Tutorial Summary: Paxos Explained from Scratch

Hein Meling and Leander Jehl

University of Stavanger, Norway

Abstract. Paxos is a flexible and fault tolerant protocol for solving the consensus problem, where participants in a distributed system need to agree on a common value. However, Paxos is reputed for being difficult to understand. This tutorial aims to address this difficulty by visualizing Paxos in a completely new way. Starting from a naive solution and strong assumptions, Paxos is derived in a step-wise fashion. In each step, minimal changes are made to the solution and assumptions, aimed at understanding why the solution fails. In this manner, a correct solution that corresponds to Paxos is eventually reached.

1 Introduction

Paxos is a flexible and fault tolerant consensus protocol that can be used in applications that need to agree on a common value among distributed participants. Paxos was proposed by Lamport in his seminal paper [1] and later gave a simplified description in [2]. Paxos can be used to solve the atomic commit problem in distributed transactions, or to order client requests sent to a replicated state machine (RSM). An RSM provides fault tolerance and high availability, by implementing a service as a deterministic state machine and replicating it on different machines. Paxos is relevant because it is often used in production systems such as Chubby and ZooKeeper [3, 4] among many others. Understanding Paxos is important because it reveals the distinction between a strongly consistent RSM and a primary-backup system.

Both before and after its publication in [1], Paxos attracted much attention for its unorthodox exposition in the form of a fictional parliamentary system, supposedly used by legislators at the Greek island of Paxos. But the scientific contribution was also significant; it provided a new way to implement RSMs, and proved that the protocol guarantees that participants make consistent decisions, irrespective of the number of failures. Clearly Paxos cannot always make progress, e.g. during network partitions, as was shown in [5]. But perhaps most important, Paxos was described in a flexible and general way, ignoring many technical details. This made it an excellent foundation for further research into RSM-based protocols [6–9], aimed at supporting different failure models, wide-area networking, to improve latency, and so on. The fact that these protocols build on the Paxos foundation, which has been formally proven, makes it much easier to reason about their correctness through step-wise modifications of Paxos.

With this powerful foundation that Paxos offered, came also a challenge: the flexible description made it harder to understand. This remains true to this day, even as numerous papers have been written aimed at explaining Paxos for system builders [10, 11] and more generally [12, 13]. These and other papers are still challenging for students and others to understand without significant efforts.

R. Baldoni, N. Nisse, and M. van Steen (Eds.): OPODIS 2013, LNCS 8304, pp. 1–10, 2013.

The aim of this tutorial is to explain Paxos from the bare fundamentals by deriving a Paxos-based RSM in a step-wise and pictorial manner. We start with a non-replicated service that we want to harden with fault tolerance and high availability. That is, the server must be replicated. Initially, we make unrealistic assumptions about the environment and propose the simplest protocol that we can imagine to coordinate the server replicas to ensure that they remain mutually consistent, and explain why the protocol is insufficient. Then in each step, minimal changes are introduced to the coordination protocol aimed at understanding why each protocol fails. Continuing, we finally reach a correct protocol that corresponds to a Paxos-based RSM.

Our objective is that you understand that many seemingly intuitive approaches do not work and why. Having read this tutorial, we hope that you will gain appreciation for Paxos' contribution, and perhaps put you in a better position to read the Paxos literature.

2 A Stateful Service: Assumptions and Notation

We will explain Paxos starting from a simple stateful service that should be made fault-tolerant and highly available. Initially the service is implemented by a single server that receives requests from a set of clients, processes the requests, updates its state, and replies back to the clients. This pattern is visualized as a message sequence diagram in Fig. 1, where server S_1 processes requests from clients, C_1 and C_2. Further notation is explained below.

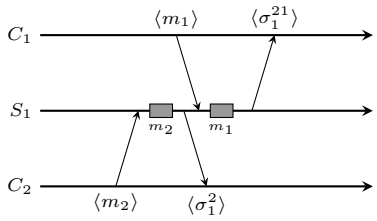


Fig. 1. Solution *SingleServer*: A single server can order and process requests from several clients

Notation. A request message received by the server causes a state transition affecting the current state of the server. The outcome of processing requests sent to the server depends on its current state. A box on the timeline is meant to illustrate that a state change has taken place, caused by the processing of some message m_i.

