Verifying Paxos in UPPAAL

Venkat Margapuri and Mitchell Neilsen

Department of Computer Science

Kansas State University

marven@ksu.edu, neilsen@ksu.edu

Abstract

Consensus in a distributed system i.e. the idea that all of the nodes in the network agree upon a common data value, typically proposed by one of the nodes in the network is critical to achieve for the efficient functioning of a distributed system. Paxos is a key algorithm in the field of distributed systems aimed at achieving consensus despite the lack of a global clock and possibility of faulty nodes in the network. Briefly, the algorithm relies on three main entities i.e. Proposer, Acceptor and Learner where a majority of the acceptors is required to agree on a value proposed by a proposer for consensus. The paper describes the implementation of the Basic Paxos algorithm in UPPAAL, an environment for modeling, validation and verification of real-time systems represented using timed automata. UPPAAL provides a real-time simulator that provides a simulation of the execution and supports the verification using Computational Tree Logic that is leveraged to verify the safety and liveness properties of the implemented model i.e. the Paxos algorithm, in this case. The verification illustrates that the behavior of the implemented model aligns with the behavior of the proposed algorithm and proves that the model satisfies the properties of the algorithm.

1. Introduction

The notion of distributed consensus, defined as the phenomenon in which all of the nodes in a distributed system agree on a value upon which the computation depends is critical to the effective functioning of a distributed system. A classic example of such a scenario is observed in distributed databases where each of the nodes needs to agree on whether to commit or abort a specific transaction [2]. The example is relevant in some of the modern day popular file systems such as Google File System, Amazon Dynamo and Apache Zookeeper. The manner in which such a consensus is achieved is by holding a voting procedure where each of the nodes chooses a value and agree upon the value chosen by majority of the nodes. However, this process is not fail-safe owing to node failures in a distributed system. It is fair to say that a consensus by majority is not possible if a majority of the nodes in the system fail.

Paxos [3] is an algorithm developed by Leslie Lamport to achieve distributed consensus while withstanding node failures in an asynchronous distributed network. The algorithm handles processes that run concurrently without shared memory, where processes may crash and later recover, and messages may be lost, delayed, reordered, or duplicated [4]. While different variants of Paxos such as Basic Paxos, Multi-Paxos and variants that reduce message delay or add preemption are available, the current work considers the study, implementation, verification and verification of the Basic Paxos algorithm. Simply put, the other variants of Paxos are modifications made to the Basic Paxos algorithm.

The implementation, verification and validation of the algorithm is performed using UPPAAL, an integrated tool environment meant for modeling, simulation and verification of real-time systems represented using timed automata. The tool is a result of the collaboration between Basic Research in Computer Science at Aalbor University, Denmark and the Department of Information Technology at Uppsala University, Sweden. The tool consists of three main parts:

1. Description Language: A non-deterministic guarded command language with data types.
2. Simulator: A validation tool that enables examination of possible dynamic executions of a system during the design phase.
3. Model Checker: A tool that checks invariant and reachability properties by exploring the state-space of a system.

Safety and Liveness properties of the model are verified using the model checker. Safety properties are properties of the form, “something bad will never happen” [5] and Liveness properties are properties of the form, “something will eventually happen” [5]. The properties for verification are expressed in Computation Tree Logic (CTL) that consists of path formulae and state formulae. State Formulae describe individual states and path formulae quantify over paths or traces of the model that are better classified as Reachability, Safety and Liveness [5].

In the remainder of the paper, Section 2 specifies related work, Section 3 provides a description of the Basic Paxos algorithm, Section 4 discusses the UPPAAL model developed for the algorithm, Section 5 describes the formal verification and Section 6 concludes the paper.

1. Related Work

Leslie Lamport first put forth Paxos algorithm in the work, ‘The Part-Time Parliament’ [6]. When it was deemed overly complex to understand, another work, ‘Paxos Made Simple’ [3] was put forth to explain the concepts of Paxos in an easily understandable manner. The Basic Paxos algorithm was explained in detail in [3] and used as the basis for the model developed in the current work. From the perspective of model checkers, the work, ‘Model Checking Paxos in Spin’ [7] where focus is on the implementation of a Promela model for the Paxos algorithm in Spin could be considered closest to the current work.

