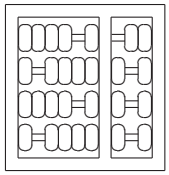

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Seamless Paxos Coordinators

Gustavo Maciel Dias Vieira
Islene Calciolari Garcia *Luiz Eduardo Buzato*

Technical Report - IC-10-13 - Relatório Técnico

April - 2010 - Abril

Seamless Paxos Coordinators

Gustavo M. D. Vieira* Islene C. Garcia Luiz E. Buzato†

Instituto de Computação, Unicamp
Caixa Postal 6176
13083-970 Campinas, São Paulo, Brasil
{gdvieira,islene,buzato}@ic.unicamp.br

Abstract

In Paxos, failures can cause the replacement of the coordinator agent, a key process of this consensus algorithm. The replacement of the coordinator, in its turn, leads to a temporary unavailability of the application implemented atop Paxos. Solutions to the unavailability problem have been sought because of the widely recognized utility of Paxos as a building block of fault-tolerant distributed applications. So far, the problem has been lessened by reducing the coordinator replacement rate through stable leader election. We have observed that the recovery of the newly elected coordinator's state is at the core of the unavailability problem. Thus, in this paper we present a new solution to the problem that allows the state recovery to occur concurrently with new consensus rounds. The complexity and performance of the solution is assessed both theoretically and experimentally. Experimental results show that the unavailability period is removed during the replacement of the coordinator making it seamless to the application.

1 Introduction

Paxos [5] is a consensus algorithm for asynchronous distributed systems; it relies on a key agent, the coordinator, to ensure its safety. The algorithm also guarantees liveness as long as there is one, and only one, coordinator. When Paxos is used to decide multiple instances of consensus, as in the case of the delivery of ordered messages, the requirement of a single coordinator can hinder its progress. Reasons for this are the higher workload processed by the coordinator [2] and the inherent cost of replacing the failed coordinator [3]. The coordinator faces higher CPU and I/O loads than the other Paxos agents because of its main task. It acts as a sequencer and processes all application messages that need to be ordered. It does so by initiating many consensus instances and keeping track of their outcome.

Even when the coordinator can handle the higher workload without compromising the overall performance of the fault-tolerant application, it is still subject to failures that eventually will cause its replacement. Coordinator *replacement* is carried out in two steps: a

*Partially supported by CNPq grant 142638/2005-6.
†Partially supported by CNPq grant 473340/2009-7 and FAPESP grant 2009/06859-8.

new coordinator is elected, and then it is validated [5]. Coordinator *election* is handled by any unreliable leader election mechanism that is equivalent to a Ω failure detector [4]. The unreliability of this mechanism means that it can erroneously change leaders many times, but it will select a single coordinator eventually. Coordinator *validation* requires the new coordinator to have its role ratified by a majority of Paxos agents. To achieve this, the new coordinator has to receive a potentially large prefix of the current state from this majority. Validation ensures that the new coordinator is up to date with the state of all active consensus instances. A newly elected coordinator can resume its activities only after the completion of validation. Thus, the replacement of a coordinator triggers a costly operation that is inevitably going to happen many times in the presence of partial failures and incomplete or inaccurate failure detection. The *temporary unavailability problem* occurs because normal Paxos operation can only be resumed after a successful validation.

The periods of unavailability are a real concern for fault-tolerant systems based on Paxos. Burrows [1] provides a concrete example of the troubles caused by the replacement of Paxos coordinators in a production system. Finding suitable solutions for the temporary unavailability problem is an interesting research challenge with practical implications. The most common way to mitigate the unavailability problem is to devise a mechanism that makes it harder to replace the coordinator. Malkhi et al [7] have proposed an election procedure with built-in leader stability. Using this procedure a coordinator is only replaced if it isn't able to effectively perform its actions. Another approach is to grant an implicit lease to the current coordinator [3]. This ensures it won't be demoted needlessly, but increases the time it takes to detect an actual failure. However, these approaches only tackle the problem of spurious coordinator replacement caused by inaccurate failure detection. They can't really help in the event of a real coordinator failure and the following coordinator replacement.

In this paper we show an alternative approach to solve the temporary unavailability problem that stems from breaking coordinator validation in two concurrent activities: activation and recovery. Coordinator *activation* corresponds to the actual ratification of a coordinator by a majority. We show that it is possible to reduce the information necessary to activate the new coordinator to a single integer. In fact, we show that the coordinator doesn't need to rebuild the complete state of a majority of processes before it can resume its work, it just needs to discover the highest consensus instance that a majority of processes agrees is free to use. This can be done using only a single exchange of fixed size messages, allowing the new coordinator to resume operation in a very short time. Coordinator *recovery* then becomes a secondary task that can take much longer to finish, but that does not block the validation. The result is a much briefer coordinator validation whose time is limited primarily by the activation time. The coordinator's state recovery, the longer step, occurs while the coordinator is already managing new consensus instances.

Experimental results show that our concurrent validation procedure guarantees progress with increased throughput, even in the presence of very costly coordinator replacements caused by process failures. In the presence of these replacements, we have observed uninterrupted operation free from the temporary unavailability problem. Additionally, we have observed that failure detector mistakes caused by high load conditions can also trigger the replacement of a coordinator, even if there are no process failures. Under high load the

proposed coordinator validation does increase the throughput. Seem as a whole, the results show that our coordinator validation mechanism makes coordinator replacement seamless to the application and increases its performance.

The paper is structured as follows. In Section 2 we give an overview of the Paxos algorithm and introduce the basic terminology we will use throughout the paper. In Section 4 we describe our seamless coordinator validation procedure and prove its correctness. In Section 5 we show our experimental results. In Section 6 we discuss the applicability of our results to the Fast Paxos algorithm. Finally, in Section 7 we describe related word and in Section 8 we make some concluding remarks.

