One Acceptor is Enough
EPFL Technical Report LPD-REPORT-2010-001

Maysam Yabandeh, Leandro Franco, and Rachid Guerraoui
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

The celebrated Paxos protocol implements a reliable service as a state machine replicated over several machines. Replication provides reliability against permanent failures and availability against temporary crash failures. Whereas, in theory, Paxos can tolerate any number f of crash failures using $2f + 1$ replicas, in practice, it is often appealing to use it in order to tolerate a single failure using three replicas for the probability of two simultaneous failures is sometimes considered low enough. We show in this paper how, in this particular case of three replicas, we can increase the throughput of Multi-Paxos, which is an efficient variation of Paxos used in deployed systems [4, 2], by a factor of 2 by using a new protocol we call OneAcceptor. In short, OneAcceptor changes Multi-Paxos to use only a single acceptor role and to switch it with another acceptor in the case of failure. The number of exchanged messages between replicas reduces considerably, without trading, however, neither the consistency nor the availability of Multi-Paxos.

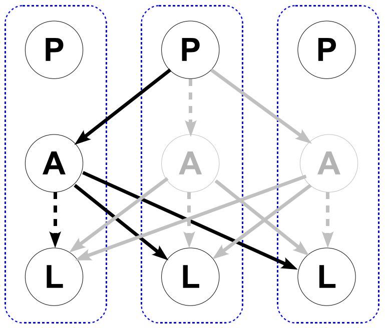

Figure 1: The reduced number of messages in OneAcceptor compared to collapsed Multi-Paxos deployed on three servers. The dotted box represents the node boundary. The dashed messages, which do not cross the node boundary, do not consume the node bandwidth. P, A, and L represent the proposer, acceptor, and learner roles, respectively. The grayed acceptors and consequently the communications to/from them are eliminated in OneAcceptor.

1 Introduction

Recently, the increased demand for data replication over multiple, possibly geographically distributed, data centers, has raised interests to design, deploy, and experiment relevant protocols for maintaining consistency among replicas [4, 16]. Multi-Paxos [15] is one of the most important such protocols. Even if it has been considered in a wide variety of settings [4, 16, 2, 7], it reportedly suffers from scalability issues [2]. In short, Multi-Paxos relies, at any point in time, on one of the replicas to act as a leader. The client commands must be first received by that leader and the leader's network bandwidth limits the number of client messages that can be received per unit of time.

In this paper, we propose a more efficient version of Multi-Paxos, OneAcceptor, designed for the deployment scenario in which the data is replicated over three servers. Indeed, Multi-Paxos can tolerate any number f of crash failures using $2f+1$ replicas. In practice, however, using more number of replicas reportedly increases the average commit latency [7] and decreases the system throughput [2]. It is appealing, therefore, to consider only three replicas which tolerate only a single failure (at a time). The probability of two replicas failing at the same time is sometimes considered low enough for the system designer to accept to wait for at least one of them to recover in that rare failure scenario. OneAcceptor is very similar to Multi-Paxos, with a small difference: OneAcceptor uses only one acceptor. (In Multi-Paxos, the role of the acceptor is to resolve conflicts between multiple proposals.) Figure 1 illustrates the difference between OneAcceptor and Multi-Paxos by highlighting the changes in the communication between nodes. The grayed acceptors are eliminated in OneAcceptor, and the communications to/from them are also eliminated accordingly.

Reducing the number of acceptors to one has a major impact on improving the system performance, yet without jeopardizing neither the consistency nor the availability of the system. In OneAcceptor, the acceptor's availability is provided by *backup acceptors*; the leader switches its failed acceptor with a fresh backup acceptor. Using three servers implies that the replicated data is reliable even against two permanent failures and the system can progress even with one failed server, just like in Multi-Paxos. In the failure scenarios, however, the commit latency increases slightly by OneAcceptor, due to longer recovery time.

To progress even with more number of crash failures, f, more replicas should be used, i.e., $2f + 1$. Inspired from previous work for switching safely between different implementations of BFT protocols [6], we explain how to safely switch from three replicas of OneAcceptor to more replicas of Multi-Paxos. Therefore, the system can still progress even with higher number of failures.

OneAcceptor targets reducing the Paxos-related traffic on the leader and uses the freed bandwidth to service more client requests. Hence it can be more beneficial in the systems that the Paxos-related traffic is the major load on the leader. If the leader server is used to also service other type of requests, to make the best use of OneAcceptor, the non-Paxos traffic should be serviced via other servers perhaps by using proxies, as it is suggested by Chubby [2]. We will discuss the workload issue in detail in Section 5.

Also, the persistent storage of data has been one of the main challenges in efficiently implementing Multi-Paxos [4]; the acceptors has to store their data persistently before responding to any request. After failure, the failed acceptor is not usable until it recovers all the important data from the persistent storage. In our approach, the data in the acceptor is no longer required to be stored persistently; in the case of reboot and losing the data in the main memory, the crashed acceptor is replaced with another backup acceptor. This feature becomes possible in OneAcceptor because it can survive loss of acceptor data.

A system like Chubby [2] could greatly benefit from OneAcceptor if deployed on three replicas. In Chubby [2], the client command messages are to acquire the lock on a particular file and are, hence, small. On the other hand, the bandwidth of the leader is partly used to receive client commands and partly to communicate with the other replicas. By decreasing the number of exchanged messages between the leader and the other replicas, the leader can use the freed bandwidth to service more client commands.

The rest of the paper is organized as follows: Section 2 explains the design of Paxos and dissects the role of each Paxos participant. In Section 3, by explaining the key insight of OneAcceptor, we illustrate the differences between OneAcceptor and Multi-Paxos. The detailed design of OneAcceptor is presented in Section 4. We present our experimental results in Section 5. Section 7 concludes the paper with some final remarks. Appendix A presents the pseudo code of OneAcceptor which is followed by the correctness proofs of OneAcceptor in Appendix B.

2 Background

In this section, we explain Paxos as well as Multi-Paxos, an optimization of Paxos for practical applications [4]. A single central server that services the commands received from a set of clients is clearly not fault-tolerant: any failure in the central server could take the whole system down. One way to address this problem is to use multiple servers and replicate the system state on them. In this way, if one of them fails, the others will continue serving the clients. There are certain challenges in preserving the consistency of the replicas, especially without making strong assumptions about the synchrony of the network. For instance, two issued commands by the clients could reach two servers in the inverse order and that would cause inconsistency between the system state in those two servers.

Paxos is an algorithm proposed by Lamport [15] to address such challenges. Paxos assumes the service to be implemented as a state machine, replicated on multiple servers. It gives an order to the issued commands by the clients and guarantees that all servers execute the commands in the same order. Note that the agreed order is not necessarily according to the time the commands have been issued by the clients. In other words, if a client issues a command C_1 before another client a command C_2, the algorithm guarantees that on all servers they will be applied in the same order, either as C_1-C_2 or as C_2-C_1.

We now give a brief description of the original Paxos algorithm [15] which is called Basic-Paxos hereafter in this paper.

2.1 Basic-Paxos

The original version of Paxos was first presented in [15] and was further explained in [11]. The participant servers in Basic-Paxos implement three different roles: proposer, acceptor, and learner. The proposers advocate the client commands, the acceptors resolve the contention between multiple proposers, and the learners learn the chosen values.