In terms of UPPAAL, the work ‘A Tutorial on UPPAAL’ [5] is a descriptive tutorial on the tool. The work helps understand the inner workings of UPPAAL and provides a well-rounded explanation of the multitude of concepts and features available in UPPAAL. In addition, several examples are available to help understand better on the website, ‘<http://www.uppaal.org/>’.

1. Paxos Algorithm

A distributed system is a collection of nodes working in tandem to achieve a common goal. As nodes in the system work together, it is often required that, they all agree on a common value to perform a certain task. This phenomenon of several nodes in the system agreeing on a common value is defined as the notion of consensus. Two of the major factors concerning consensus in distributed systems are node failures and lack of a global clock. [9] shows the impossibility of attaining distributed consensus even with one faulty node in an asynchronous network.

Paxos is an algorithm introduced to address the issue of consensus in a distributed system while being resilient to node failures and the lack of a global clock. The algorithm is expressed in terms of three different roles namely, proposer, acceptor and learner. The proposers are nodes that propose a value, acceptors are nodes that accept a value from a set of proposed values and learners are nodes that learn the value chosen by the acceptors from the set of proposed values. There is a clear demarcation in terms of the tasks each of the roles is required to perform. Hence, any given node in the network can take on any or all of the roles as long as the tasks for the role are performed correctly. Needless to mention that each of the nodes in the network has global knowledge of the presence of other nodes in the network and the roles that they intend to perform. The algorithm consists of two main steps: Prepare and Accept. The steps are explained as follows:

**Prepare Request:**

1. A proposer in the network proposes a unique ID (typically greater than the previous ID accepted by the acceptors in the network) by sending it to all or at least a majority of the acceptors in the network. In the event there exist multiple proposers in the network, it is critical to ensure that multiple proposers do not propose the same value. The use of timestamp or assigning a range of numbers that each of the proposers can propose are good techniques to do so.
2. Upon receiving the ID proposed by a proposer, an acceptor checks for one condition that is, is the currently proposed ID less than the previously accepted ID. If yes, ignore the currently proposed ID, else, send a response to the proposer indicating that the willingness to accept the proposed ID and the willingness to ignore any proposed ID of lower value.

**Accept Request:**

1. The proposer, upon receiving responses from the acceptors checks to see if a majority of acceptors are willing to accept the proposed ID. If so, the proposer prepares an Accept request that consists of the proposed ID and the value that the proposer wants the acceptors to record. The Accept request is sent out to all or at least a majority of the acceptors.
2. The acceptor, upon receiving the Accept request checks for one condition that is, is the ID in the Accept request an ID to be ignored. (A typical scenario in which an ID is ignored is when a different proposer proposes a higher ID between the time the acceptor has expressed their willingness to accept a certain proposed ID in the Prepare Request and the receipt of the Accept request). If yes, ignore, else, respond to the proposer with the proposed ID and value.
3. In addition, the acceptor sends a message to the learners informing them about the ID and the value on which the system currently reached consensus. A key point to remember is that consensus is on the value and not on the proposed ID. The proposed ID is only to identify the proposer that proposes the value.

As the algorithm is executed further upon reaching consensus for the first time on a value, it is to be noted that the ID on which a majority is attained could change but the value cannot change. In other words, the proposer upon receiving a majority to their Prepare request could send out an Accept request with the proposed ID and a value different from the value on which consensus is attained but the acceptors would only accept the change in ID but not value and the acknowledgement i.e. the Accept response from the acceptor to the proposer would be sent with the new ID and the old value on which consensus is first attained. In addition, the algorithm is guaranteed to reach consensus with n acceptors up to f = simultaneous failures, but only if learners have enough time to take a decision (i.e. to detect a majority) [7].

1. UPPAAL Model for Paxos

The developed UPPAAL model closely follows the algorithm stated in section 3. For the purposes of the experiment, a model with 8 nodes is considered of which 2 nodes i.e. nodes 0 and 1 act only as proposers, 3 nodes i.e. nodes 2, 3 and 4 act only as acceptors and all 8 of the nodes act a learners but nodes 5, 6 and 7 act only as learners. While several features of UPPAAL are used to build the model, seven features are key to the model. They are declarations, broadcast channels and arrays, Record Types and Guards, Committed Locations and Rate of Exponential. The features are briefly explained to ensure that the reader is provided with enough knowledge to understand the model. In case a much more detailed explanation is desired, please refer to [5].