2 Paxos

Informally, the *consensus* problem consists in each process of a distributed system proposing an initial value and all processes eventually reaching a unanimous decision on one of the proposed values. The Paxos algorithm is both a solution to the consensus problem and a mechanism for the delivery of totally ordered messages that can be used to support active replication [8]. In this section we give a summarized description of Paxos and make explicit the key role performed by the coordinator. Full descriptions of the algorithm can be found in [5, 6].

Paxos is specified in terms of roles and agents; an agent performs a role. Different implementations of Paxos may choose different mappings between agents and the actual processes that execute them. Agents communicate exclusively via message exchanges. The usual asynchronous crash-recovery computation model is assumed. The roles agents can play are: a *proposer* that can propose values, an *acceptor* that chooses a single value, or a *learner* that learn what value has been chosen. To solve consensus, Paxos agents execute multiple *rounds*, each round has a *coordinator* and is uniquely identified by a positive integer. Proposers send their proposed value to the coordinator that tries to reach consensus on it in a round. The coordinator is responsible for that round, and is able to decide, by applying a local rule, if any other rounds were successful or not. The local rule of the coordinator is based on quorums of acceptors and requires that at least $\lfloor N/2 \rfloor + 1$ acceptors take part in a round, where N is the total number of acceptors in the system [6]. Each round progresses through two phases with two steps each:

- In Phase 1a the coordinator sends a message requesting every acceptor to participate in round r. An acceptor accepts the invitation if it has not already accepted to participate in round $s \geq r$, otherwise it declines the invitation by simply ignoring it.
- In Phase 1b, every acceptor that has accepted the invitation answers to the coordinator with a reply that contains the round number and the value of the last vote it has cast for a proposed value, or *null* if it has never voted.
- In Phase 2a, if the coordinator of round r has received answers from a quorum of acceptors, it analyzes the set of values received and picks a single value v. It then asks the acceptors to cast a vote for v in round r, if v is not *null*, otherwise the coordinator is free to pick any value and picks the value proposed by the proposer.

- In Phase 2b, after receiving a request from the coordinator to cast a vote, acceptors can either cast a vote for v in round r, if they have not voted in any round $s \geq r$, otherwise, they ignore the vote request. Votes are cast by sending them and their respective round identifiers to the learners.
- Finally, a learner learns that a value v has been chosen if, for some round r, it receives Phase 2b messages from a quorum of acceptors announcing that they have all voted for v in round r.

This description of the algorithm considers only a single instance of consensus. However, Paxos also defines a way to deliver a set of totally ordered messages. The order of delivery of these messages is determined by a sequence of positive integers, such as each integer maps to a consensus instance. Each instance i eventually decides a value v and this value is the message (or ordered set of messages) to be delivered as the ith message of the sequence. The value v is offered by the proposers, and they can either select a suitable i from their local view of the instance sequence or ask the coordinator to select i from its view. Each consensus instance is independent from the others and many instances can be in progress at the same time. In fact, for any agent its local view of the set of all instances can be divided in three proper subsets: the decided instances, the undecided instances that were initiated (Phase 1a) and the infinite set of uninitiated instances. Figure 1 shows an example of the status of the consensus instances as seem by an agent. In this example the set of decided instances is $\{0, 1, 3\}$, the set of undecided instances is $\{2, 4, 6\}$ and the set of uninitiated instances is $\mathbb{N} \setminus \{0, 1, 2, 3, 4, 6\}$.

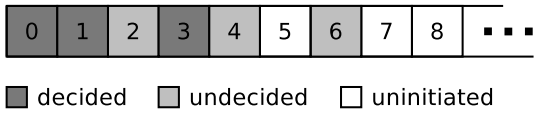

Figure 1: Local View of an Agent

Paxos assumes the crash-recovery failure model, where agents fail by crashing and later recover with access to only the data previously stored in stable memory. To guarantee the correctness of a consensus instance the acceptors must store in stable memory: rnd_a—the last round they have took part (Phase 1a), $vrnd_a$—the last round where they cast a vote and $vval_a$—the vote cast (Phase 2a). The coordinator must store: $crnd_c$—the last round it has started, to ensure it won't start the same round twice [6]. Stable memory requirements for the set of consensus instances in Paxos are the same for a single instance, but multiplied by the number of instances. Thus, each agent must store an array of instances, where for each instance i it records $rnd_a[i]$, $vrnd_a[i]$, $vval_a[i]$ and $crnd_c[i]$. Additionally, the learner agent may store $dval_l[i]$, the value decided in instance i, but this isn't strictly necessary as a new successful round will yield this same value. Usually, all agents are implemented in each process and agents may use the information stored by other agents to implement optimizations. For instance, a coordinator can inform proposers that their selected instance number i is already decided, or similarly, acceptors can inform a coordinator that an instance i it is trying to start is already decided.

In Paxos, any process can act as the coordinator as long as it correctly chooses a value, if any, that is proposed in Phase 2a. There can be only one active coordinator at any given time for the algorithm to ensure progress. If two or more processes start coordinator agents, the algorithm can stall as the multiple coordinators cancel each other rounds with fast increasing round numbers. For this reason, liveness of the algorithm resides on a coordinator selection procedure. This procedure doesn't need to be perfect. Safety is never compromised if zero or more than one coordinator are active at any time. However, the coordinator selection needs to be robust enough to guarantee that only a single coordinator will be active most of the time.

