ABSTRACT
Paxos is an algorithm that provides an elegant and optimal solution to the consensus problem in distributed systems. Despite its conceptual simplicity, industrial strength and high performance implementations of Paxos are very hard – most of the problems stem from state management and garbage-collection. This paper presents a novel variation of the Paxos consensus algorithm that exploits overwrite semantics to eliminate most of the complexities and inefficiencies introduced by state management. This variation is applicable in applications where the current state depends only on the last update as opposed to the entire history, such as group management and distributed key-value stores. The paper presents the State Paxos algorithm and compares its performance with other industrial strength implementations of consensus algorithms.

Categories and Subject Descriptors
Distributed Algorithms [Performance Evaluation]: Middleware

General Terms
Paxos

Keywords
Paxos, DDS, Middleware, Group Management, Zookeeper, Performance Evaluation

1. INTRODUCTION
The use of communication middleware is today a common practice to reduce the complexity involved in the design and implementation of distributed systems. In spite of the advances in performance and functionalities, mainstream platform are still lacking support for high-level coordination primitives, such as leader election, consensus and distributed mutual exclusion.

As a result, distributed systems’ architects are often faced with the challenge of designing distributed coordination into the application layer – often adopting solutions that are either sub-optimal or incorrect. We strongly believe that communication middleware should raise the abstraction level to provide, along with high level communication primitives, high level coordination primitives. As such, we have started to define Levitha [10], an Open Source toolkit that provides a collection of composable coordination abstractions designed on top of Opensplice Mobile [13], a DDS [3, 4] compliant pub/sub middleware developed at PrismTech. Levitha is written entirely in Scala [14] and can be used with any programming language targeting the JVM, such as Java, Scala, and Groovy.

This paper introduces, State Paxos, a novel distributed consensus algorithm that constitutes one of the building block of our distributed coordination library.

1.1 The Consensus Problem
Consensus is a problem that arises in concurrent and distributed systems when different processes (threads) need to agree on a common value. Distributed consensus is extremely powerful and complex. It has been extensively studied in a large number of papers [2, 8], and one of the most relevant contribution has been the design of the Paxos algorithm [6].

Consensus is defined by four properties: termination, validity, integrity and Agreement. Paxos perform consensus in a partially asynchronous system, with non byzantine faults. Paxos requires little assumptions over the set of processes, it does not guarantees the liveness property (termination) during the asynchronous periods, but always ensures safety properties. Assuming that the synchronous time spans are long enough to complete a round, then, in general, all properties are satisfied. Although Paxos has been thoroughly investigated in the academia [7, 9, 11, 15], there are only few publicly available implementations [1], and despite its conceptual simplicity, industrial strength and high performance implementations of Paxos are known to be very hard – most of the problems stem from state management and garbage-collection.

Paxos is a two phase protocol based on three sets of logical entities: proposers, acceptors and learners. The only assumption over these sets is that a quorum of acceptors never crashes. The protocol uses a version number which is a unique and non-decreasing. An acceptor replies to an adopt message, having a version greater than the current,
through an adopted message containing the last accepted
<value,version> couple. Then, it moves to (adopts)
the message version. It reacts to an accept message having
a version greater or equal to the current with an accepted
message, containing the version of the accept message, and
updates (accept) the current value and version to the ones
from the accept message. Accepted messages are broadcasted
towards all learners. An eventual leader is elected among
proposers and will attempt to force learners into taking a
decision. In phase one, it sends an adopt message towards
acceptors. When the proposer receives at least a quorum
of adopted messages, it moves in phase two. It selects the
highest versioned non-null value among the adopted mes-
sages. Otherwise, if messages have all null values, it can
choose an arbitrary value. It then sends an accept message
towards acceptors containing the selected value. When a
learner receives at least a quorum of accepted message, it
will broadcast a decide message with the value contained
in the accepted message. When a learner receives a decide
message, it takes the decision.

The classical version performs only a single decision. How-
ever we can extend the version with a slot number i, which
identifies the ith decided event, as least significant number.
With this simple manipulation we can serialize a stream of
events.