A common assumption also adopted here, is that requests from different clients are unrelated, and the order in which they are executed is irrelevant. Clearly, requests from the same client should be executed in sending order. We use σ_i^{kl} to denote the local state of server S_i after having processed messages $m_k m_l$, in that order. We ignore the server index and write σ^{kl}, when the origin is irrelevant. In our examples, the reply sent to clients is determined by the server's state, denoted $\langle \sigma_i^{kl} \rangle$. In practice, the reply is usually not the server's state, but rather some value computed from the server's state.

Single Server. In the single server case shown in Fig. 1, it is easy to see that the two clients observe a consistent reflection of the server's execution of the two requests. It is easy for the server to determine an ordering for the client requests that it receives. However, implementing the service with a single server is not fault-tolerant.

3 Fault Tolerance with Two Servers

As a first attempt at improving the fault tolerance and availability of our service, we can add one server to the system, under the assumption that if one of the servers fail, the other can take over and service client requests. This architecture is frequently used, and is called primary-backup. In our first naive solution we use two servers without coordination between them; i.e. the clients simply send their requests directly to the two servers, as shown in Fig. 2. However, as is apparent from this diagram, the two requests can be processed in different orders at the two servers, e.g. because of message delays: m_1m_2 at S_1 and m_2m_1 at S_2, resulting in deviating server states. We say that the servers become inconsistent. This inconsistency is also exposed to the clients: C_1 observes possibly inconsistent replies σ^1 and σ^{21}, while C_2 observes replies: $\sigma^2\sigma^{12}$.

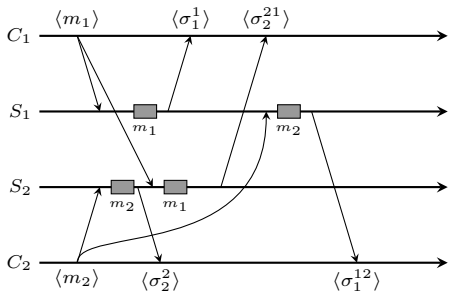


Fig. 2. Problem: Two servers cannot order messages from several clients without coordination

Our first solution to coordinate among the servers is to let one server be leader, also called the primary. The leader simply sends an *accept* message to the other server and executes the request locally. The accept message $\langle \text{ACC}, m_i, j \rangle$ is used to tell the other server that m_i should be executed as the jth request, where j is a *sequence number*. This approach is illustrated in Fig. 3. It is easy to see that both servers remain consistent and that replies to clients are also consistent, since σ^2 is a prefix of σ^{21}. Since the service is implemented as deterministic state machine, processing a request results in a unique state transition. Therefore σ_1^2=σ_2^2 and σ_1^{21}=σ_2^{21}.

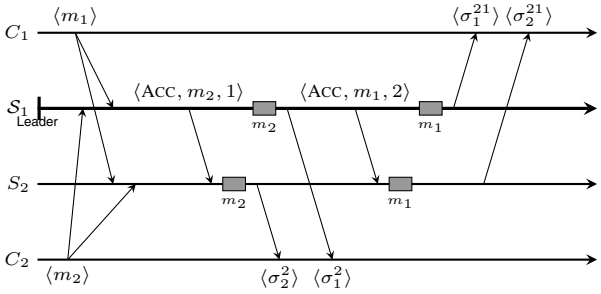


Fig. 3. Solution *SendAccept*: Leader ($\mathcal{S}_1$) sends an accept message to the other server telling it the order in which the messages should be processed

This approach works fine as long as messages are not lost. However, if $\langle \text{ACC}, m_2, 1 \rangle$ in Fig. 3 is lost, then S_2 gets stuck and cannot process the next message, m_1.

The solution to this problem is simply to add a lost message detection mechanism. That is, let the leader retransmit its accept message until it is acknowledged by S_2. This solution is shown in Fig. 4, where a *learn* message $\langle \text{LRN}, m_i \rangle$ corresponds to an acknowledgement. This approach allows for the servers to eventually make progress as long as messages are not lost infinitely often.