It is clear that the coordinator in Paxos has a very important role, as all successful consensus instances must be started (Phase 1) and lead to completion (Phase 2) by a coordinator. After receiving all Phase 1b messages the coordinator discovers that no value was previously voted for most of the consensus instances. This is expected as only rounds that happen after failed rounds carry a potentially decided value. It is possible to use this observation to reduce the latency to reach consensus through a validation procedure. During validation the coordinator tries to start a new round for all uninitiated consensus instances concurrently. If successful, the coordinator is then able to use this round to continue any instance directly from Phase 2. This way, it is possible to reduce from five to three message delays the time required to reach consensus.

3 Original Coordinator Validation

We now describe in more detail how validation is performed in the original Paxos specification [5]. During validation a coordinator selects a round number r and starts *all* consensus instances at the same time with a single message, as the Phase 1a message carries only the round number. If r is large enough, acceptors will respond to this message with a finite number of Phase 1b messages with the actual votes and an infinite number of Phase 1b messages with no votes. Lamport [5] notes that only the finite set of messages containing an actual vote need to be sent back to the coordinator, framed in a single physical message. No message has to be sent to the coordinator for each of the infinite instances that have had yet no vote. The coordinator processes all Phase 1b messages received and it *assumes* that the infinitely many omitted messages correspond to Phase 1b messages with no vote. All messages received or presumed voteless are processed as usual and a suitable value will be selected to be voted for each instance, or the instance will be marked free (no previous value) and will be used when necessary. This way a coordinator can start the Phase 2 of any free instance as soon as it receives a proposal, and consensus for this instance can be reached in three message delays [6].

This validation procedure requires the coordinator to learn the status of all decided and undecided consensus instances of a quorum of acceptors to determine the exact identities of the infinite uninitiated consensus instances. So, the combined state of a quorum of acceptors represents the state footprint a new coordinator must obtain to be able to start passing new consensus instances. To reduce the footprint of the recovery state, it is possible to determine a point d_c in the instance sequence as seen by the coordinator such as all instances $i \leq d_c$

are in the decided set. The point d_c doesn't necessarily determine all instances in the decided set, but this isn't necessary. The coordinator can then indicate the prefix d_c of the instances it already knows are decided and the acceptors need only send information about larger instances [5]. This combined message is finite, but even with the footprint reduction it can be very large and must be fully recovered so the coordinator can (1) discover all instances this acceptor has voted and (2) use this information to infer the set of instances the acceptor *has not* voted. Moreover, the coordinator must expect complete responses from a quorum of acceptors before it can complete Phase 1 for all instances. While this happens, the coordinator remains blocked and no progress is possible.

4 Seamless Coordinator Validation

Our proposal for a seamless coordinator validation is based on the observation that validation can be broken in two concurrent activities: activation and recovery. Activation is a procedure where the acceptors inform a recovering coordinator of the instances they have not voted, so it can decide values for these instances in only three message delays. Recovery is the task of updating the coordinator view of the consensus instances, learning the outcome of decided instances and initiating rounds for the undecided ones. This split view of the validation procedure is interesting because only activation is required to be finished before a coordinator can resume its activities. Recovery, while necessary, do not pose any restriction on the coordinator operating on the uninitiated consensus instances. This happens because a coordinator doesn't need to immediately start the consensus instances it isn't necessarily free to choose a value for and, as a consequence, it doesn't need to be informed of their status beforehand. To use this fact to create a faster and seamless validation we have to devise an activation procedure that do not require the transfer of the finite, but possibly very large, set of decided and undecided consensus instances. Before we describe how the coordinator does this, it is useful to understand why the coordinator doesn't need to have knowledge of the status of the consensus instances currently in progress to function.

If we look at the sequence of round numbers effectively used in a consensus instance, it is possible to notice that these numbers are distinct and increasing but that they need not to be sequential. In fact, if they are partitioned among the processes in a way that gives each process equal chance of having a larger number, they are never sequential. Thus, a coordinator picks a round number, for instance i, to be any round number larger than $crnd_c[i]$, but not necessarily $crnd_c[i] + 1$. From this simple observation it is easy to see that if the coordinator only records the largest round number initiated for *all* instances it is guaranteed to be able to always choose a larger round number for any individual instance when necessary. In this case, the stable memory footprint of the coordinator is reduced to a single integer $crnd_c$, no matter how many instances of consensus where ever initiated. Clearly, the coordinator still must keep track of the progress of the rounds it initiates, including the round numbers of the rounds in progress, but this information may be stored in volatile memory.

This simple modification makes clear the fact that the coordinator doesn't concern itself with the decided or proposed values of consensus, but only with the proper initiation and

progress of rounds. It still computes the Phase 2a rule, but can do so only in volatile memory. Furthermore, the coordinator can still keep this detached view of the sequence of consensus instances even when it is not the single agent in a process. The coordinator can even peek at the learner stable memory, for example, to avoid starting a round for an already decided instance. In this setup, the coordinator depends on the functionality of other agents to implement optional functionality, but still requires just a single integer in stable storage to guarantee safety.

As the coordinator maintains only a very small state in stable memory, it needs to query the acceptors on all information it needs to complete a consensus round. More importantly, it depends on the information held by the acceptors to execute the activation procedure and be able to start rounds immediately. Then, it is crucial to understand the view of the consensus instances held by each acceptor. To fulfill their role in the algorithm each acceptor keeps in persistent memory the following information for each instance i: $rnd_a[i]$, $vrnd_a[i]$ and $vval_a[i]$. According to the algorithm, i starts in the uninitiated set, with all this information set as *null*. As the algorithm progresses, data is stored in these fields and the instance passes to the active but undecided set and eventually to the decided set. As instances are picked for use in a strict incremental order and, more importantly, there are an infinite number of instances to be used, we can establish a point f_a in the instance sequence as viewed by acceptor a such as every instance i where $i \geq f_a$ had never had a vote cast. It is possible to find other not voted instances smaller than f_a, but we know for sure that all instances larger than or equal to f_a were never voted. For example, in the local state depicted in Figure 1 we have $f_a = 7$.