The ultimate goal of Basic-Paxos is to assign orders to client commands. The order of a client command, which

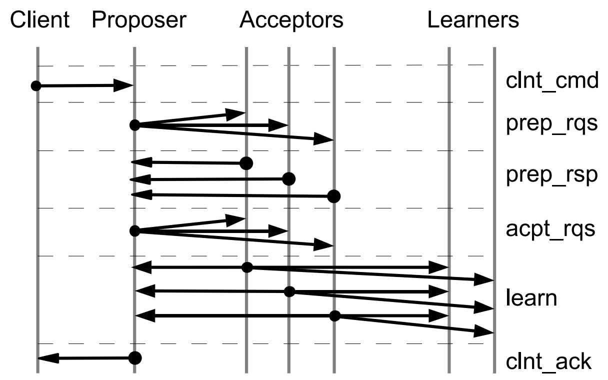

Figure 2: The interaction between nodes in Basic-Paxos. This example consists of one proposer, three acceptors, and two learners. In Multi-Paxos, the leader skips the first phase, i.e., prepare_request and prepare_response.

is called a value in Paxos terminology, is specified by an instance number. To assign values to instance numbers, Basic-Paxos requires two phases. In the first phase, a proposer attempts to become leader for a particular instance number. In the second phase, the leader proposes a value to the acceptors and, this value is learned by the learners. For each instance number, the proposers try to settle on a value. All the message transmissions related to a particular order constitute a separate *instance* of Basic-Paxos. The interaction between nodes is depicted in Figure 2.

We now explain one instance of Basic-Paxos, step by step:

1. client_command: The client_command message, which comes from a client to proposers, contains a command from the client. The proposer then advocates the client command.

2. prepare_request: The proposer first picks an order for the client command, which is called Paxos instance number. Then, it tries to take the leadership position and asks the acceptors to recognize it as such by broadcasting a prepare_request message, which contains a proposal number pn. The proposal number distinguishes different attempts of the proposer for the same instance number.

3. prepare_response: Upon receipt of a prepare_request message, each acceptor checks the proposal number. If the proposal number, pn, is greater than the proposal number of the previous accepted proposals, the acceptor sends a prepare_response message back to the proposer. By that it promises not to accept any proposal number smaller than pn. The highest proposal number must be stored in a persistent storage. If the acceptor has already accepted a value, the value will be included in the prepare_response message.

4. accept_request: After receiving the prepare_response messages from a majority of the acceptors, the proposer assumes itself as the leader. It first decides on a value; one selected from values received from a majority of the acceptors if they have already accepted a value, or any value otherwise. It then sends to all the acceptors an accept_request message with the proposal number pn and the proposed value.

5. learn: When an acceptor receives the accept_request message corresponding to the promise it has made, it accepts the proposal and broadcasts a learn message to all the learners as well as the proposer. The accepted value must be stored in a persistent storage.

6. client_ack: When a learner receives the learn message from a majority of the acceptors, it recognizes the proposed value as chosen and can inform the clients with a client_ack message. Alternatively, this can be done by the proposer that advocates the client command.

Basic-Paxos guarantees the following two safety properties [15]: i) non-triviality: only the proposed values can be learned; and ii) consistency: two different learners cannot learn two different values.

Each role can be implemented by a separate server. But usually a single machine implements all the three roles, which is then called Collapsed Paxos. The advantage is that the transferred messages between two roles that are located on the same server do not cross the node boundary and thus less bandwidth will be consumed. According to the liveness property of Basic-Paxos[13] a value will be eventually chosen, given that enough servers are running. For example, in the mentioned deployment setup, the liveness property holds as long as two of the three servers are running.

2.1.1 The Roles in Paxos

In this section, we take a closer look at the different roles in Paxos. This is essential to understand the rationale behind the applied changes into Multi-Paxos by the proposed protocol, OneAcceptor. As mentioned before, there are three major roles in Paxos: i) proposer, ii) acceptor, and iii) learner.

The proposer role is to advocate the client command. This is essential for scalability of the system. If the clients have to be involved in the consensus process (for example by advocating their own request), then the system cannot scale with the number of clients. By relinquishing this task to the proposers, the consensus is required among only a few servers and thus it is more scalable with the number of clients.

The learner is the actual long-term memory of the system. When a Paxos instance is finished successfully and its value is learnt, this value is kept in the multiple available learners. The clients then can read this value from each of the learners.

The acceptor is the main role in Paxos that makes the consensus achievable. If multiple proposers want to propose values for the same Paxos instance, the acceptor is the key role to resolve the contention between the competing proposers. Suppose some acceptors accept value v_0 from Proposer P_0, and for some reasons the Paxos instance does not complete successfully. Now, to finish the instance, Proposer P_1 must first read the *accepted value* by the acceptors (i.e. v_0) and propose the *same value*. It implies that the acceptors play the role of the short-term memory for the system; they must remember a few values during the short period of one Paxos instance.

2.1.2 Replication in Paxos

In general, we have two types of replication: i) replication of service and ii) replication of data. Replication of service increases the availability of the system. In other words, when a client requests for the service, we want to make sure that there is at least one responding server, ready to receive the client commands. The replication of data, however, is for increasing the reliability of the system. In other words, it decreases the chance of data loss by missing some servers (after permanent failures).

The Paxos roles are replicated, but each one for a different purpose. The replication of the proposers is to increase availability, as the proposers provide a service for the clients, i.e., advocating their request. In contrary, the learners store the data of the system, and the purpose of their replication is to enhance reliability.

The acceptor replication is partly for service availability and partly for data reliability. The proposers start the consensus process by contacting the acceptors. Thus, they require the provided service by the acceptor role to be available. In addition, as mentioned before, there are a few data kept by the acceptors such as the accepted value and the promised proposal number, which should be kept during the Paxos instance. However this data is required only for the active Paxos instance, and in the case of failure, we can think of some workaround solutions.

The main insight of this paper, which will be explained later in Section 3, comes from the following observation: the replication of the acceptor role is mainly for availability, and if its availability is provided via other mechanisms, then the replication of acceptor is no longer necessary.

2.1.3 Persistent Storage

If an acceptor node crashes, Paxos still can progress as long as a majority of the acceptors are running. The crashed acceptor can get back to the game as soon as it recovers the stored data from the persistent storage, such as the highest proposal number and the accepted proposals. Persistent storing of data is one of the main challenges in efficient implementation of Multi-Paxos [4]. The memory buffer of the file must be flushed immediately after each write; otherwise the data in the buffer would be lost by a sudden crash.

2.2 Multi-Paxos

After a proposer takes the leadership position for one instance in, it could be more efficient if it assumes the leadership position for the next Paxos instances in' ($in' > in$) as well. The other proposers can still try to become leader when they suspect that the last leader is failed. Multi-Paxos [11] is the version of Paxos which implements the mentioned optimization. The algorithm is schematically explained in Figure 2.

The first round is similar to Basic-Paxos. When a Proposer P becomes leader, it uses the same proposal number pn for the next Paxos instances. Hence it can skip the first phase of Basic-Paxos, i.e. prepare_request, and start directly with the accept_request message. If in the meanwhile, another Proposer P' tries to become the leader with a higher proposal number pn', then the proposal number of P will not be the maximum proposal number any longer, and its accept_request messages will be rejected. Proposer P can then either relinquishes the leadership position to Proposer P' or try to become the leader again by sending a prepare_request message with a new proposal number.

3 Main Insight of OneAcceptor

As we explained in Section 2, the availability of the acceptor role can be provided in different ways. One approach, which is taken by Multi-Paxos, is the replication of the acceptor. A side-effect of this approach is the increase in the number of exchanged messages between acceptors and other roles. An alternative approach is to have some *backup acceptors* ready to use, and replace the failed (or suspected to be failed) acceptor with a new fresh one from them. Taking this approach is the main insight underlying OneAcceptor; which reduces the number of exchanged messages between servers by a factor of two.

Figure 1 depicts message transmission in a collapsed Multi-Paxos setup that consists of three nodes. The messages that cross the node boundary must be included in the total number of messages. Therefore, we have the following equation for Msg$_{multi-paxos}$, the total number of exchanged messages between servers in a normal Multi-Paxos instance:

$$\text{Msg}_{multi-paxos} = (A-1).(A+1) \qquad (1)$$

, where A is the number of acceptors. Then, for the usual setup of three nodes, this value would be equal to 8 in Multi-Paxos as opposed to 4 in OneAcceptor.