**Declarations:** There are three main types of declarations, system declarations, global declarations and local declarations. System declarations are used to define each of the processes (nodes) in the system. Global declarations are global variables and values that can be consumed and modified by all of the processes in the system. Local declarations are the values, which are specific to a certain instance in the system.

**Broadcast Channels:** The idea is that a sender can synchronize with any number of receivers over a channel. Syntactically, the channels are defined as ‘broadcast chan c’ where c! sends out a synchronization message and c? helps receive the broadcasted message. It is important to remember that channels in UPPAAL do not necessarily transport data but instead are indicative of an event in the system. It is the responsibility of the nodes to perform the necessary updates upon receiving a signal.

**Arrays:** Arrays are data structures that can hold information. An array can be of a specific size and are allowed for clocks, channels, constants and integer variables.

**Record Types:** Record Types are declared using the ‘struct’ keyword like in C and are similar to classes in object-oriented languages such as Java in the sense that they can hold multiple properties and data types.

**Guards:** A Guard is an expression satisfying a certain condition. It is always side effect free and evaluates to a Boolean. Guards can call a side effect free function that evaluates to a Boolean value and are typically used as gate checks before entry to a state in the model.

**Committed Locations:** Committed Locations are used to ensure that instructions execute atomically and there is no interleaving as instructions are executed. When a process enters a committed location, time does not pass and it must take the next available transition. However, when multiple processes reach a committed location, interleaving may occur. The use of committed locations improves significantly reduces the amount of time needed for verification of the model and hence is regarded as an optimization technique.

**Rate of Exponential:** In the absence of an invariant over time for a given location, the probability of the model leaving the location is distributed according to exponential distribution. It plays an important role in statistical model checking.

The first step to building the model is the declaration of each of the proposers and acceptors in the system. System declarations are used to define each of the nodes in the system. Figure 1 shows the system declarations where 2 proposers and 3 acceptors are defined. Learners are not declared explicitly because the model assumes that each of the nodes in the network acts as a learner.

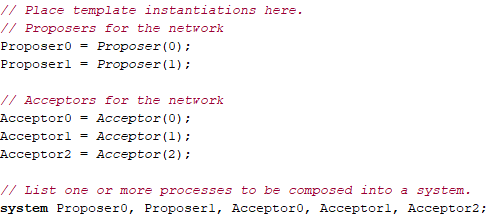


Figure 1: System Declarations Showing Nodes in the System

Figure 1 shows the system declarations where two proposers and three acceptors are defined. Learners are not declared explicitly because the model assumes that each of the nodes in the network acts as a learner. Once the nodes are setup, the next step is to define the behavior of the nodes in the network i.e. proposers, acceptors and learners in this case. Separate templates for Proposer and Acceptor are created along with several global declarations and are as described below.

**Declarations:** In order to keep track of the state i.e. ID and value of each of the nodes in the network, Nodes, a struct which consists of two properties, paxosID and value is created. Dummy values of -1 and 9 are assigned to the paxosID and value of each of the nodes initially. The paxosID property tracks the ID on which consensus is currently attained and the value property is set to the proposed value if a consensus is reached, else, remains at 9. For the purposes of this experiment, the value proposed by the proposer that reaches majority on an ID is set to 4. Once the value of the acceptors is set to 4, further consensuses will only result in a change of the paxosID but not value. Other key data structure declarations include ‘ValueAgreed’ and ‘AccRejAcceptor’. ValueAgreed is an array in which acceptors toggle a flag to indicate that consensus is attained on a value. It helps proposers know that a consensus is reached in a previous execution and that proposals with a new value are no longer accepted. AccRejAcceptor is also an array in which acceptors change the value to that of the proposed ID to indicate their willingness to accept a proposed ID. The proposer, upon receiving responses from each of the acceptors checks this array to gain knowledge of the number of acceptors willing to accept a proposed ID. Initially, both ‘ValueAgreed’ and ‘AccRejAcceptor’ have a value of 0 assigned. The definitions of each of the data structures discussed above are as shown in Figure 1.

