, Multi-Paxos, Stoppable Paxos, and many more [cornell]. All of these consider optimizations on speed and performance. However, there is one version of Paxos that was addressed but not detailed in the original specification—Byzantine Paxos.

During the development of the Echo distributed filesystem, the question of failure was followed by the question of failure with a variable of sabotage. The algorithm that recovers a fault-tolerant system from malicious processors was one of the key features desired in early distributed systems development. While redundancy was always a key factor, recovering consensus from potential malicious processors benefitted an algorithm greatly to its favor and adoption. Echo states that while failing processors and malicious processors operate in the network clients will just see other clients “making strange but valid operations” but might fall int o a denial of service to other clients [echo].

In Paxos’s case, the paper “Byzantizing Paxos by Refinement” [Byzantine Paxos] delivers proofs to satisfy that a Byzantine Paxos algorithm, named *BPCon*, can accept *f* faulty acceptors with 2­*f* + 1 nonfaulty processors and maintain actual consensus. The way BPCon does this is a big development upon regular Paxos, but it does serve to bring the topic considering the implementation scenario of this paper—smart meter privacy reporting.

Other notable of Paxos development can be attributed to its spread in distributed system products. Nearly all the major cloud providers such as Google, Microsoft, and Amazon implement Paxos in their backend systems to provide maximum uptime. Almost every single implementation had Leslie Lamport on the development team. Every other database system typically has Paxos as feature for developers to use and implement such as Apache Zookeeper and Cassandra.

# Explanation of the Paxos Algorithm

In this section, we will explain the fundamentals of the Paxos algorithm in the complexity of a computer science undergraduate networks class. By learning the fundamentals in this way, readers can move onto the material referenced by this paper. Although this explanation is not substantial enough for a course in distributed systems, it can provide an gateway example into the field.

## **Concept**

Paxos is a distributed, peer-to-peer algorithm that tries to achieve consensus with three basic roles. Paxos attempts to provide consensus to clients, *processors*, knowing that these processors are faulty and may stop broadcasting at any time. Consensus is a problem of distributed systems to make sure every processor has the correct data value recorded in memory.

Paxos uses a ledger of *ballots* (also called suggestions, proposals, or messages) to keep track what ballots have been voted on and which ballots have been rejected. Ballots in the ledger come with a sequential, unique id that can be verified to be checked by all processors. Ballots also come with the unique id of the processor that proposed them and a data value to be accepted.

Before any discussion of Basic Paxos, two things are certain:

* Peers work at any speed and may fail by stopping or restart at any point before, during, and after consensus.
* Ballot messages may be lost or duplicated, but, at least in Basic Paxos, never corrupted.

## **Roles**

Paxos has three basic roles:

1. ***Proposers:*** Any peer processor can propose a ballot message to be accepted. This ballot message is tagged with the Proposer’s unique id, a data value, and a unique, sequential id for the ballot message itself.
2. ***Acceptors:*** Any peer processor can accept or reject a ballot message. The basis for rejecting or accepting a message is the algorithm run to produce a result. In other words, any algorithm to be implemented and its result communicated to peers.
3. ***Learners:*** Proposers can play the role of the learner or assign one to a peer. Learners are to see if consensus has been reached by querying the majority of the *quorum*—all current operational peers in the system.

We can then say *consensus* has been reached when the majority limit of peers accept the data value of a ballot message by a Proposer.

This is the basic structure of Paxos, but for the proof to be complete the following conditions also have to be true:

* *Ballot messages must have an id greater than the last ballot message sent.* The unique, sequential id of the ballot message must be greater than the one before otherwise peers will progress on consensus on ballots already rejected or accepted.
* *If a ballot is rejected, a peer can propose its value again, but with a higher sequential id of the original ballot message.* This statement reiterates the last but adds the fact that a previous data value can be proposed again by either the same Proposer or another one.
* *The majority limit for consensus and the quorum of processors must be known ahead of time.* In other words, information about peers in the network and the majority must be known in order for consensus to occur.
* *Values that are accepted may not be transmitted to all learners.* Paxos does not guarantee that all learners, and thus all peers, will know that there has been an accepted value. This is possible since peers may leave the network and reenter when consensus has already been reached.

Other forms of Paxos address these issues to achieve goals based on reliability, availability, and scalability.

## **Message Conflict**

Let us analyze the state where peers have exited from the network, make a consensus of values by their own Proposers, and then reconnect to the network only to see that consensus had already been reached. Such an example can be shown by extension of Appendix A.

Let us suppose that D had two neighboring peers E and F that also disconnected from the network. Then, D proposes a ballot, say (4, “C-”) to E and F. E and F accept the message, and so consensus was reached with the value “C-”.

Later, D, E, and F reconnect with A, B, and C. However, A, B, and C reached consensus during the disconnect and picked the value “A+”. The original Proposer A will send the ballot it sent before as consensus, (3, “A+”), but D will reject it automatically since the sequential id number of the ballot is lower than the one it last received.

Therefore, A will have to propose the value with a higher sequential id to *each peer in the network* in order to get its value “A+” accepted by at least the majority again (A could pick only two peers but will go for more in Basic Paxos). However, there is lost time and performance due to the previous disconnect. Each proposal and acceptance round takes at least two round trips to communicate, but the guarantee is in the reliability that the disconnected networks reach consensus at the end.

##### Appendix A: Example of Paxos protocol

* Three nodes A,B,C, D. Maximum to reach consensus is 2.
* A proposes Message(2, “A+”)
* B, C, D receive. Only B accepts.
* Message approval failed. Await next message.
* A proposes Message(3, “A+”). D cannot receive due to system restart.
* This time B, C accept the message. Consensus was reached.
* D reconnects. If it tries to propose Message(2, “A+”) it will be rejected. It can try to send another message like Message(4, “B”).

##### References

1. Birrell, A., Hisgen, A., Jerian C., et al. “The Echo Distributed File System”. 1993.
2. Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. ACM, 2007.
3. Chen, P., Yang, S., McCann J., et al. (2018). Detection of false data injection attacks in smart-grid systems - IEEE Journals & Magazine.
4. Leslie Lamport, "Paxos Made Simple", ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), December 2001
5. Leslie Lamport, “Fast Paxos” *Distributed Computing.* 14 April 2006.
6. Leslie Lamport, “The Part-Time Parliament”, ***ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.***
7. Renesse, R. Altinbuken D., “Paxos Made Moderately Complex”, *ACM Computing Surveys*, Vol. 47, No. 3, Article 42, Publication date: February 2015.
8. T. F. Rezende, P. Sutra, R. Q. Saramago, L. Camargos, "On Making Generalized Paxos Practical", 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), March 2017
9. Tom Cocagne, Essential Paxos, January 2013, Github Repository, <https://github.com/cocagne/paxos>