The total number of messages affects the overall consumed bandwidth between servers. Beside that, one interesting parameter is the number of sent/received messages by the leader node, Msg$^{leader}_{multi-paxos}$. The leader exchanges more messages compared to the other nodes and hence when it gets saturated, the system cannot service more client commands. This is reportedly a problem for scalability of Multi-Paxos [2]. In the common setups, each server plays all the Multi-Paxos roles and hence the leader node is also a learner as well as an acceptor. Thus, the total number of messages exchanged between the leader node and the other nodes is:

$$\text{Msg}^{leader}_{multi-paxos} = 3.(A-1) \qquad (2)$$

Again, for the usual setup that includes three servers, this number is equal to 6. OneAcceptor reduces this number to 3 by using only one acceptor. Consequently, we expect at least a factor of two increase in the system throughput by switching from Multi-Paxos to OneAcceptor.

One interesting variation of collapsed Multi-Paxos that we also considered uses fewer acceptors. In such a case, fewer messages would be exchanged since some acceptors are not active. For example, in the common setup with three nodes, if Multi-Paxos uses only two of them as acceptor, then the number of exchanged messages by the leader would be 4 per command, as opposed to 6. Thus, the throughput would be improved by a factor of 1.5. This is less than the improvement by our protocol which is a factor of 2. Using fewer proposers and learners will reduce the availability and reliability of the system, respectively. Therefore, we do not compare OneAcceptor with such variations.

It is worth noting that the above argument is based on the assumption that the nodes communicate through unicast messages. In the particular case that the Multi-Paxos nodes are deployed on the same LAN, the messages could be transferred via broadcast. However, we do not confine OneAcceptor by making any assumption about the deployment environment. Chubby [2] uses Multi-Paxos over wide-area networks where the broadcast is not available. Even inside a data server, the Multi-Paxos nodes are likely to be placed into separate racks, [1] which are connected by a network switch. Broadcasting through switches is not free as it is inside a LAN. Therefore, assuming the availability of broadcast messages is far from realistic scenarios.

So far we have shown that instead of replicating the acceptor role, we can keep the other acceptors as backup, ready for use but not involved in the message passing process of Multi-Paxos. Although, using backup acceptors addresses the problem of availability and yet provides better performance, we still need to find a solution for the reliability of the acceptor data.

Recall that the acceptors also keep a few data, which are necessary during short-term period of a single Paxos instance to address the possible contention between multiple proposers. Missing this data by switching from the active acceptor to a fresh backup acceptor in the middle of a Paxos instance can violate the reliability of the system. For instance, if the active acceptor promises not to take any proposal number less than N, then a fresh new acceptor would not be aware of this promise and might accept proposal numbers less than N. Nevertheless, if the proposers get properly notified of this data loss, they can safely restart the Paxos instance without risking the protocol integrity. For example, upon receipt of the failure notification of the active acceptor, the proposers know that the promised sequence number by the previous acceptor is no longer held.

We will explain in Section 4 that if we assume that the leader and the active acceptor nodes do not fail at the same time, then there exist a process in which the leader can safely notify the other proposers of the active acceptor switch. This assumption is valid in the common setup that consists of three physical servers implementing three proposer, three learner, and one acceptor roles.

By carefully placing the proposer and acceptor roles among the physical servers, in a way that the leader and the active acceptor are placed in two separate physical servers, we can make the assumption that the leader and the active acceptor do not fail at the same time. The violation of this assumption cannot occur unless two of the three physical machines crash. In this case, we would be left with one machine which is less than the minimum required machines for Multi-Paxos to progress ($min > total/2$).

Because OneAcceptor discards a failed acceptor, in contrast to Multi-Paxos, the acceptor data is no longer required to be stored persistently. We need only to identify the silently rebooted acceptors. The leader can detect the reboot by either using TCP as the transport protocol

[1]Google reports that due to cost issues, they replace the whole rack of servers in the case of a failure. Therefore, if all of the Multi-Paxos nodes are placed in the same rack, they all will be unavailable after a single failure.

or initializing the peer after the first contact. A simple variable in the acceptor, such as $IamFresh$, which is initialized by the first contact from the leader, can help the other proposers to detect the reboot of the acceptor.

4 Design

In this section, we explain the design of OneAcceptor in detail. As mentioned in Section 3, the idea is to use only one active acceptor and provide the availability via some backup acceptors. The small changes applied to the protocol also take care of the reliability when the active acceptor is replaced. This idea can be applied to any of the numerous available versions of Paxos. For the sake of simplicity in presentation, in this paper we focus only on Multi-Paxos, which is the stable version used in practical deployments [4, 2].

OneAcceptor uses all the Multi-Paxos messages in a similar format. We first start this section by describing the process in the error-free scenario. Then, we explain the changes we need to make in the Multi-Paxos algorithm to handle each of the failure scenarios.

4.1 Error-free scenario

The steps in the error-free scenario are similar to Multi-Paxos, except that the messages are sent to only one acceptor, i.e. the active acceptor. The roles in OneAcceptor and the interaction between them is depicted in Figure 1.

1. Proposer P decides to take the position of the leader. It first obtains the Id of the active acceptor, A (we will explain the process of obtaining this Id in the next subsection), and sends a prepare_request message including a proposal number, pn, to Acceptor A. By that, the proposer asks the acceptor to recognize it as the leader.

2. If the proposal number, pn, is greater than all the previous proposal numbers received by the acceptor, it sends a prepare_response message back to Proposer P. By that, the acceptor promises not to accept any proposal number smaller than pn.

   Notice that, similar to Multi-Paxos, these two steps are necessary only for the first time a proposer contacts the acceptor. After that, the proposer becomes leader and skips these two steps.

3. Proposer P then sends an accept_request message including the proposal number pn as well as a proposed value, to Acceptor A.

4. When Acceptor A receives the accept_request message corresponding to the proposal number, to which it has given its promise, it accepts the proposal and broadcasts a learn message to all the learners.

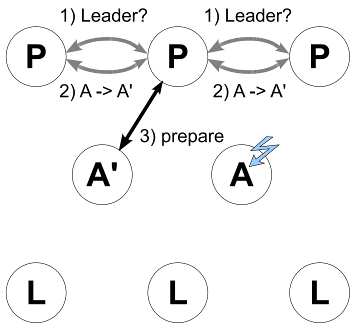

Figure 3: The interaction between nodes in OneAcceptor to switch failed Acceptor A with another backup Acceptor A'. In step 1, the leader makes sure that it is still known as the leader by a majority of the nodes. Then in step 2, it announces the change of the active acceptor. Finally in step 3, it sends a prepare_request message to the new active Acceptor A'.

4.2 Acceptor Failure

Here, we consider the scenario in which active Acceptor A fails and the leader switches it with another backup Acceptor A'. It is worth noting that we do not assume a perfect failure detection. Hence Acceptor A might be still running and just mistakenly be suspected for failure. Thus, the failure recovery scenario must take that into consideration.

When the active acceptor fails, the leader is the only node that is allowed to switch it with another backup acceptor. This change, however, must be confirmed by a majority of the nodes. This is necessary to avoid having multiple instances of active acceptors running in the system. The scenario is illustrated in Figure 3.

Obtaining the confirmation of a majority of the proposers is a separate consensus problem which can be solved by any Paxos-like algorithm. Although, it is possible to merge this consensus into the main operation of the algorithm, for the sake of simplicity, we assume that the consensus over the new active acceptor is achieved by a separate basic implementation of Paxos, which hereafter is called PaxosUtility. Notice that PaxosUtility instance which handles consensus over the new active acceptor is totally separate and independent from OneAcceptor algorithm that we are explaining here.