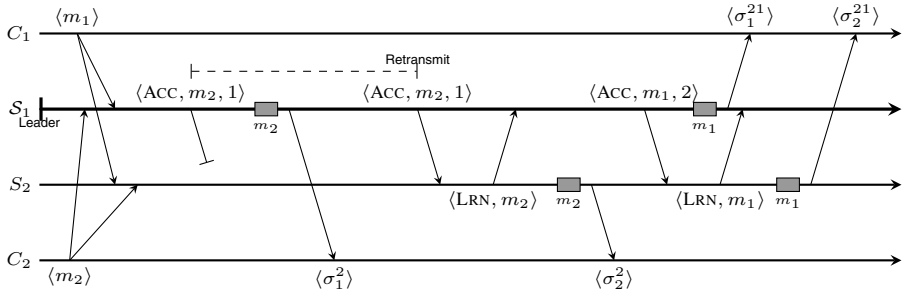


Fig. 4. Solution *RetransAccept*: Retransmit the $\langle \text{ACC}, m_2, 1 \rangle$ message if it does not receive a corresponding $\langle \text{LRN}, m_2 \rangle$ message

4 Server Crashes

We have seen that messages can be lost, and that our *RetransAccept* protocol can fix the problem. However, if one server crashes, the other will wait indefinitely for an accept or learn message. We therefore adopt the rule that once a server crashes, the remaining server continues to serve clients following the *SingleServer* protocol (Fig. 1). With this rule we can see from Fig. 5, that our *RetransAccept* protocol is insufficient. This is because the initial leader (S_1) replies to request m_1 before learning that S_2 has seen its $\langle \text{ACC}, m_1, 1 \rangle$ message, and because S_1 crashes before it can retransmit the accept. Instead S_2 takes over and decides to execute request m_2 before m_1, and thus the two clients observe inconsistent replies; σ^2 is seen by C_2 while σ^{21} is expected.

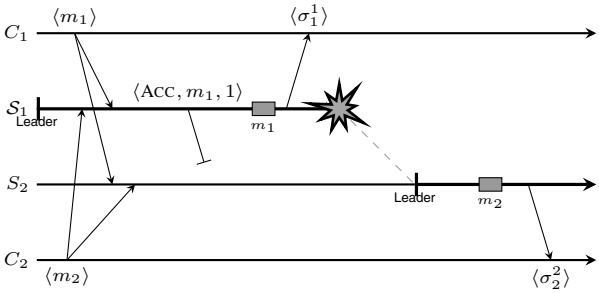


Fig. 5. Problem: The leader crashes after sending reply to client C_1, without ensuring that S_2 has learned about the ordering message, $\langle \text{ACC}, m_1, 1 \rangle$

To solve this problem, we require that the leader wait for the $\langle \text{LRN}, m_1 \rangle$ message before executing the request as shown in Fig. 6, and we also require a retransmission in case of message loss. If S_1 receives the learn message and replies to C_1 before crashing, the ordering information has already been propagated to S_2. If S_1 crashes before sending

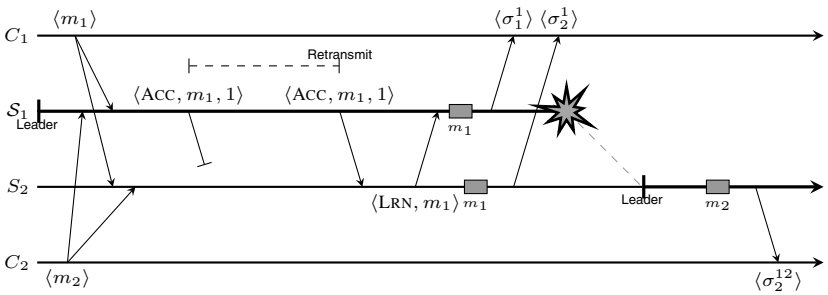


Fig. 6. Solution *WaitForLearn*: The leader waits for $\langle \text{LRN}, m_1 \rangle$ before executing the request m_1

its reply to the client, S_2 may or may not have seen the accept message. If S_2 has seen the accept, S_2 will obey it. If the accept didn't reach S_2, it can decide its own ordering.

5 Network Partitions

So far we have not specified how failures are actually detected. In practice a server is assumed to have failed if it is unresponsive for a given period of time. This is typically done using a timer mechanism, which upon a timeout triggers a *failure detection*. However, identifying a suitable timeout period is difficult in practice, and there is always a chance of false detections due to the stochastic nature of networked systems.