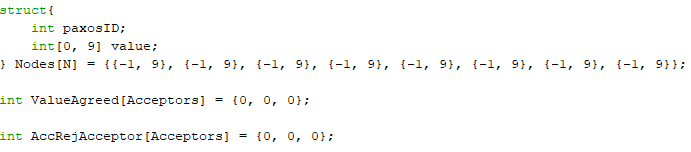


Figure 2: Key Data Structures for the Paxos Model

**Proposers:** In order to ensure that there is no collision of the IDs selected by each of the proposers, proposer 0 is modeled to pick only even numbered IDs and proposer 1 is modeled to pick only odd numbered IDs. As shown in Figure 3, the execution starts from the ‘Start’ state where the proposer creates an ID using the function ‘PrepareProposedID’ and sends out a proposed ID to each of the acceptors i.e. acceptors 0, 1 and 2 over a broadcast channel named send which takes in three parameters, proposer ID (not to be confused with the proposed ID. It is either 0 or 1 for proposers 0 and 1 respectively), the acceptor ID (either 0, 1 or 2) and a flag which is used to distinguish between a Prepare request and an Accept request (0 for Prepare request and 1 for Accept request). Simply put, send[0][0][0] indicates that proposer 0 sends a Prepare request to acceptor 0.

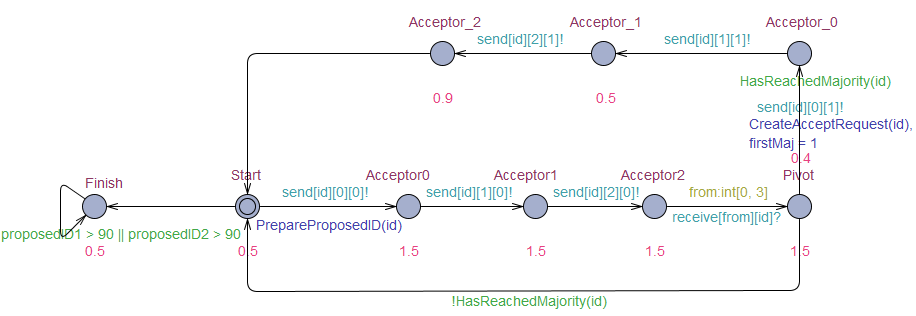


Figure 3: Proposer Template

Upon sending out Prepare requests to each of the acceptor nodes, the proposer node waits for a response from the acceptor nodes. The acceptor nodes send out a response on a broadcast channel named ‘receive’ which takes two parameters, the acceptor node and the proposer node, and hence, the proposer node is setup to receive a response (synchronize) on the receive channel as shown in the ‘Pivot’ state. Upon receiving responses from the acceptor nodes, the proposer nodes checks to see if a majority of the nodes have approved of the proposed ID by executing a guard named ‘HasReachedMajority’. In the event that a majority of the acceptor nodes have approved, the proposer goes on to create an Accept request using a function named ‘CreateAcceptRequest’ and distributes it to each of the acceptor nodes, else, the proposer goes back to the ‘Start’ state and repeats the process again by proposing a different ID. When the proposed ID of a proposer goes past 90, the proposer is sent to the ‘Finish’ state. This is done solely to limit the state space of the model to aid in verification. The numbers below each state are the rates of exponential which are used primarily during the verification of the model and don’t hold any significance in terms of the implemented business logic.

**Acceptors:** The template for the acceptors is designed to depict scenarios in which a node is functional and non-functional. Probabilistic branching i.e. the idea of making stochastic interpretations based on the probabilistic choices assigned to the template is a feature available in UPPAAL and leveraged for this purpose.