The seamless coordinator validation is based on an activation procedure that relies on the determination of a point f_Q consistent with the individual points f_a for each acceptor a of a quorum Q. The seamless activation follows these steps:

1. The coordinator sends a Activation Phase 1a message, with round number r starting *all instances*.

2. When an acceptor a receives this message it computes its f_a. If r is larger than the last ballot used in another *activation* or there was no previous activation, then a sends a single Activation Phase 1b message containing its f_a, meaning that it is sending Phase 1b messages for all instances $i \geq f_a$ and *only for these instances*.

3. The coordinator, when it receives Activation Phase 1b messages from a quorum Q of processes, computes f_Q to be the larger f_a, for all acceptors $a \in Q$. It then considers that it has received a Phase 1b message with no votes from all acceptors in the quorum for instances $i \geq f_Q$, and from now on proceeds as traditional Paxos.

Figure 2 shows an example of the activation process for four acceptors a_1, a_2, a_3 and a_4. Assuming all of them are able to take part in the activation, they compute f_a respectively as $f_{a_1} = 4$, $f_{a_2} = 6$, $f_{a_3} = 7$ and $f_{a_4} = 6$. The coordinator computes $f_Q = 7$ and finishes activation.

The seamless coordinator validation presented here requires considerably less information to be propagated from the acceptors to the coordinator. It takes one broadcast from

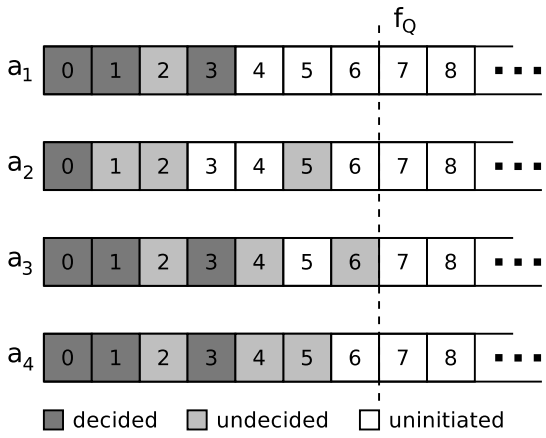

Figure 2: Global View as Observed by a Coordinator

the coordinator containing the round number and Q unicasts from the acceptors to the coordinator containing a single integer f_a. This contrasts with the original coordinator validation [5] in which the coordinator broadcast is answered by Q unicasts containing all previous votes for consensus instances $d_c < i < f_a$, as described in Section 3. Each vote contains, besides the round number, the contents of the actual application message (or ordered set of messages) voted in one specific consensus instance. It is not difficult to see that sending and processing these messages can have a considerable cost for Paxos. Moreover, while these messages are being sent the coordinator can't proceed and Paxos blocks.

4.1 Correctness

The correctness of this seamless coordinator activation is derived from the correctness of individual Paxos consensus instances. Even as the coordinator initiates many of these instances, each of them respects strictly the rules of the Paxos protocol, and the proofs contained in [5] still hold. In this section, we show that the activation procedure we propose does not perform any operation forbidden by the Paxos algorithm.

The first step is straightforward. Any process that considers itself to be the coordinator can start a round in any instance, as long as the rules for selecting a new round number are respected. The sending of an Activation message (page 7, item 1) containing the command to start all rounds, if setup with a suitable round number, doesn't violate any of Paxos invariants.

The correctness of the second step (page 7, item 2) depends on the following facts relative to the behavior of an acceptor: (i) it can always determine the f_a point, (ii) it respects Paxos rules when answering Phase 1a messages, and (iii) it doesn't violate the algorithm liveness. Uniqueness of f_a follows from the observation that we can always find a larger instance than the last instance voted, because voted instances are finite and we have an infinite number of instances. We can consider this point as the frontier to the infinite number of instances $i \geq f_a$ that can be treated as a single instance I, with respect to the

stable memory storage requirements. This is possible because the only way an instance i can deviate from the group is by leaving the uninitiated set, but this leads by construction to $i < f_a$. This means that, as successive coordinator activations lead to evaluations of f_a, initiated instances stop being represented by I and are treated as regular instances. Thus, as we run Paxos for this especial instance I, as part of the activation, we are following Paxos for all instances $i \geq f_a$. Acceptor a is able to decide to answer the Activation Phase 1a message of the coordinator by comparing r with the value of $rnd_a[I]$ and it does so by using a single message, and records the new value of $rnd_a[I]$. Finally, we must ensure that the activation process does not violate liveness by acting only on $i \geq f_a$. By Paxos rules, an acceptor can almost always refrain from answering a message (for example, if it refers to a smaller round), so once a determines f_a it is free to ignore the requests for instances $i < f_a$. Explicitly non answering a message indefinitely could cause a liveness violation, but as f_a is always defined, the Activation Phase 1a message is eventually answered. Also, the instances $i < f_a$ are subsequently treated as normal instances and a sends timely answers for Phase 1a messages not related to activation.

In the third step (page 7, item 3), we must show that the determination of f_Q allows for the correct evaluation of the coordinator's rule. In any Paxos round, the coordinator is only free to set an arbitrary value to a instance if it receives only *null* votes from all acceptors in a quorum. For any given acceptor a, the coordinator considers that it has received a *null* vote for all instances $i \geq f_a$. The coordinator receives answers from a quorum Q and establishes the point f_Q to be the larger f_a, for all acceptors $a \in Q$. It is easy to see that only for instances at least as large as f_Q a full quorum of *null* votes is received. All instances smaller than f_Q will miss at least one vote to complete a quorum. The coordinator then can treat all instances $i \geq f_Q$ as started and free to use. This leaves many instances $i < f_Q$, that are not yet decided, from the acceptors where $f_a < f_Q$. These instances will be treated normally later, as they are not required for the coordinator operation.