Beside the Id of Acceptor A', the leader also includes the uncommitted proposed values into the message sent to the PaxosUtility. This is to cover the cases where Acceptor A has received an accept_request message with

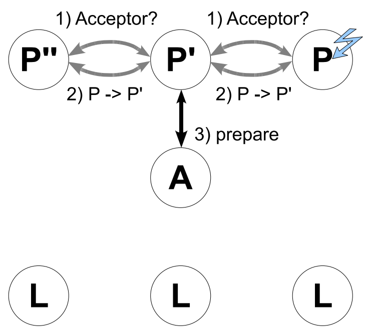

Figure 4: The interaction between nodes in OneAcceptor when Proposer P' takes the leadership position from Leader P. In step 1, Proposer P' inquires for the active acceptor Id. It then announces itself as leader in step 2. Finally in step 3, it sends a prepare_request message to the active acceptor.

value v_{in} for instance number in, but the corresponding issued learn message is not received by the other nodes yet. In this way, it guarantees that the next leader will try to propose the same value as v_{in} for instance number in.

After the leader finishes the consensus over the active acceptor, the leader switches from Acceptor A to Acceptor A', i.e., the new active acceptor. Because the acceptor node has changed, the leader must start over with a prepare_request message to take the leadership of the new acceptor.

4.3 Leader Failure

In Multi-Paxos, every proposer can spontaneously try to take the leadership position by sending a prepare_request message to the acceptors. In practice, this usually happens when a proposer suspects that the current leader is failed. In OneAcceptor also, when the leader fails, any proposer can try to take its position by sending a prepare_request message to the active acceptor.

Assume that Proposer P' suspects failure of Leader P and decides to become the leader. The active acceptor Id, A, can be obtained by inquiring a majority of the nodes. This is because of the fact that the last leader always makes a consensus over changing the active acceptor. The sequence of messages is demonstrated in Figure 4.

Care must be taken to ensure that in the meanwhile the active Acceptor A is not switched by the last leader. Otherwise, we end up with two leaders which are using two different active acceptors. To this aim, Proposer P' uses PaxosUtility to start a consensus instance in which Proposer P' announces that it is going to take the leadership position by assuming A as the active acceptor. Accordingly, every leader must always check for this announcement before switching the active acceptor. If the leader observes this announcement, it must consider its position as relinquished. This step is marked as step 1 in Figure 3.

4.4 Leader and Acceptor Failures

If the active acceptor fails, the leader is in charge of switching it with a fresh backup acceptor. On the other hand, if the leader fails, then any proposer can safely take its position, given that the active acceptor is still running. The only remaining case to handle is when both the leader and the active acceptor fail together.

As mentioned in Section 3, to handle this failure scenario we carefully assign the Multi-Paxos roles to the physical nodes in a way that the leader and the active acceptor are located in two separate physical nodes. Assume that we have N machines available and each machine implements all the roles; the proposer, the acceptor, and the learner. In OneAcceptor that there is only one active acceptor, we have the option to pick the machine that will also play the active acceptor role. This deployment is demonstrated in Figure 1.

The idea is to assign the active acceptor and the leader roles to two separate physical nodes. In this way, the failure of the leader and the active acceptor cannot occur together, unless two of N physical nodes fail at the same time. In the usual setup of Multi-Paxos which consists of three physical nodes, this failure scenario implies that two of the three physical nodes are failed. On the other hand, Multi-Paxos cannot progress with just one running server out of three. Consequently, we can assume that if the failures of the leader and the active acceptor occur at the same time, there is only one machine left. In this situation, neither Multi-Paxos nor OneAcceptor can progress.

It is worth noting that the failure of the leader and the active acceptor at the same time does not jeopardize the consistency of the system. It only prevents the system from progressing, which is the same way that Multi-Paxos would react to this failure scenario.

The detailed pseudo code of OneAcceptor is presented in Appendix A. Furthermore, Appendix B provides the correctness proofs of OneAcceptor.

4.5 More than One Failure

Using only three replicas, no consensus system including OneAcceptor can progress if more than one replica fail; both Multi-Paxos and OneAcceptor can resume though after the failed replicas recover from failure. A recent proposed framework [6] suggests methods to safely switch from one byzantine tolerant consensus protocol (BFT) to another. Inspired by that, we can switch from

OneAcceptor to other consensus protocol implementations such as Multi-Paxos [2] when more than one node are expected to fail in near future. For example, the first failure can issue a warning triggering the switch to Multi-Paxos. Note that by switching to Multi-Paxos, we will lose the increased throughput offered by OneAcceptor till we switch back to it. Therefore if recovery is quick, continuing with OneAcceptor offers better performance.

The switching in crash-only failures is much simpler than BFT since we do not need to consider malicious nodes. To safely switch from OneAcceptor to Multi-Paxos, it is required to ensure two properties: i) OneAcceptor must abort, and ii) Multi-Paxos must be initialized with the pending proposals registered in PaxosUtility.

After getting to the aborted state, nodes must not issue proposals nor process messages of the OneAcceptor protocol. Not all the nodes might respond to an abort request. However, to stop issuing learn messages, it is enough to abort either the leader or the active acceptor. Care must be taken because the aborted node can be suspected to be failed and get replaced with another node. For example, even if the active acceptor aborts, the ignorant leader can switch the active acceptor and the protocol continues working with the new active acceptor.

To make OneAcceptor safely abort, we make use of PaxosUtility, similar to the solution for switching the active acceptor and leader. All nodes abort after observing an abort entry in PaxosUtility. Before making any change into the working set, the nodes have to check the entries in PaxosUtility, and hence they will be surely notified of the abort. To ensure that the current working set is also aware of the abort, either the leader or the active acceptor must be the one that initiates the abort. The node that initiates the abort, also include the pending proposals into the abort message, i.e., the proposed proposals in the leader and the accepted proposals in the acceptor. The Multi-Paxos nodes must be initialized with the pending proposals in the abort entry of PaxosUtility.

The detailed steps of switching the protocol from OneAcceptor are as following:

1. The node that initiates abort, N_a, must be either the leader or the active acceptor. This can be indicated by checking the PaxosUtility entries.

2. Node N_a goes to the aborted state.

3. Node N_a tries inserting an abort entry into PaxosUtility. If Node N_a is a proposer (acceptor), it adds the proposed proposals (accepted proposals) to the abort entry.

4. Other nodes accept abort entry in PaxosUtility, only if no entry in the meanwhile is inserted into PaxosUtility.

5. Every node that is notified of the abort entry aborts.

6. Each node of Multi-Paxos starts working after initialization by the proposals in the abort entry of PaxosUtility.

[2]The Multi-Paxos nodes can be located on separate servers or possibly share some servers with OneAcceptor.

4.6 Avoid Persistent Storage

The important data of the active acceptor is kept partly by the leader and partly by the PaxosUtility. Thus, upon the active acceptor failure the leader can safely discard it. Consequently, there is no need for a persistent storage in the acceptor. However, cares must be taken not to use a rebooted acceptor as its data is lost after reset. The proposers need to distinguish between a rebooted acceptor and a fresh one. To this aim, we use an $IamFresh$ variable in the acceptor which is initially true. Upon adopting a leader, the active acceptor sets this variable to false. The only time that the leader expects the acceptor to be fresh is immediately after switching to it. For the other cases, the proposer can pass a $YouMustBeFresh$ parameter, which is assigned to false, in each communication with the acceptors. If this parameter is not set while the local $IamFresh$ variable is set, the acceptor detects that it was silently rebooted. The implementation details can be found in Appendix A.

4.7 Performance Discussion

On common error-free scenarios, OneAcceptor can perform better because it involves only one acceptor instead of three. Hence the number of sent messages in the protocol reduces, which leads to less consumed bandwidth. Specifically, less transmission to/from the leader enables the leader to service higher number of client commands per unit of time. Under high jitter in network delays, it could also decrease the response time, as the leader and the learners would need to wait for the less number of acceptors to respond; waiting for less number of messages implies being less sensitive to the variance of message delays.