Recall that we adopted the rule to fall back to the *SingleServer* protocol when failure is detected. This rule was intended to allow the service to *make progress* after a server had failed. However, if we cannot reliably detect that the other server *really failed*, then we have a problem, as is illustrated in Fig. 7. Here we see that both servers remain operational, but are unable to communicate due to a network partition. After the failure detection time, both servers fall back to the *SingleServer* protocol and continue to process client requests, exposing the clients to different server states, σ^1 and σ^2. This state divergence violates our desire to remain consistent, especially towards clients. This is since reconciling the state divergence when communication is reestablished would involve rollback on multiple clients, and would quickly become unmanageable.

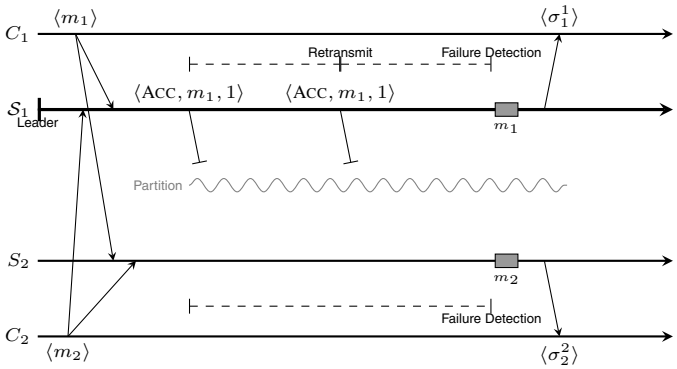


Fig. 7. Problem: Our *SingleServer* protocol can make progress in separate partitions, but it will lead to inconsistencies

A partition is indistinguishable from a crash, e.g. S_2 cannot distinguish between the situations shown in Fig. 5 and Fig. 7. Thus, waiting for a partition to end would also require us to wait indefinitely for a failed server. The solution is to add another server and use the *WaitForLearn* protocol, as shown in Fig. 8. *WaitForLearn* allows a partition to make progress if it contains a majority of the servers. That way we can at least make progress in one of two partitions. In this example we do not consider what needs to happen when the two partitions merge and become one again. We defer this problem until after we solve another problem.

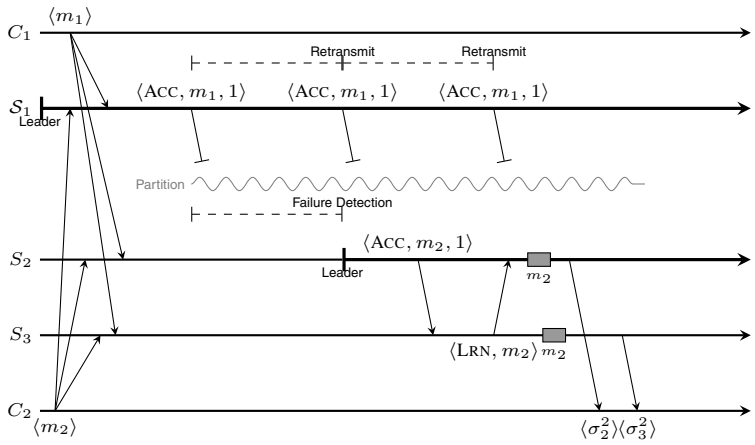


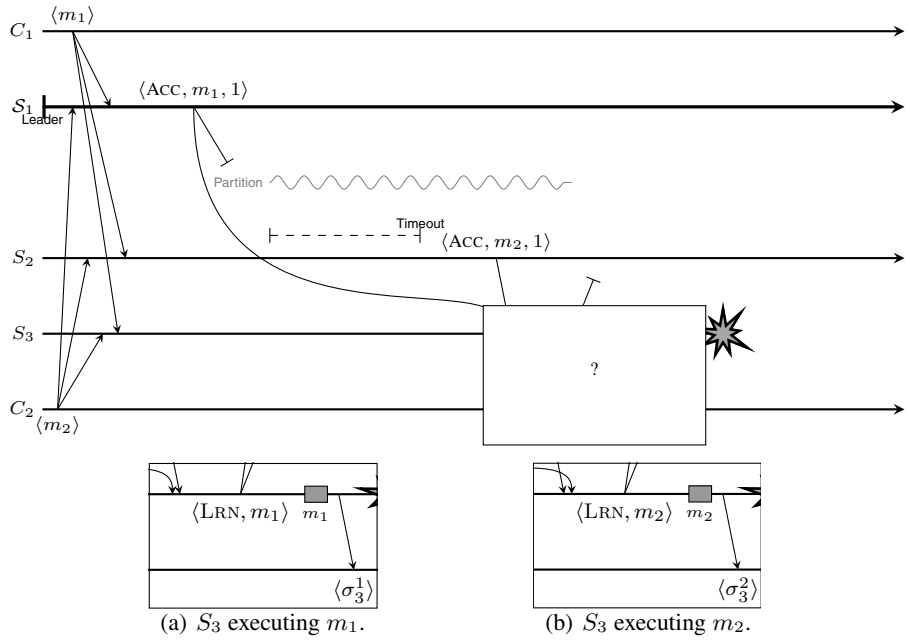
Fig. 8. Solution: Add another server and use the *WaitForLearn* protocol