5 Experimental Evaluation

The seamless coordinator validation allows activation and recovery to occur concurrently. It is reasonable to suppose that the added concurrency will reduce considerably the time taken to setup a new coordinator, allowing Paxos to fulfill its function as a support for highly-available applications without interruption. We have designed two sets of experiments to assess our hypothesis. One set compares the performance of the two versions of coordination validation, original and seamless, in the presence of induced coordinator and network failures. The other set investigates the performance of the same coordinator validation versions during executions where processes do not fail, but adverse conditions associated with the environment where Paxos executes make the failure detector misbehave triggering coordinator replacements. The results of both sets of experiments show that the seamless coordinator validation guarantees that Paxos does not stop delivering ordered messages during the replacement of a coordinator, providing a significant performance increase for the application. In the next section we provide an outline of the experiments, with an emphasis on the components used and parameters that are shared by both sets of

experiments. The description of experimental conditions and setup that are specific to each experiment set are described in the following two sections.

5.1 Method

Our tests were made using Treplica, a modular total-order broadcast toolkit that implements Paxos and Fast Paxos [9]. Treplica has been designed to be easily instrumented to generate the measurable indicators necessary to assess the performance of Paxos. The toolkit provides a programming interface that allows the construction of applications that adhere to the state machine replication approach.

Our experimental method consists in comparing the relative performance of two coordinator validation procedures. So, to minimize any possible effect of the application execution upon the performance measurements we have devised an application that performs very simple operations: a hash table. The object that is replicated using Treplica is a wrapper around the original Java hash table implementation that turns it into a replicated and persistent object. The workload is exclusively composed of a sequence of hash table put operations, where each operation associates an integer, sequentially incremented, with a random five character string. Read operations were ruled out because they do not represent a significant cost for Paxos. This way, we have a workload that is homogeneous in terms of system resource use and that is always guaranteed to make Treplica the only sub-system of the experiment responsible for the performance variations observed. Treplica is configured to use the local disk of the computing system where replica is executed as its persistent data store, so disk accesses do not trigger any network usage. This way we guarantee that the network is used only to carry the messages exchanged by the replicas as a consequence of Paxos activity.

The experiments were carried out in a cluster with 18 nodes interconnected through the same 1Gbps Ethernet switch. Each node has two Intel Xeon 2.4GHz processors, 1GB of RAM, and a 40GB disk (7200 rpm). System software in each node include Fedora Linux 9 and OpenJDK Java 1.6.0 virtual machine.

5.2 Induced Failures

In the first set of experiments we measure the performance of the application while coordinator failures are induced through the controlled injection of faults. The process that runs the coordinator agent is killed, recovers and is again elected as the coordinator even as it has now fallen behind the remaining correct processes. As a consequence, it has to recover its state by learning about all the consensus instances that occurred while it was down and also has to resume coordination.

Faultload To make possible the occurrence of the failure scenario just described, we have changed the coordinator selection procedure implemented in Treplica to always elect as the coordinator the same process p_f. This is easily accomplished by swapping the original leader election of Treplica by a simple priority-based leader election with the highest priority assigned to p_f. With the new coordinator selection process in place, it is possible to trigger

the occurrence of the scenario through the injection of faults at two different sub-systems: process and network.

Process faultload: Process p_f is crashed and stays down for 30s. Then, it is allowed to recover and is immediately elected coordinator again.

Network faultload: Process p_f network interface is brought down and after 90s it is brought up again. Then, p_f is immediately elected coordinator again.

All faults are injected at the operating system level, using automated scripts that do not require any human intervention during the duration of the experiment. All faults are injected 30s after the end of the ramp-up time, that is, 120s after the beginning of the run.

Workload Server replicas and workload generators share the same hosts, but care has been taken to ensure that the load generation wasn't competing with the application processing and that the specified workload was being generated. The generated load is measured in operations per second (op/s) and is generated at a fixed rate equally divided among all the load generators of hosts that do not fail. This way, the load is unaffected by failures.

For all experiments we run a system with 5 replicas under a continuous load of 1000 op/s for 10 minutes. The first 90 seconds and final 30 seconds are discarded as ramp up and ramp down time, for a total of 8 minutes of effective run time. During the ramp up time caches are being filled and the Java just-in-time compiler is optimizing code. During the ramp down time some operations can be left incomplete as the replicas are brought down. For each faultload (process and network), and for each type of validation procedure (original and seamless) we have performed 10 distinct runs and recorded the average performance in operations per second, continuously throughout the entire execution time.

Results Figures 3 and 4 show the data for the process and network failure scenarios, respectively. For both faultloads we observe the same general behavior. As expected, performance is much affected by the recovery of a replica. Moreover, for the original validation procedure the throughput of the system is effectively zero while in the seamless validation it drops but maintains itself at about 20% of the peak performance. Comparing process and network failures shows little difference. Network failure requires no local recovery from stable storage and preserves local data and code caches, thus it happens faster and have a smaller impact on system throughput.

These results allow us to conclude that the seamless coordinator validation introduced here definitely improves the availability of the application supported by Paxos in the presence of a coordinator failure and recover. It is worth observing that the duration of the recovery (width of the valleys) is practically identical for the original and seamless graphs. This is expected as the recovery time is proportional to the size of the state that has to be recovered. The average throughput of the 10 runs for each experiment are listed in Table 1, with the corresponding coefficients of variation (CV)[1].