Performances can be improved through the Multi-Paxos vari-
ant, where the elected proposer(s) starts acting as in the
standard version, we say that he is in transient state. Af-
fer completing a sequence of decisions seamlessly, the pro-
poser can switch into steady state. Now the protocol skips
the adopt/adopted message exchange (phase one) and only
performs phase two. In addition, only the slot number is
increased after each round, instead of the version number.
If any problem occurs, the proposer goes back to transient
state.

1.2 Paxos Challenges

The main issue when going from the abstract description to
an actual implementation is managing the acceptors – they
represent the memory of the protocol, preventing compet-
ing proposers to decide different values. To avoid breaking
correctness, a quorum of acceptors must never forget the
version number they accepted or adopted, and the value
they accepted. This is not an issue when dealing with a
single decision, but becomes a major burden when dealing
with multiple decisions. Since memory is finite, a part of
the acceptor state must be discarded. However, discarding
the data related to a decision breaks the correctness prop-
erties of Paxos since it would allow a proposer to issue a
decision with a different value. There are only two ways out
of this. The easier one is to weaken the liveness property,
meaning that proposal for dropped slots are rejected. Oth-
erwise a strongly consistent sub protocol should be added,
that would allow the removal of a slot state once the users
of the system have agreed to never deal again with that
slot. State Paxos eliminates the need of taking care of these
issues.

The remainder of the paper is organized as follows, in Sec-
tion 2 we introduce the State Paxos protocol; in Section 3
we evaluate the performance of State Paxos when applied to
solve a membership problem and compare it with the perfor-
manence provided by Zookeeper; in Section 4 we outline future
work and in Section 5 we provide our concluding remarks.

2. STATE PAXOS

Paxos provides a consensus algorithm that poses the foun-
dations for distributed state machine replication. In this
context, the consensus abstraction serializes events that are
proposed by replicas, while replicas act accordingly to the
decided sequence of events. In State Paxos we do not se-
rialize events, instead we decide on the most recent value
of a state. In practice, the state is represented as a couple
<value,epoch>. The protocol decides the sequence, la-
beled with an epoch, of values taken by the state. In other
terms, the protocol serializes the stream of changes to the
shared state.

As the state has overwrite semantics, there is no point in
knowing past values and only the last one is kept around
by the protocol. It is obvious that this kind of consensus is
not applicable in a setting where the state that should be
decided is too large, unless techniques like hashing and stable
storage are combined. However, a large class of applications
often have to decide on a value of relatively small shared
state. Thus, for these cases the reduced complexity of the
algorithm pays out with respect to the increased network
latency – which for small states is practically negligible.

2.1 Protocol Specification

Now that we have provided the intuition behind State Paxos
we will formally define the algorithm. Processes are divided
into four logical sets: clients, executors, proposers and ac-
ceptors. As for the classical version of Paxos, a quorum as-
sumption is imposed on acceptors, and an eventual leader is
elected among proposers. Clients propose a state value for
an epoch through proposal messages sent to the proposer
leader. On the other side, executors are responsible for re-
ceiving decide messages from proposer(s). Both messages
are <value,epoch> couples.

The version stored-by and exchanged between acceptors and
proposers is composite. The most significant part is as serial
number, the middle part is the identifier of the proposer, and
the least significant part is the epoch number.

snp will denote the proposer current serial number, pidp the
proposer identifier and ep the proposer current epoch. snp, pidp
and ep will denote respectively the last serial number,
proposer identifier and epoch accepted or adopted by the
acceptor. Ep is the value selected or chosen by the proposer
as candidate value and ev is the last value accepted by the
acceptor. Ev and e are the client request value and epoch.

An acceptor adopts an adopt message [snp,pidp, ep] if its
version is greater than the current version < snp, pidp, ev >.
If the epochs are the same (case one), then it replies with an
adopted message [snp,pidp, e, v, snv, pidv], which contains
the adopt message version, and the last accepted value and
version. If the adopt message epoch is greater (case two),
the acceptor resets the last accepted value and replies with
the same adopted message. In both cases it moves its ver-
sion to the new one: < snp, pidp, ep > → < snv, pidv, snv >.