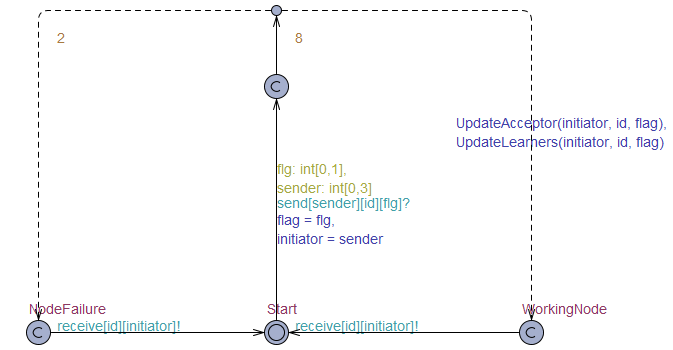


Figure 4: Acceptor Template

As shown in Figure 4, the left and right branches are assigned probabilistic choices of 2 and 8 respectively. In simple words, the template takes the left branch 2 out of 10 times and the right branch 8 out of 10 times. As and when a signal on the ‘send’ channel is received, the acceptor, if alive (right branch), makes the necessary updates using the ‘UpdateAcceptor’ function. As part of the function, the acceptor observes the flag value to distinguish between a Prepare and Accept request, and acts accordingly. In case of a Prepare request, if the acceptor is interested to accept the proposed ID, it changes the value in the ‘AccRejAcceptor’ array to the proposed ID. In the event that it has also agreed to a proposed value in a previous execution, the acceptor flips the value in the ‘ValueAgreed’ array to inform the proposer of this. In theory, the proposer should be aware of the agreed upon value since it is also a learner and the acceptors every other node including the proposers upon a consensus. In case of an Accept request, the acceptor updates the properties of PaxosID and value in the Nodes struct to the currently proposed ID and value respectively. However, in the event of a node failure (left branch), no updates are performed to mimic the behavior of a failed node. Observe that the states are marked with a ‘C’. The reason is that the states are made to be committed states to ensure that the template takes the next available transition and does not wait indefinitely in a state. A signal is broadcast over the ‘receive’ channel to indicate the proposer of the acceptor’s response.

**Learners:** In the network, nodes 5, 6 and 7 act only as learners while every other node in the network acts also as a learner. An acceptor, upon approving an Accept request to reach consensus, sends out a message informing every other node in the network of the consensus. The ID and the value on which consensus is reached are sent as part of the message. It helps the proposers and acceptors not part of the consensus stay informed about the state of the network. In terms of the model, the acceptor updates entries in the Nodes struct for each of the nodes with the ID and value on which consensus is reached. While it helps all of the acceptors remain in sync, it makes the proposers aware of the new ID for consensus that in turns helps the proposer choose a greater ID to attain consensus. Smaller IDs, even if pitched by the proposers, are rejected by the acceptors resulting in wasted attempts and an unnecessary increase in search space. Figure 5 is a code snippet used to update the Nodes struct for each of the nodes specifically acting as learners in the network.

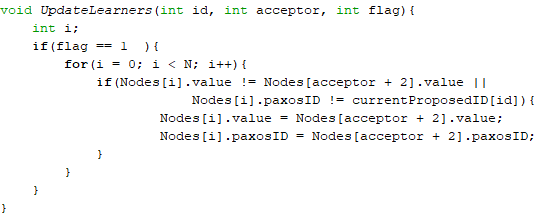


Figure 5: Acceptor’s Updates to the Learners in the Network

1. Formal Verification

In order to verify the model, the UPPAAL Verifier, built into the tool is used. Verification is performed on the three properties deemed essential for consensus in [9]. In addition, other liveness properties are used for verification to show that the model eventually converges. In UPPAAL, all of the properties for verification are written in CTL and the pass/ fail of the properties determines the behavior of the model.

**Property 1:** Only a value that has been proposed may be chosen [9].

**Logic:** A[](Nodes[2].value == 4 || Nodes[2].value == 9)

**Result:** Pass

**Justification:** The above property in plain words translates to, ‘always, it is the case that the value of Node 2 i.e. acceptor 0 is set to either 4 or 9’. Recall that the acceptors are initially assigned a dummy value of 9 until a consensus is reached. For the purposes of this experiment, the consensus value chosen is 4. Upon reaching consensus, the value should invariably change to 4 and should remain at 4 in perpetuity. While verification is done on Node 2, it holds true for any node, be it acceptor, proposer, or learner in the network.

**Property 2:** Only a single value is chosen [9].

**Logic:** E<>((ValueAgreed[0] == 1 && Nodes[2].value == 4) && (ValueAgreed[1] == 1 && Nodes[3].value != 4) && (ValueAgreed[2] == 1 && Nodes[4].value != 4))

**Result:** Fail

**Justification:** The data structure ‘ValueAgreed’ is used to record the attainment of consensus. A value of 1 in the data structure means that a node (acceptor) has agreed on a consensus value. The property in plain English says, ‘eventually on some path, it is true that consensus among nodes is attained and one of the nodes agree on a value of 4 whereas the other acceptors in the network agree on a value which is not 4’. This is not valid since all of the nodes are supposed to agree on a common value and rightly, the property fails which proves that such a scenario is impossible.