[1]The coefficient of variation is the ratio of the standard deviation to the mean.

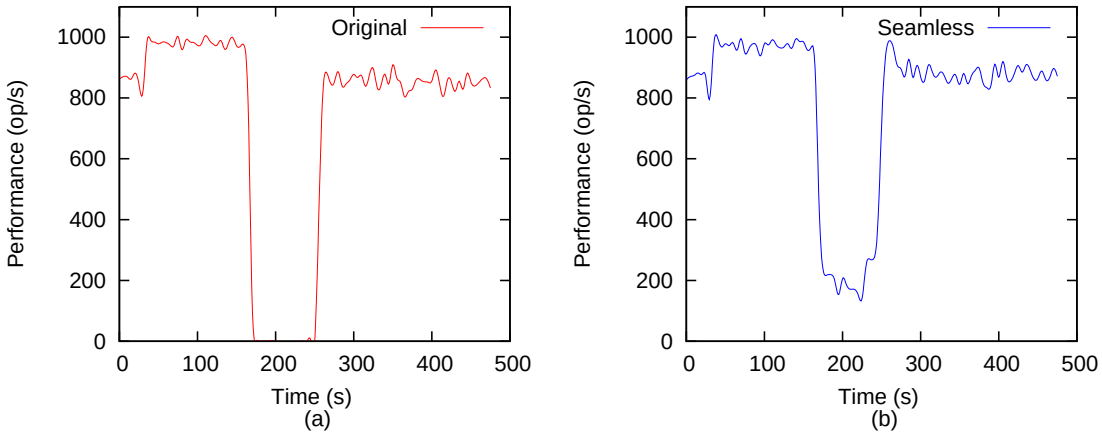

Figure 3: Process Faultload

Scenario	Original (op/s)	CV	Seamless (op/s)	CV
Process faultload	732.66	0.0136	795.44	0.0376
Network faultload	783.28	0.0186	814.15	0.0176

Table 1: Performance with Induced Failures

5.3 Intrinsic Failures

The first set of experiments showed that the seamless coordinator validation is better than the original coordinator validation in the presence of coordinator failures. In this set of experiments we would like to verify whether the seamless validation is worthwhile in relation to the original validation when intrinsically occurring transient failures are the adversary of Paxos. Intrinsic failures that occur at the host environment of the replicated application can misguide the failure detectors of each Paxos replica. For example, a failure detector can report a correct coordinator as having crashed due to a transient communication delay or due to a delayed execution of a thread. Irrespective of the underlying causes, failure detector mistakes trigger coordinator changes. As with the first set of experiments, we could have designed a faultload and a workload that would induce the failure detectors to fail. Instead, we have decided to pursue an indirect but more realistic approach: accelerate the rate of occurrence of intrinsic failures by overloading the application's host environment. The increased number of transient failures should increase the likelihood of failure detector mistakes, this, in their turn, should trigger validations and should allow the measurement of any performance differences between the two versions of Paxos, if they exist.

Workload As for the first set of experiments, the measure used to discriminate between the two versions of coordinator validation is the performance of the replicas. The experiments rely on speedup and scaleup trials to subject the environment to increasingly higher

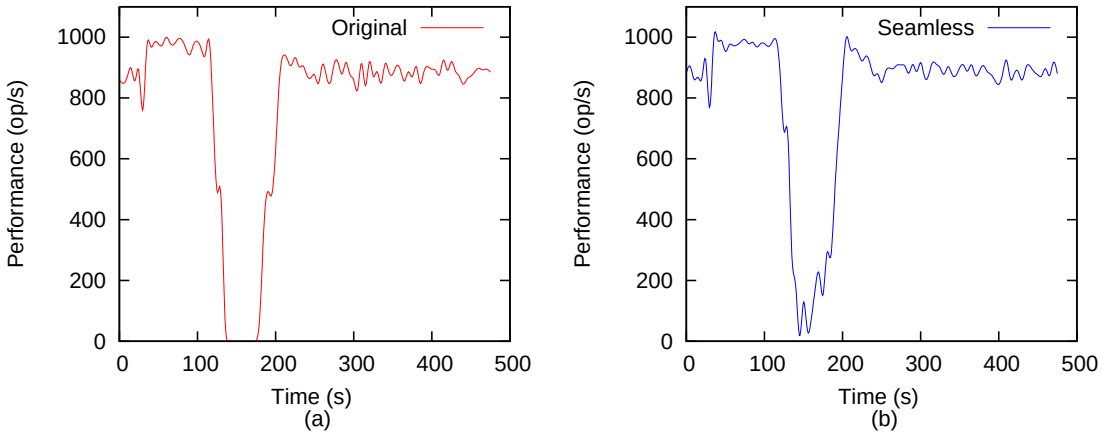

Figure 4: Network Faultload

workloads. The configurations used for the speedup and scaleup are as follows:

Speedup: For a fixed number of 9 replicas, the workload varies from 100 op/s to 3000 op/s in steps of 400 op/s.

Scaleup: For a fixed workload of 3000 op/s, the scale of the system goes from 3 to 15 replicas in steps of 2 replicas.

Both configurations were run for 10 minutes for each data point. For the same reasons stated for the first set of experiments, the first 90 seconds and final 30 seconds are discarded as ramp up and ramp down time, yielding a total of 8 minutes of steady-state runtime. For each data point in each workload (speedup and scaleup) and for each type of coordinator validation (original and seamless), we have measured the performance of the replicas in operations per second as the average of five distinct runs. Care has been taken to verify that none of the processes failed during the runs. Thus, any coordinator replacement will have to result from a failure detector mistake, and measurable differences between runs can be attributed to the relative efficiency of the validation strategies being compared.