On failure scenarios, however, OneAcceptor requires to take a few more steps compared to Multi-Paxos to recover from a failure, i.e., steps of PaxosUtility instance. Nevertheless, the frequency of failures is usually low enough that we can ignore the performance drawback of the few additional steps in failure scenarios. Considering the Amdahl's law, the overall performance is still significantly higher. Section 5 covers the experimental results for both error-free and failure scenarios.

4.8 Complexity Discussion

As you can see in Figure 11, the handlers implementing the acceptor role are simpler compared to Multi-Paxos. This is because there is only one acceptor in OneAcceptor and thus the complexity due to dealing with quorum of nodes is eliminated. The handlers of the proposer role, however, implement more logic for safe recovery from failures. Overall, our implementation of OneAcceptor service in Mace [9] framework is 460 LoC as opposed to 539 LoC for implementation of Multi-Paxos service. Note that the utility functionalities such as membership management are implemented in separate services and hence are not included in the reported LoC.

5 Evaluation

In this section, we report on the evaluation of OneAcceptor and Multi-Paxos. The main advantage of OneAcceptor is the reduced traffic between servers, which becomes the dominant factor when the servers are geographically distributed. This is the case for many practical deployments of Paxos such as the global cell replication in Chubby [2]. Therefore, in this section we focus on experimental setups which emulate wide-area networks.

We basically explore the following:

1. The OneAcceptor side-effect on the latency of committing the proposals
2. The overhead of OneAcceptor in failure scenarios
3. The increase of the system throughput using OneAcceptor?

5.1 Experimental Setup

Our experiments make use of three machines with 2.83 GHz Xeon X3360s and 8 GB of RAM. These machines run GNU/Linux 2.6.26-1. All machines are interconnected by a full-rate 1-Gbps Ethernet switch. We run the distributed system on top of the ModelNet [18] network emulator. The one-way emulated delay of each link is 50 ms. Both Multi-Paxos and OneAcceptor are implemented in Mace [9] framework. We use TCP as the transport protocol. OneAcceptor also needs a PaxosUtility which can be any implementation of Paxos-like systems. For the sake of simplicity, we have implemented Basic-Paxos to be used as PaxosUtility.

Each machine implements all three roles of Multi-Paxos: proposer, acceptor, and learner. An application on top of the consensus service issues application calls for initialization and proposing values. We have implemented a simple failure detector which is triggered by receiving the TCP RST signal from the crashed node. The workload is discussed in Section 5.2. The commit latency of OneAcceptor is verified in Section 5.3. The overhead of OneAcceptor in failure scenarios is illustrated in Section 5.4. Section 5.5 presents experimental results of comparison between the throughput of OneAcceptor and Multi-Paxos. The throughput under failure scenarios is covered in Section 5.6.

5.2 Workload

The request payloads must be carried to the learners by all fast-path protocol messages (i.e., accept_request and learn_request). Thus, the consumed bandwidth is still proportional to the number of protocol messages. We are not aware of any application that does not require the received payload by the leader to be transferred to the learners. Even so, a set of proxies that are located in the same data center as the leader can receive the large client requests and instead issue some small requests to the leader, referring to the client requests. Inspired from traffic size in Chubby [2], we do not consider large request sizes.

The Paxos protocol messages are issued as a result of each client command, which is the type of traffic targeted by OneAcceptor. The leader might also be involved in other kind of traffics. For example, Chubby [2] reports using some KeepAlive traffic related to the used lease mechanism. As proposed by Chubby [2], these kind of traffics can be proxied to reduce their load on the leader.

In general, the read requests do also cause issuing Paxos protocol messages. This is because the read requests often require the last updated data, which is not necessarily updated in every learner, including the leader node. Thus, the read traffic can be treated as normal client command traffic. There are particular usecases that can change the load on servers. For example, if the read requests do not necessarily ask for the last updated data, then they can be handled directly by each learner. Hence, the read traffic will be balanced on all nodes. In this case, if the proportion of the read traffics is much more than client command traffic, then OneAcceptorś impact on reducing the overall load will be less profound. The other particular case is that the leader has the privilege to directly respond the client read traffic without going through the Paxos steps. For example, in Chubby [2] thanks to the employed additional lease mechanism beside Paxos, the leader directly respond the read traffic as long as the lease is not expired.

Not to lose the generality, we do not assume the above particular cases for handling the read traffic in the experimental results. Nevertheless, as suggested by Chubby [2], the read traffic will be proxied in such cases and will not consume tangible part of node's bandwidth.

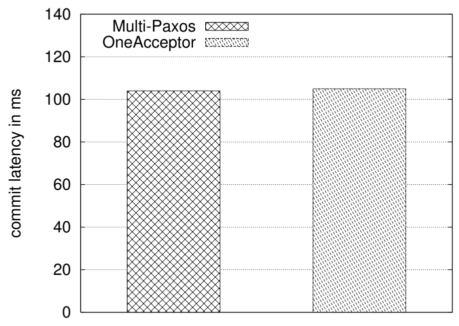

Figure 5: Comparison of commit latency observed in Multi-Paxos and OneAcceptor.

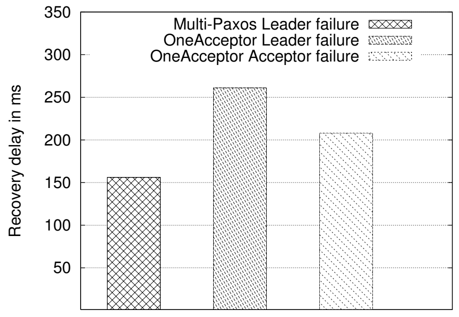

Figure 6: Recovery delay from different failure scenarios in Multi-Paxos and OneAcceptor.

5.3 Commit Latency

To measure the commit latency, the application layer generates 100 proposal requests at rate of 0.5 requests per second. We chose this low request rate to make sure that the measured latency is affected only by the message delays and not the competition between different requests and queuing delays. After committing the last proposal, we measure the average latency between the time a proposal is made and the time it is learnt. Figure 5 depicts the average latency experienced by each client request. The average latency is almost the same in both Multi-Paxos and OneAcceptor; 105 ms in Multi-Paxos and 104 ms in OneAcceptor. This shows that in error-free scenarios, OneAcceptor does not impose any overhead in terms of commit latency.

5.4 Recovery from Failure Scenarios

According to the OneAcceptor design, we expect more overhead in failure scenarios. Nevertheless, because the failure happens very rarely, a negligible amount of imposed overhead would be acceptable. In this section, we measure the imposed overhead in different failure scenarios. The results are depicted in Figure 6. OneAcceptor requires further calls to PaxosUtility service, when either the active acceptor or the leader fails. The first scenario that we consider is the failure of the active acceptor in OneAcceptor. We kill the process of active acceptor and measure the delay between the time that the active acceptor crash is detected by the failure detector and the time that the leader receives prepare_response message from the new active acceptor. This time is equal to 208 ms.

In the second failure scenario, we kill the leader process in OneAcceptor and measure the delay between the time that the leader crash is detected by the failure detector and the time that another proposer stands in the leader position upon receipt of the prepare_response message from the active acceptor. The delay is equal to 261 ms. We repeat the same failure scenario for Multi-Paxos as well. The difference is that in Multi-Paxos a proposer becomes the leader after receiving the prepare_response message from a majority of acceptors, i.e., two. The recovery delay in Multi-Paxos is equal to 156 ms.

As the results show, the recovery delay from the leader failure in OneAcceptor is more than the delay in Multi-Paxos. However, considering the fact that the failure happens very rarely, the small overhead is acceptable. OneAcceptor also takes some time to recover from failure of acceptor. This delay does not exist in Multi-Paxos since Multi-Paxos uses replicated acceptors, and hence does not need to recover from failure of one of them. Using the same argument as used for the leader failure, the recovery delay is acceptable when the failure of servers happens rarely. Furthermore, this delay is much less than the failure detection delay in practical deployments of Multi-Paxos. For example, Chubby [2] reports 4 to 6 seconds (and sometimes up to 30 s) delay for election of a new leader. Therefore, some extra milliseconds does not affect the performance of practical systems such as Chubby [2].