6 Leader Change

Our *WaitForLearn* protocol tolerates either a crash or a partition. However, a concurrent partition and crash is not handled by our protocol. In cases of false detection, several servers may send out accepts concurrently. In Fig. 9 both S_1 and S_2 send accepts for different messages. If S_3 crashes shortly after receiving these accepts it might have executed one of the requests and sent a reply to the client. In this case it is impossible for the remaining servers (S_1,S_2) to decide whether or not a message was executed before the crash. Fig. 9(a) and 9(b) depict the two possible executions at S_3. These are unknown to the other servers since learn messages may be lost.

The above problem is rooted in the possibility of multiple leaders sending accepts. It can be solved by introducing an explicit leadership takeover protocol. To take over leadership, a server sends a prepare message to the other servers, who acknowledge with a promise to ignore all messages sent by the old leader. Only after receiving promise messages from at least one other server, can the new leader start sending accept messages. This is depicted in Fig. 10. To distinguish between messages from the old and the new leader, we now add the leader's id to accept, learn, prepare, and promise messages.

Furthermore, to ensure that potentially executed requests become known to the other servers, we add those requests to the promise message. Fig. 10 shows an example where no requests have been executed, indicated as $(-)$ in the promise. If the promise contains requests, the new leader resends the accept for these requests, as depicted in Fig. 11.



(a) S_3 executing m_1. (b) S_3 executing m_2.

Fig. 9. Problem: Both S_1 and S_2 sent an accept message to S_3. Since S_3 crashes afterwards, the remaining servers cannot determine whether S_3 executed m_1 or m_2.

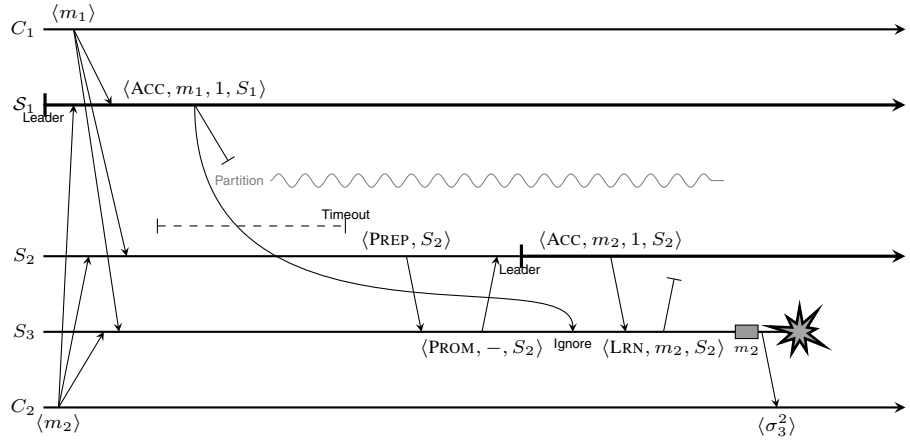


Fig. 10. Solution *LeaderChange*: S_2 announces its wish to become leader by sending a $\langle \text{PREP}, S_2 \rangle$. S_3 replaces S_1 by S_2 as leader and confirms this with a $\langle \text{PROM}, -, S_2 \rangle$ message. S_2 acts as leader after receiving this promise.

Merging Partitions. When two partitions merge, the leader resends accept messages to servers that missed them. However, the merged partition may now have several leaders. For example, when the partition in Fig. 11 ends, both S_1 and S_2 consider themselves leaders. To establish a single leader, we assume a predefined ranking. In Fig. 11,

S_2 assumes leadership and resends accepts to S_1. That is because we assume S_2 to have a higher rank than S_1.