Results Figures 5(a) and 5(b) shows the data for the scaleup and speedup experiments, respectively. In the speedup trials, both the seamless and original coordinator validations have statistically identical performance for all but the 3000 op/s workload. This can be explained by the small state a coordinator will have to obtain during recovery. Contrary to the induced failures experiments, the runs of the intrinsic failures experiments are free of process crashes. Thus, it is reasonable to expect a fairly good synchronization to be maintained by the replicas for most of the workloads. By synchronization we mean that all replicas have a very similar view of the decided, undecided and uninitiated consensus instances. Both versions of coordinator validation show similar performance across all of the speedup runs because the main reason behind the validation unavailability problem is

related to the time it takes to recover the state of a replica. As the recovery state is small, the advantage of the seamless validation over the original validation procedure should also be small.

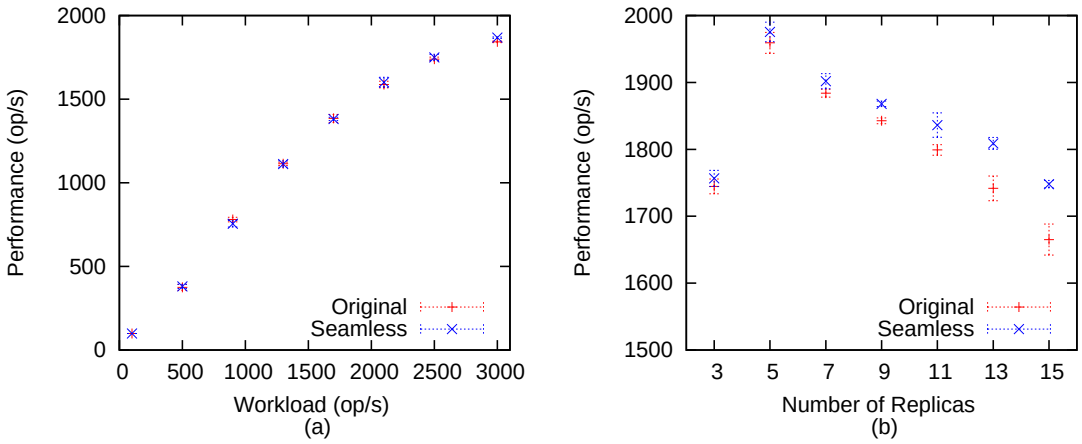

Figure 5: Speedup (9 replicas) and Scaleup (3000 op/s)

In the scaleup trials the seamless coordinator validation has performance similar to the performance of the original coordinator validation up to 7 replicas. At these scales, the same analysis used to explain the speedup results is valid. For scales ranging from 9 to 15 replicas the seamless coordinator validation outperforms the original coordinator validation. Interestingly enough the first scale where this happens is the same as the one observed for the speedup trials (9 replicas, 3000 op/s). The behavior of both versions of coordinator validation can be explained by two factors. The first is that the mistakes induced by the failure detectors are now going to affect increasingly larger subsets of the replicas. So, larger sets of acceptors are going to initiate validations concurrently. In the worst case this process can take more replicas out of their relative synchronization, making their states diverge. The second factor is related to Paxos itself. As the scale increases so does the minimum number of replicas required by Paxos to activate the new coordinator. This means that a larger number of recovery states has to be transferred and processed by the new coordinator, at least from a quorum of acceptors. For example, for 13 replicas the coordinator will have to process recovery states from at least 7 replicas, but it will very likely receive 13 responses. As the scales increases, in the absence of process crashes, the combined effect of the two factors cause the growth of the recovery states. In these scenarios, as expected, the advantage represented by the parallelism introduced by the seamless coordinator validation shows its effectiveness and outpaces the original coordinator validation.

6 Fast Paxos

Fast Paxos is a variant of Paxos that reduces the overall latency of the consensus rounds, measured in communication delays, by allowing the proposers to send proposed values directly to the acceptors. To achieve this, rounds are separated in *fast* rounds and *classic* rounds. Fast and classic rounds have different quorums associated with them, with properties such that the coordinator is still able to detect if a previous round was successful, even if the round was directly conducted by the proposers. Fast Paxos quorums are larger than the ones used by Paxos. For example, a possible configuration has both fast and classic quorums containing $\lfloor 2N/3 \rfloor + 1$ acceptors, from a set of N acceptors [6]. A Fast Paxos round is very similar to a Paxos round, except that Phase 2 is changed to:

- In Phase 2a, if the coordinator of round r has received answers from a quorum of acceptors, it analyzes the set of values received and picks a single value v. It then asks the acceptors to cast a vote for v in round r, if v is not *null*. Otherwise, if r is a fast round the coordinator sends a *any* message to the proposers indicating that any value can be chosen in round r. In this case, the proposers can ask the acceptors directly to cast a vote for a value of their choice in round r.

- In Phase 2b, after receiving a request to cast a vote from the coordinator (if the round is classic) or from one of the proposers (if the round is fast), acceptors can either cast a vote for v in round r, if they have not voted in any round $s \geq r$, otherwise, they ignore the vote request.

Coordinator validation is central to the performance of Fast Paxos. When a new coordinator runs validation, it completes Phase 1 and Phase 2a of the algorithm for all undecided consensus instances. It then sends a collective *any* message authorizing the proposers to initiate any of these instances. Proposer initiated instances can reach consensus in only two communication latencies without the need of further coordinator intervention [6]. Unfortunately, Fast Paxos cannot always be fast. Proposers can propose two different values concurrently, in this case, their proposals may collide. Also, process and communication failures may block a round from succeeding. Different recovery mechanisms can be implemented to deal with collisions and failures, but eventually the coordinator intervention may be necessary to start a new classic round [6].