For the sake of simplicity, we used Basic-Paxos to serve OneAcceptor as PaxosUtility. Each commit in Basic-Paxos requires 4 message transfers between servers which partly contribute to the recovery overhead of OneAcceptor. For production quality implementation of OneAcceptor we can remedy the overhead by using other more efficient versions of Paxos.

5.5 Throughput

To measure throughput, the application layer generates 200,000 proposal requests at full rate. The buffer size for outstanding proposals is selected proportional to the throughput which is 1000 and 500 requests in OneAcceptor and Multi-Paxos, respectively. Using larger buffers for each of the protocols resulted in high fluctuation in throughput. After committing the last proposal, we measure the rate at which the client requests were ser-

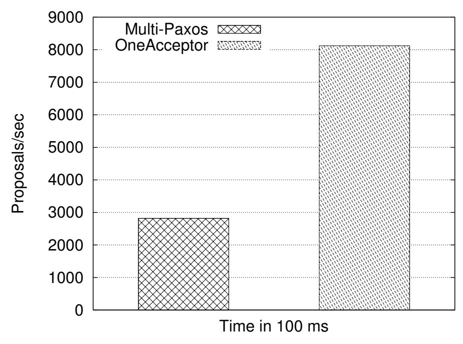

Figure 7: Comparison of throughput achieved by Multi-Paxos and OneAcceptor.

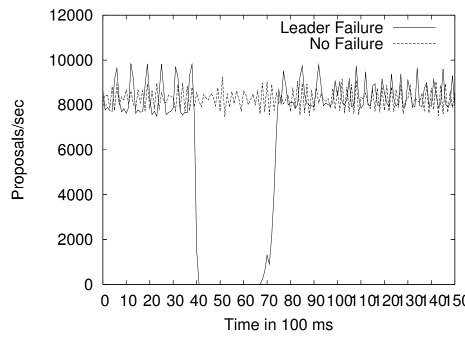

Figure 8: The changes in throughput achieved by OneAcceptor when the leader fails.

viced. Figure 7 presents the achieved throughput by each approach. Multi-Paxos managed to service 2816.9 requests per second. This number increases up to 8120 in OneAcceptor by a factor of 2.88. This increase was expected because of the reduction in the inter-server traffic by OneAcceptor. The leader in OneAcceptor issues accept_request commands to only one acceptor and receives replies from only one acceptor as well. The reduced outgoing and incoming traffic, allows the leader to use the freed bandwidth to service more client requests.

5.6 Throughput in Failure Scenarios

In this section, we measure the changes in system throughput in different failure scenarios. The setup is the same as in Section 5.5 for measuring throughput. Figure 8 plots the throughput of OneAcceptor when the leader fails. There is a gap between the times that the last leader fails and the new leader is elected. In this period, the throughput drops to 0. The Multi-Paxos algorithm behaves the same when the leader fails. As it is illustrated in Figure 9, the low-throughput period is less in Multi-Paxos. This is because OneAcceptor needs some extra steps to safely recover from the leader failure, and

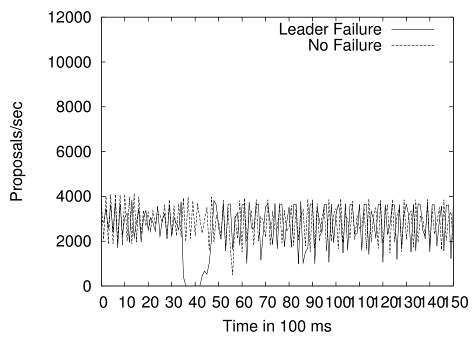

Figure 9: The changes in throughput achieved by Multi-Paxos when the leader fails.

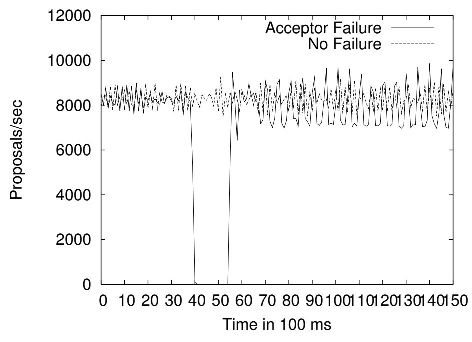

Figure 10: The changes in throughput achieved by OneAcceptor when the active acceptor fails.

in this period it cannot issue new proposals.

Figure 10 plots the throughput of OneAcceptor when the active acceptor fails. Similar to leader failure, OneAcceptor requires some extra steps to switch the active acceptor, during which the throughput drops to 0.

Overall, Multi-Paxos performs better during failure scenarios. Nevertheless, given that the failure happens rarely, the significant increased throughput by OneAcceptor justifies its longer recovery time in the case of failures. In some practical implementations, the application which uses Paxos is deliberately designed to be less sensitive to unavailability. For example, Chubby [2] uses coarse-grained locks as opposed to fine-grained locks to be less sensitive to system availability. Therefore the small unavailability in the failure scenarios is well justified in such a systems.

6 Related Works

Multi-Paxos is the optimized version of Paxos used in practical deployed systems [4, 2]. Paxos provides both reliability and availability of acceptors via replication. In OneAcceptor, we propose to use a single acceptor in the common deployment setup that involves three servers.

We provide the availability of the acceptor via some backup acceptors. To guarantee the reliability of the data kept by the active acceptor, OneAcceptor runs a separate consensus algorithm to achieve consensus over the replaced acceptor. The main insight underlying OneAcceptor is not weaved to any particular version of Paxos and can be applied to all of them to increase the system throughput. In this paper, we applied OneAcceptor to Multi-Paxos.

Mencius[17] is derived from Paxos to distribute the load of client commands among multiple leaders [14]. Assuming a balanced load of client commands received by the leaders, it splits the space of Paxos instance numbers among the leaders and each leader proposes the received client commands only for its range of instance numbers. By doing so, the leaders can in total service more aggregate commands from clients. Note that each leader still has to communicate with all the acceptors to make a proposal. However, OneAcceptor is targeting the load on each leader individually. It is not limited by assuming a balanced load on leaders. By reducing the number of messages exchanged between servers, each leader in OneAcceptor can service more client commands. The main insight of OneAcceptor can be applied to any protocol in the Paxos family. Mencius could also benefit from the main insight of OneAcceptor and increase the system throughput even more. By that, each leader in Mencius would be assigned to a single separate acceptor, and the overall throughput would increase even more.

A subset of protocols in Paxos family target the commit latency of client commands [13, 5, 12]. In Basic-Paxos, each client command takes four message delays between the servers. Multi-Paxos is similar to Basic Paxos for the first command but for the next commands it requires only two message delays between servers. This does not include the RTT delay between the client and the leader. Fast Paxos [13] using more replicas, $3f + 1$, saves the delay between the leader and the acceptors by allowing the client to optimistically send the accept_request messages directly to the acceptors. If the commands from different clients collide, it will be resolved by spending some more steps. The average latency can be lower if the rate of collisions is low, as it is stated by Lamport: ”If collisions are too frequent, then classic Paxos might be better than Fast Paxos.”

In the scenarios that the throughput of the system is a bottleneck, the number of client commands is very high, and consequently the probability of collisions will increase accordingly. OneAcceptor is designed for high-throughput systems and reducing the commit latency of the client commands is not targeted by the algorithm. Fast Paxos cannot outperform the throughput of Multi-Paxos, as the number of sent/received messages to/from each acceptor does not change; although the leader-to-acceptor messages of Multi-Paxos are eliminated in Fast Paxos, the messages must be sent to more acceptors, $3f + 1$. For $f = 1$, the message/node is equal to 6 per command, which is the same number as Multi-Paxos. Moreover, recent studies have questioned the actual latency of Fast Paxos under realistic deployment scenarios [8].