Round Numbers. The above scheme allows S_2 to take over leadership from S_1 because of its higher rank. However, after the server with the highest rank (S_3) has taken over, we will be unable to change the leader. Paxos therefore uses round numbers instead of leader ids. Thus in Fig. 10, we can replace the server id S_1 with round 1 and S_2 with round 2 and so on. With this scheme, S_1 can become leader again by sending a prepare with a higher round, e.g. 4. To avoid that servers send a prepare for the same round, we can preassign rounds, e.g. S_1 can use rounds $1, 4, 7, \ldots$ and S_2 can use $2, 5, 8, \ldots$

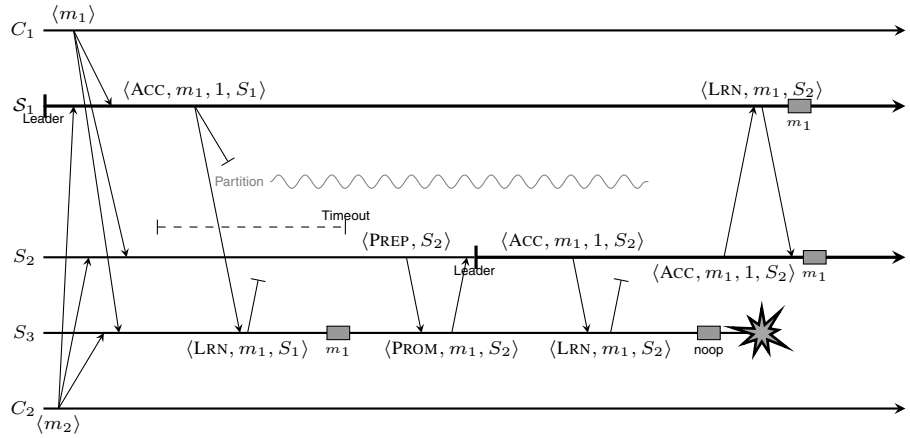


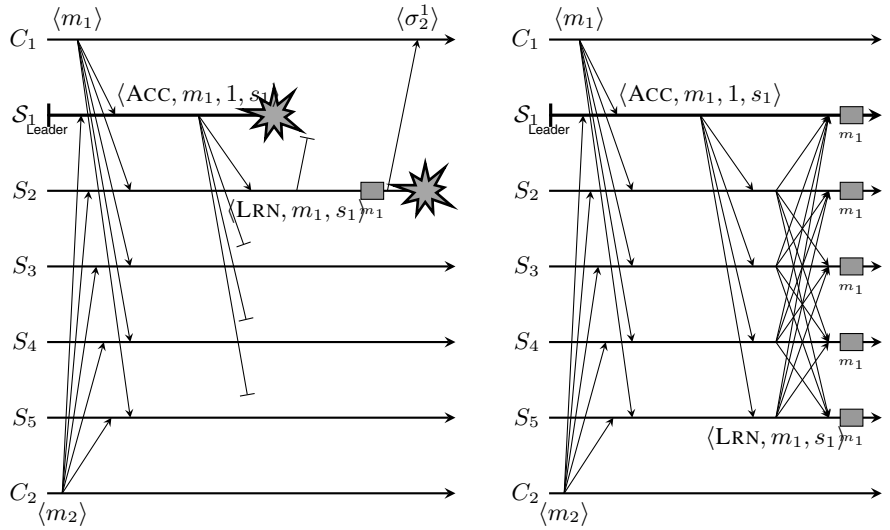
Fig. 11. Solution *LeaderChange*: Previous leader (S_1) sent accept for m_1, but only S_3 executed it. During leader change, S_3 must tell the new leader about this execution in its $\langle \text{PROM}, m_1, S_2 \rangle$ message.

7 Five Servers

Thus far we have explored a protocol that can tolerate a single crash using three servers. To achieve a higher degree of fault tolerance, we can clearly add more servers. However, to ensure that only a majority partition makes progress, as explained in Sec. 5, we can only tolerate that fewer than half of the servers fail. Thus, to tolerate f crashes, we need at least $2f + 1$ servers.

In a scenario with five servers, we can no longer execute a request after receiving the accept. Fig. 12(a) shows that otherwise all servers that knows about this request can fail. We therefore adjust our protocol to send learns to all servers, as depicted in Fig. 12(b), and only execute after receiving three learns for one message. Note that the accept is an implicit learn from the leader, and every server can also send a learn to itself. Therefore a follower can, in practice, execute after receiving one accept and one learn, while the leader can execute after receiving two learns.