The way Fast Paxos bypasses the coordinator to reduce communication latency removes coordinator validation from the critical processing path required to decide a consensus round. So, one might assume that the reduced role of the coordinator means that there is no need to optimize the coordinator's operation. However, even in this restricted role, the coordinator still oversees all activity of the algorithm and ensures that instances are decided timely in the presence of collisions or message loss. Moreover, Fast Paxos can only be fast if a suitable coordinator has successfully performed validation, instructing the proposers on how to proceed. Thus, coordinator validation must be quick in the presence of process or network failures, so the system can resume operations in its coordinator-free state.

Seamless coordinator validation is easily adapted to Fast Paxos. During activation the coordinator decides if it will start all unused instances with a classic or fast consensus round.

If it decides for a classic round, the procedure is exactly the same as described in Section 4. If it decides to start fast rounds, it must wait until it receives Activation Phase 1b messages from a fast quorum Q_F of processes. It them computes f_{Q_F} as described in Section 4 and sends a *any* message informing the proposers that instances $i \geq f_{Q_F}$ are prepared and free to use. In essence, the activation step of the seamless coordinator validation doesn't deal with the specific steps required to complete a consensus round. It only divides the consensus instances in sets in a way that it is possible to act on the infinite set of unused instances with a simple message exchange representing Phase 1 of the complete Paxos or Fast Paxos algorithm. This is clearly visible in the simple way the mechanism can be adapted to Fast Paxos.

7 Related Work

The importance of the coordinator replacement procedure was observed by Chandra et al. during the design and operation of the Chubby distributed lock system [3]. The designers of this system decided to make it harder for a replica to loose its coordinator status at the cost of slower detection of process failures. This is justified by the low incidence of observed process failures. In general, coordinator stability is considered the best way to deal with the cost of coordinator replacement. For example, the leader election algorithm proposed by Malkhi et al. captures precisely the network connectivity requirements of a working coordinator while guaranteeing stability during failure-free operation [7]. Ensuring stability makes sure a working coordinator will operate for the larger time possible, distributing in time the cost of replacement. However, even the cleverly designed algorithm of Malkhi et al. can't ensure stability if its weak network connectivity requirements aren't met, even if it only happens for a brief time. In this case, a faster validation procedure is desired.

The fact that Paxos requires a single coordinator is at the root of the unavailability problem. This single process will eventually fail, or be mistakenly taken for failed, requiring a new coordinator to take its place. Another approach was taken by Camargos et al. and consists in not relying in a single one but on a group of coordinators [2]. Their justification is that multiple coordinators make the algorithm more resilient to coordinator failures without requiring the use of Fast Paxos and its larger quorums. The resulting algorithm is considerably complex and increases the number of messages exchanged between the acceptors and the group of coordinators. Our simpler seamless coordinator validation procedure has similar coordinator resilience if we consider the whole set of replicas that can act as a coordinator as a coordinator group where only a master is active at any time and master changes are very cheap.

8 Conclusion

In this paper we have shown a novel way to avoid the temporary unavailability problem caused by Paxos coordinator replacements. Our solution is based on the observation that the validation of a new coordinator is composed of two activities: activation and recovery. We have shown that only the completion of the activation is strictly required before

the coordinator can resume its operation. This fact has led us to a seamless coordinator validation has two important characteristics. First, it allows activation and recovery to be performed concurrently. Second, it reduces the information required to activate the new coordinator to a single integer exchanged between the acceptors.

We have verified experimentally that the seamless coordinator validation avoids the temporary unavailability problem in the presence of process crashes, providing uninterrupted operation for the application built atop Paxos. We have also observed the seamless coordinator validation performs better than the original validation in scenarios where only intrinsic transient failures make the failure detectors trigger validations. This second set of results showed that the seamless coordinator validation is particularly interesting in environments that change their number of replicas dynamically.

Finally, the seamless coordinator validation have other implications for the research on failure detectors for Paxos. Our enhanced validation procedure removes the restriction that the occurrence of validations must be avoided. In this case, instead of using more complex stable leader elections, it is possible to use very simple leader election mechanisms to choose the new coordinator. A fairly imprecise leader election procedure, but one that responds fast to failures or is less costly to implement, can be used without hindering the performance of Paxos. Actually, one can even consider election procedures that decide which process becomes the new leader not only based on the detection of failures but also on other factors, such as the load experienced by the replicas at the moment of the election. In summary, the seamless validation procedure is not only effective, it encourages research on the combined use of failure detectors and load balancers to create more adaptive versions of Paxos.

Acknowledgments Luiz and Gustavo would like to thank Prof. W. Zwaenepoel and Olivier Cramieri. Prof. Zwaenepoel for letting us have full access to the computer cluster at EPFL. Olivier for his always prompt answers to our calls for help whenever a machine or a switch was in need of attention.

References

[1] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[2] L. J. Camargos, R. M. Schmidt, and F. Pedone. Multicoordinated agreement protocols for higher availabilty. In *NCA '08: Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*, pages 76–84, Washington, DC, USA, 2008. IEEE Computer Society.

[3] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

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

[6] L. Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, Oct. 2006.

[7] D. Malkhi, F. Oprea, and L. Zhou. Ω meets Paxos: Leader election and stability without eventual timely links. In *DISC '05: Proceedings of the 19th International Conference on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2005.

[8] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[9] G. M. D. Vieira and L. E. Buzato. Treplica: Ubiquitous replication. In *SBRC '08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*, Rio de Janeiro, Brasil, May 2008.