The proposed BFT protocols [10, 3] target safety assuming not only crash-only but also byzantine faults. BFT protocols are more expensive than Paxos family of protocols as they provide stronger guarantees. Multi-Paxos has been successfully integrated into a number of practical deployed [4, 16, 2] systems. OneAcceptor also focuses on Paxos family of protocols.

Zookeeper [1] is a centralized service for maintaining configuration information and providing distributed synchronization. It uses replication for scalability and reliability which means it prefers applications that are heavily read-based. Taking advantage of the assumptions regarding existence of some capabilities such as atomic file create and ephemeral files, Zookeeper's design is much simpler than Paxos. In contrast, OneAcceptor is designed for general usage of Paxos and it is not confined with any assumption about the availability of such capabilities. For the similar reason, the workloads used in Zookeeper are not good benchmarks for evaluation of OneAcceptor and other Paxos-based systems.

7 Conclusions

In this paper, we proposed OneAcceptor which changes Multi-Paxos to use only a single acceptor and to switch it with another backup acceptor in the case of failure. For the deployment setup which uses three servers, the leader can safely handle the replacement of the acceptor. We have shown how to switch to more number of nodes, when more than one failure is envisaged in near future. Furthermore, in OneAcceptor the acceptor's data does not require to be stored persistently, which has been a major challenge in efficient implementation of Multi-Paxos. OneAcceptor addresses the problem of saturated leaders which practical systems such as Chubby [2] have been dealing with. The experimental results show a factor of 2 increase in the system throughput by using OneAcceptor.

References

[1] Zookeeper project. http://hadoop.apache.org/zookeeper, 2008.

[2] M Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI*, volume 11, 2006.

[3] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[4] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007.

[5] D. Dobre, M. Majuntke, and N. Suri. CoReFP: Contention-Resistant Fast Paxos for WANs. Technical report, Technical report, TU Darmstadt, Germany, 2006.

[6] R. Guerraoui, V. Quema, and M. Vukolic. The next 700 bft protocols. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, page 1. Springer-Verlag, 2008.

[7] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, San Francisco, April 2008.

[8] F. Junqueira, Y. Mao, and K. Marzullo. Classic paxos vs. fast paxos: caveat emptor. In *Proceedings of the 3rd USENIX/IEEE/IFIP Workshop on Hot Topics in System Dependability (HotDep.07)*. Citeseer, 2007.

[9] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[11] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[12] L. Lamport. Generalized consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[13] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[14] L. Lamport, A. Hydrie, and D. Achlioptas. Multi-leader distributed system, November 21 2002. US Patent App. 10/302,572.

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

[16] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[17] Yanhua Mao, Flavio Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *OSDI*, 2008.

[18] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*, December 2002.

A APPENDIX: OneAcceptor

The pseudo code for our OneAcceptor algorithm, explained in Section 4, is presented in Figure 11. If the pro-

```
Upon AcceptorFailure
  if (!$IamLeader$) return;
  ($P_i$,$instance$) = PaxosUtility.lastLeader();
  if ($P_i \neq me$) //somebody thought I am dead
    $A_a$ = $null$; $IamLeader$ = false;
    return;
  $A'_a$ = selectAcceptor();
  $proposals$ = uncommitedProposals();
  $success$ = PaxosUtility.propose($instance$,
                AcceptorChange($A'_a$,$proposals$));
  if (!$success$) return;
  $A_a$ = $A'_a$;
  $IamLeader$ = false;

Upon LeaderFailure
  propose();

proc propose()
  if ($IamLeader$)
    $in$ = next_uncommited_instance_number();
    $v$ = getAny($in$);
    sendto $A_a$ accept_request($in$, $pn$, $v$);
  else
    $YouMustBeFresh$ = true;
    $pn$ = new_pn();
    if ($A_a$ == $null$)
      ($A_a$,$instance$,$proposals$) =
        PaxosUtility.lastActiveAcceptor();
      $success$ = PaxosUtility.propose($instance$,
                    LeaderChange($me$, $A_a$));
      if (!$success$)
        $A_a$ = $null$; return;
      registerProposals($proposals$);
      $YouMustBeFresh$ = true;
    sendto $A_a$ prepare_request($in$, $pn$, $YouMustBeFresh$);

Upon Receive prepare_response($A_i$, $pn$, $ap$)
  if ($IamLeader$ || $A_i \neq A_a$) return;
  $IamLeader$ = true;
  registerProposals($ap$);
  $in$ = next_uncommited_instance_number();
  $v$ = getAny($in$);
  sendto $A_a$ accept_request($pn$, $v$);

Upon Receive prepare_request($P_i$, $pn$, $YouMustBeFresh$)
  if ($pn > hpn$)
    if ($IamFresh$ != $YouMustBeFresh$)
      return;
    $IamFresh$ = false;
    $hpn$ = $pn$;
    sendto $P_i$ prepare_response($pn$, $ap$);
  else sendto $P_i$ abandon($hpn$);

Upon Receive accept_request($P_i$, $in$, $pn$, $v$)
  if ($pn \neq hpn$)
    sendto $P_i$ abandon();
  else if ($ap[in] \neq null$)
    multicast $L$ learn($in$, $ap[in]$);
  else
    $ap[in]$ = ($pn$, $v$);
    multicast $L$ Learn($pn$, accepted);
```

Figure 11: OneAcceptor Algorithm

poser recognizes itself as the leader of the active acceptor, Variable $IamLeader$ is set. As in Multi-Paxos, the leader does not need to send a prepare_request message to the acceptor and starts directly with the accept_request message using the last promised proposal number. Variable A_a refers to the active acceptor Id; ap is the map structure keeping the accepted proposals; $IamFresh$ indicates that the acceptor has adopted no leader yet. The highest proposal number is stored in Variable hpn. Initially the highest proposal number is equal to -∞. Procedure *init*, presented in Figure 12, initializes the mentioned variables. As for the rest of the variables, pn is the proposal number, in is the Paxos instance number, v is the value (proposed or accepted), P is the list of the proposers, L is the list of learners, me is the node Id, and $YouMustBeFresh$ indicates that the proposer expects to be the first proposer that contacts the acceptor.

Upon a failure of the active acceptor, the leader first checks whether the others still believe him as the leader or not. If not, one other proposer has taken its position (probably because of a false leader failure alarm). In this case, it relinquishes the leadership position and return. Otherwise, it calls *selectAcceptor* function to select a new acceptor which is located on a separate machine than the leader node. The leader then announces the change of the active acceptor, *AcceptorChange*, through PaxosUtility. It also attaches the uncommitted proposed values to the *AcceptorChange* entry. The failure of this step indicates that another item is chosen for the current instance of PaxosUtility. In this case, the leader returns from this procedure to try again later. In case of success, however, the leader resets Variable $IamLeader$ because it has to start from the first phase of Paxos with the new active acceptor.

Upon failure of the current leader, a proposer tries to take its position by calling Procedure *propose*. The procedure then obtains the active acceptor Id and sends a prepare_request message to it.

Procedure *propose* proposes a value for the next uncommitted instance number. If the node is already the leader, it directly sends an accept_request message to the active acceptor. Otherwise, it sends a prepare_request message to the active acceptor, in accordance with the first phase of the Paxos algorithm. If the active acceptor Id is unknown to the proposer, it must be obtained via PaxosUtility. The *lastActiveAcceptor* method checks the sequence of committed entries looking for the last *AcceptorChange* entry; this entry contains the active acceptor Id. Next, the proposer adds a *LeaderChange* entry via PaxosUtility. The failure of this step indicates that another item is chosen for the current instance of PaxosUtility. In this case, the procedure resets the value of A_a and returns. We assume that the implementation retries the failed attempt via timers or some other mechanisms.

```
proc init()
  $IamLeader$ = false; $A_a$ = $null$;
  $ap$ = emptyMap(); $hpn$ = $-\infty$
  $IamFresh$ = true;

proc getAny($in$, $ap$)
  $v$ = $proposed[in]$;
  if ($v \neq null$) return $v$;
  $v$ = nextClientRequest();
  $proposed[in]$ = $v$;
  return $v$;

proc registerProposals($proposals$)
  foreach p in $ap$
    $proposed[p.in] = p.v$;
```

Figure 12: The implementation of Procedures init, getAny, and registerProposals, in OneAcceptor Algorithm

In the case of success, before sending the prepare_request message, it first registers the proposed values which have been recorded with the last AcceptorChange entry. If the acceptor is supposed to be a fresh backup acceptor, it also sets Variable $YouMustBeFresh$ which is sent by the message.