An acceptor accepts an accept message [snv, pidv, ev, v] if

the message version and epoch is greater than or equal to the current version. Accepting means that it moves to the new version, updates the last accepted value with the accept message value \((v^a \leftarrow v^p)\) and replies with an accepted message \([s^{an}, \text{pid}^a, e^a]\). If an acceptor receives either message with a version or epoch that does not match the criteria, it replies with a rejected version message \([s^{an}, \text{pid}^a, e^a]\).

The proposer leader waits for proposal messages from clients. If a proposal has an epoch less than the current one \((e^c < e^p)\), it replies with a rejected epoch message \([e^p]\). If the proposal epoch is equal to the current one \((e^c = e^p)\) and no round is running (case one), then it starts the round as it will be explained later. If a round is running and the candidate value has not been chosen yet (case two), then the proposal is added to the proposals set. Otherwise (case three) the proposal is dropped. If the proposal epoch is greater then the current one \((e^c > e^p)\), then the proposer aborts any running round and moves to this greater epoch \((e^c \leftarrow e^p)\), starting the new round. The proposer goes to phase one, increases the current version \((sn^\pi \leftarrow sn^\pi + 1)\) and sends an adopt message \([sn^\pi, \text{pid}^a, e^\pi]\) to the acceptors. When it receives at least a quorum of adopted messages \([sn^\pi, \text{pid}^a, e^\pi, \ldots]\), it goes to phase two. As for the classical version of Paxos, State Paxos picks a candidate value \(v^\pi\) as follows: it selects the highest versioned non null value among the adopted messages, if none, it chooses a \(v^\pi\) value from the proposal set. It then sends an accept message \([sn^\pi, \text{pid}^a, e^\pi, v^\pi]\). When it receives at least a quorum of accepted messages \([sn^\pi, \text{pid}^a, e^\pi]\) the proposer sends towards executors a decide message \([v^\pi, e^\pi]\) and increases the current epoch \((e^\pi \leftarrow e^\pi + 1)\).

If a proposer receives any rejected version message \([sn^\pi, \text{pid}^a, e^\pi]\) having a version greater than the current one but with the same epoch, (case one) the current round is aborted and restarted, after moving to a version greater than the one from the rejected version message \((sn^\pi \leftarrow sn^\pi + 1)\). If the epoch is greater, but the version is not, (case two) then the current round is aborted, the epoch is updated to one greater than the one from the rejected version message \((e^\pi \leftarrow e^\pi + 1)\) and a rejected epoch message \([e^\pi]\) is sent to the clients. If both version and epoch are greater (case three), the behavior is the same as in case two, but we increase the version as in case one.

A Multi State Paxos can be easily derived from the protocol description provided above and the extensions required for Multi-Paxos as described in [6].

### 2.2 Adding operations and AnyEpoch

In this section we will present two performance and flexibility improvements over the previously presented algorithm. While designing group management we noticed that we could not allow all clients to propose the group view to the State Paxos service, otherwise each join would have been too costly in terms of network traffic and service load. A first solution would have been to elect an eventual leader that would have been responsible for submitting group view changes. However this would have added a round trip latency and may have overloaded a client. Thus we introduced the add and remove operations, as well as read and write. The latter allows to keep the semantics of the vanilla algorithm. Another weak point is that clients must specify the epoch in their proposals, even if they do not care in which epoch their proposals are served. In order to remove this burden, that causes many unnecessary rejections, epoch numbers have been extended with a special value named AnyEpoch.

The proposer state keeps track of the last decided value \((lde^p)\) and epoch \((lde^p)\). The proposal and decide messages are extended with an op field that specifies the client requested operation and the operations performed by the proposer to draw the decided value from the last decided value \((lde^p)\). The read operation simply returns the last decided value \(lde^p\) and epoch \(lde^p\). The write operation overwrites the state value. The add and remove operations manipulate the sub state items. Since they deal with the state concrete representation, their actual implementation is provided by the user. A client can issue multiple times the same proposal, thus the provided implementation must guarantee that adding or removing one or several times the same item has the same effects on the state.