Similarly, the new leader needs to collect two promises from the other servers to begin its leader role. Also here the new leader makes an implicit promise to itself as the third one. After multiple, successive leader changes it is possible to receive promises



(a) With five servers we can no longer execute immediately after receiving an accept message.

(b) Execute after receiving one accept and one learn, or two learn messages.

Fig. 12. Paxos with five servers requires additional messages

including different values, sent by different leaders. E.g. a leader receiving promises $\langle \text{PROM}, m_1, S_3 \rangle$ and $\langle \text{PROM}, m_2, S_3 \rangle$ has to choose wether to send an accept for m_1 or m_2. We solve this by adding the identity used in the accept to the promise. Our promises now look like $\langle \text{PROM}, (S_1, m_1), S_3 \rangle$ and $\langle \text{PROM}, (S_2, m_2), S_3 \rangle$. The new leader S_3 sends $\langle \text{ACC}, m_2, S_3 \rangle$, since S_2 has a higher rank than S_1. As in Sec. 6, we can also here replace the server identity with round numbers.

8 Summary

We have presented Paxos, aiming to understand the fundamental mechanisms Our presentation differs significantly from previous attempts to explain Paxos, and in this section we explain how it relates to the presentation in *Paxos made Simple* (PMS) [2].

The first distinction is that PMS introduces separate agent roles: proposers, acceptors, and learners. These roles are at the heart of Paxos' flexibility, and allows one to structure a Paxos system in different ways. While this is very useful for formal reasoning over a wide variety of structures, it can be challenging to comprehend at first. Our servers each combine these three roles. Another difference is that PMS presents the protocol for agreeing on a single client request, among several requests seen by the servers. Thus, one instance of Paxos is used to agree on the next request to be executed. PMS then explains how multiple Paxos instances can be combined to build a Paxos-based RSM. These instances are numbered sequentially, and corresponds to our sequence numbers. In PMS, Lamport also explains that Paxos instances can be optimized to run with only the accept and learn messages, when the leader is stable. Instead we

delay this step, introducing the prepare and promise messages only to solve the leader take over problem. We use the same message naming as in PMS for ease of recognition, but we only gradually augment the content of each message as it is demanded by the different mechanisms that we introduce. In particular, we deferred the introduction of round numbers in messages, which is used in PMS to identify the leader, until the end of Sec. 6. The purpose of the round numbers in PMS is a common source of confusion for many students.

We have focused on scenarios illustrating the need for and function of each individual mechanism in Paxos, sometimes omitting a complete and precise algorithmic description. PMS gives a short and precise description. For readers interested in a blueprint for implementing Paxos, we recommend [13].

Acknowledgements. We would like to thank Maarten van Steen for asking all the right questions that lead us down this path. Also thanks to Keith Marzullo and Alessandro Mei for untangling some early confusions about Paxos back in 2010/11. This work is partially funded by the Tidal News project under grant no. 201406 from the Research Council of Norway.

References

1. Lamport, L.: The part-time parliment. ACM Trans. on Comp. Syst. 16(2), 133–169 (1998)
2. Lamport, L.: Paxos made simple. ACM SIGACT News 32(4), 18–25 (2001)
3. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proc. OSDI, pp. 335–350 (2006)
4. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proc. USENIX ATC (2010)
5. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
6. Martin, J.P., Alvisi, L.: Fast byzantine consensus. IEEE Trans. Dependable Secur. Comput. 3(3), 202–215 (2006)
7. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machines for wans. In: Proc. OSDI, pp. 369–384 (2008)
8. Meling, H., Marzullo, K., Mei, A.: When you don't trust clients: Byzantine proposer fast paxos. In: Proc. ICDCS, pp. 193–202 (2012)
9. Lamport, L.: Fast paxos. Distributed Computing 19(2), 79–103 (2006)
10. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. Technical report, Stanford University (2013)
11. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proc. PODC, pp. 398–407 (2007)
12. De Prisco, R., Lampson, B., Lynch, N.: Revisiting the paxos algorithm. Theor. Comput. Sci. 243(1-2), 35–91 (2000)
13. van Renesse, R.: Paxos made moderately complex. Technical report, Cornell University (2011)