Upon receipt of the prepare_request message from Proposer P_i, the acceptor verifies the highest proposal number hpn to be less than the requested proposal number, pn. Otherwise, it sends an abandon message back to Proposer P_i. If Variable $IamFresh$ is set but Variable $YouMustBeFresh$ is not, it indicates that the proposer expected the acceptor to be already adopted by the last leader. However, due to the acceptor reset, the acceptor has lost its data, including hpn and ap. This check avoids the cases where the active acceptor silently reboots before the leader switch. In this case, the last leader should switch the rebooted acceptor.

Upon receipt of the prepare_response message from the active acceptor, the proposer claims the leadership position by setting Variable $IamLeader$. The *getAny* method, presented in Figure 12, picks a value to be accepted for the instance in. The picked value can be any given value, unless there is already a proposed but uncommitted value for the instance in. This case can occur in change of the active acceptor, when some proposed values are not committed yet by the previous active acceptor. If any proposal matches the instance number in, to avoid inconsistency, the proposer picks the same previously proposed value. It then sends an accept_request message to the active acceptor.

Upon receipt of the accept_request message from the leader, the acceptor first checks for the proposal number. Also, it checks that there is no proposal accepted corresponding to the instance number, i.e., $ap[in]$. Otherwise, it broadcasts the learn message of the accepted proposal again to cover the cases that the lost learn message has motivated the proposer to retry. It then stores the proposal in the accepted proposal map, $ap[in]$. Afterwards, the accepted proposal is broadcasted to all the learners accordingly.

B APPENDIX: Proof of Correctness

Here, we prove the correctness of the algorithm presented in Appendix A. We first prove some properties for the entries in PaxosUtility, which we then use to prove that no two different values would be accepted for the same instance number. The proof for the simple case where there is no change in the active acceptor nor the leader node, is trivial and similar to the proofs of Paxos. Here, we focus on the complex cases where the algorithm switches the leader and the active acceptor.

PaxosUtility contains entries for changing the active acceptor, i.e. AcceptorChange, and entries for changing the leader, i.e. LeaderChange. We define the *Global leader* and *Global acceptor* as follows:

definition: In the sequence of PaxosUtility entries, the node which has inserted the last LeaderChange entry is the *Global leader*. Similarly, the active acceptor announced by the last AcceptorChange message, represents the *Global acceptor*. We use GL_i to represent the ith Global leader and GA_i to represent the ith Global acceptor.

Lemma 1: An AcceptorChange entry is inserted only by the Global leader.

Lemma 1 is guaranteed by lines 3..13 of Figure 11. In Line 4 the leader verifies that it is still the Global leader. It also keeps the index of the last empty instance number, $instance$. Later in Line 10, it proposes the AcceptorChange message for that instance number. The failure of this phase implies that another node has inserted something in the meanwhile. In this case, the handler returns to retry the procedure later from scratch. Therefore, the AcceptorChange message is inserted only by the Global leader.

According to Lemma 1, the Global acceptor represents the active acceptor which the Global leader is working with.

Now we prove by induction that the same value will always be accepted for a particular instance number. The first step is to show that a Global leader does not propose two different values for the same instance number when it switches between the acceptors. Hereafter, we use the pair (v,i) to represent the value v and instance number i of a given accept_request messages.

Lemma 2a: Suppose that GL_l has issued two accept_request messages, (v_a,i_a) and (v_{a+1},i_{a+1}), to two consecutive Global acceptors GA_a and GA_{a+1}, respectively. If $i_a = i_{a+1}$, then $v_a = v_{a+1}$.

Lemma 2a is directly followed by the implementation of the Procedure *getAny* in Figure 12. There, the leader first checks the history of the proposed values. If any value has already been proposed for the requested instance number, then the procedure returns the same value. Hence, as long as the Global leader is not changed, the proposed value for a particular instance number will be always the same.

The next step is to show that an acceptor accepts the same proposals from two consecutive Global leaders.

Lemma 2b: Suppose that the active acceptor GA_a accepts two accept_request messages, (v_l,i_l) and (v_{l+1},i_{l+1}), from two consecutive Global leaders GL_l and GL_{l+1}, respectively. If $i_l = i_{l+1}$, then $v_l = v_{l+1}$.

Node GL_{l+1} becomes the Global leader only after successfully inserting a LeaderChange entry via PaxosUtility. In the algorithm presented in Figure 11, this happens only at Line 30 inside the Procedure *propose*. It also implies that the value of Variable $Iamleader$ is false (Line 25). GL_{l+1} will not start proposing values unless the value of Variable $Iamleader$ changes to true (Line 21). Line 41 is the only location where the value of this variable is changed to true upon receipt of a prepare_response message. It indicates that the active acceptor has received the prepare_request message, approved the proposal number, and responded by the prepare_response message which is also piggybacked by all the previous accepted proposals, ap. The received accepted proposals are registered by the leader (Line 42). The registered values will be later used for all the next proposals in Procedure *getAny*. In other words, the GL_{l+1} will propose the same values which acceptor GA_a has already accepted.

Similar to Basic-Paxos, GA_a will reject all the other potential issued accept_request messages by GL_l after sending the prepare_response message to GL_{l+1}. On the other hand, as we showed above, if GA_a has accepted any value from GL_l for a particular instance number, it will not receive any different value from GL_{l+1} for that sequence number. Consequently, GA_a always accept the same values from two consecutive Global leaders.

Having Lemma 2a and Lemma 2b, now we present the correctness proof of the algorithm.

(*) Suppose that two acceptors GA_a and $GA_{a'}$ accept two accept_request messages, (v_a,i_a) and ($v_{a'}$,$i_{a'}$), received from the Global leaders GL_l and $GL_{l'}$, respectively, where $l' \geq l$ and $a' \geq a$. If $i_a = i_{a'}$, then $v_a = v_{a'}$.

The proof is by induction on the size of sequence of entries in the PaxosUtility utility. Assume that property (*) holds when PaxosUtility has k entries. We prove that it still holds when PaxosUtility has $k + 1$ entries.

Recall that the entries in the PaxosUtility utility are either AcceptorChange or LeaderChange. If the $k+1$th entry is AcceptorChange, based on Lemma 1 it is inserted by the last Global leader. Thus, the GL is the same and the GA changes. This is the case in Lemma 2a for which we proved that no two values will be proposed for the same instance number. If the $k+1$th entry is LeaderChange, we can assume that GA is the same during this change. This is provided by the Lines 29..30 in Figure 11, where the new leader takes the same active acceptor as was taken by the last leader. This case is covered by Lemma 2b for which we proved that no two values will be accepted for the same instance number. Consequently, if we assume that no two values are accepted for the same instance number in the first k entries of PaxosUtility utility, this also holds for the first $k+1$ entries.

Now, to complete the proof, we need to show that the theory holds for $k = 2$. We can make it hold by an initialization process. At the start up, the node with the smallest Id can insert two entries for LeaderChange and AcceptorChange to announce itself as the Global leader and its active acceptor as the Global acceptor. Because, no change in the roles happens in the initial case, neither for the leader nor for the active acceptor, then the theory directly holds for this case.