**Property 3:**  A process never learns that a value has been chosen until it has actually been chosen [9].

**Logic:** E<>((Nodes[2].value == 9 && Nodes[3].value == 9 && Nodes[4].value == 9) && (Nodes[5].value == 4 || Nodes[6].value == 4 || Nodes[7].value == 4))

**Result:** Fail

**Justification:** In the model, nodes 5, 6 and 7 specifically act as learners i.e. only when consensus is attained, they should be informed. The same is verified using the logic which in plain words translates to, ‘eventually on some path, it is the case that one of the learners is aware of a consensus even before any of the acceptors are aware of a consensus themselves’. The property is deemed as a failure by UPPAAL that shows that a learner can never be aware of a consensus before the acceptors have agreed on a value.

**Property 4:** A consensus is not reached unless a majority is attained.

**Logic:** E<>(firstMaj == 0 && (!HasReachedMajority(1) || !HasReachedMajority(0)) && (Nodes[2].value == 4 || Nodes[3].value == 4 || Nodes[4].value == 4)

**Result:** Fail

**Justification:** The property translates to, ‘eventually on some path, it is the case that a majority has never been reached on any of the IDs proposed by the proposers, yet at least one of the acceptors has agreed to a value’. The property fails since a majority should be attained at all times before a consensus.

**Property 5:** Consensus is reached if a majority is attained and the value agreed upon is the value proposed by the proposer.

**Logic:** E<>((HasReachedMajority(1) || HasReachedMajority(0)) && (Nodes[2].value == 4) && (Nodes[3].paxosID == 4) && (Nodes[4].paxosID == 4))

**Result:** Pass

**Justification:** The property translates to, ‘eventually on some path, upon reaching a consensus on at least one of the proposers, a consensus is attained on the proposed value of 4’.

1. Conclusion

The work throws light on the use of UPPAAL, a model checker in proving Paxos that is regarded as a complex algorithm to prove. The focus of the work is to demonstrate the capability of UPPAAL to build complex models that mimic real world systems, and verify the safety and liveness of the models to provide certain guarantees concerning the behavior of such systems when implemented in the real world. In fields such as academia where there is no shortage of novel ideas, model checkers are a boon to both students and faculty wishing to experiment and demonstrate their ideas without requiring any hardware components, thus lowering the cost of prototyping significantly.

References

[1] Fischer, M. J. (1983, August). The consensus problem in unreliable distributed systems (a brief

survey). In *International conference on fundamentals of computation theory* (pp. 127-140). Springer,

Berlin, Heidelberg.

[2] Gray, J. A Discussion of Distributed Systems. Research Report RJ2699, IBM, Sept., 1979.

[3] Lamport, L. (2001). Paxos made simple. *ACM Sigact News*, *32*(4), 18-25.

[4] Chand, S., Liu, Y. A., & Stoller, S. D. (2016, November). Formal verification of multi-Paxos for

distributed consensus. In *International Symposium on Formal Methods* (pp. 119-136). Springer,

Cham.

[5] Behrmann, G., David, A., & Larsen, K. G. (2004, September). A tutorial on uppaal.

In *Formal methods for the design of real-time systems* (pp. 200-236). Springer, Berlin,

Heidelberg.

[6] Lamport, L. (2019). The part-time parliament. In *Concurrency: the Works of Leslie*

*Lamport* (pp. 277-317).

[7] Delzanno, G., Tatarek, M., & Traverso, R. (2014). Model Checking Paxos in Spin. *arXiv*

*preprint arXiv:1408.5962*.

[8] Coccoli, A., Bondavalli, A., & Simoncini, L. (2000). Consensus in Asynchronous

Distributed Systems. In *Proc. of the 5th Int. Conf. on Integrated Design and Process*

*Technology (IDPT’00)*.

[9] Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985). Impossibility of distributed

consensus with one faulty process. *Journal of the ACM (JACM)*, *32*(2), 374-382.