Now we explain the changes with respect to the algorithm presented in subsection 2.1 when selecting the candidate value, and when managing the AnyEpoch case. If the proposer is free to choose the candidate value \(v^\pi\), it selects a write operation from the proposals set and uses the \(v^\pi\) proposal value as candidate value \(v^\pi\). If no write operations are present, then it performs all add and remove operations in the set and the resulting state value is the candidate value \(v^\pi\). If a proposal epoch is AnyEpoch, then either a round is running (case 1), then the proposal is stored in the next round proposals set. Otherwise (case 2), the proposal is added to the proposals set and a round is started for the current epoch. When a decision is taken, if the next round proposals set is not empty, then it is merged into the proposals set and a new round for the following epoch is fired.

### 2.3 Applying State Paxos to Group Management

State Paxos has been designed taking into account that it the most natural application is group management. Here we present how this abstraction has been built on top of of the algorithm explained in subsection 2.2.

The state on which State Paxos takes decision is defined as a set of identifiers. The add and remove operations add and remove the identifier shipped in the proposal value from the current state. The group management client colocates a State Paxos client and executor, in order to send proposals and receive decisions. It keeps track of the group view, which is initially empty. When a decision with an epoch greater than the current one is received, then the group view is overwritten with the decided state value. Given the previous and current view, the leaving and joining members are computed. Events encapsulating their identifiers and the state epoch are fired to the upper layer. Joining and leaving the group is done through the following proposal messages \([\text{identifier}, \text{AnyEpoch}, \text{add}]\) and \([\text{identifier}, \text{AnyEpoch}, \text{remove}]\).

The group management client exposes an interface with the following methods: JOIN, LEAVE, VIEW and SIZE. JOIN and LEAVE methods send the mentioned proposal messages. The VIEW and SIZE methods return the group view and group
view size together with the current group epoch. The user registers a listener to the group management client in order to received group view changes notifications as join and leave events encapsulating a set of identifiers (of joining and leaving members) and the related group epoch. The group epoch, which is the State Paxos state epoch, allows a user of the group management to detect if its group view is stale with respect to the group view of some other interacting process.

3. PERFORMANCES EVALUATION
In this section we compare the performance of State Paxos with those of Zookeeper. Zookeeper [5] is an open-source server developed by the Apache Foundation which enables highly reliable distributed coordination. Zookeeper is based on an two phase commit protocol similar to Paxos. Group membership will be the use case for the tests as is a natural application for both State Paxos (SP) and Zookeeper (ZK). The Zookeeper implementation uses a permanent parent node, while clients add and remove a child node in order to join and leave the group, notifications of the group view changes are received through a watch for child changes on the parent node. For the purpose of performance evaluation, disk operations performed by Zookeeper for logging and snapshots were removed.

3.1 Test Description
To evaluate the relative efficiency of State Paxos and Zookeeper we measure the time required for a client to join/leave a group. Specifically, the group members will perform a series matched join/leave operations. In other terms, a client issues a join, waits to see the new group view as established and then leaves. At this point, it waits to see the new group view established and starts over with a join/leave sequence.

The performance measure is the the latency between a join or leave request and the related group view update.

3.2 Test Scenarios and Testbed
We have run the tests in five configurations with different client populations listed in Table 1. Our testbed is a small cluster of five machines connected by a 1Gbps Ethernet running Linux and equipped with core i7 and 4 to 8 GB of memory. Three of these machines host the servers and the remaining two host the clients – servers and clients are evenly split among allocated machines. All runs consist of a warmup and test phases. The warm-up phase is sufficiently long limit the effects due to cold-start.

<table>
<thead>
<tr>
<th>server configurations</th>
<th>clients populations</th>
</tr>
</thead>
<tbody>
<tr>
<td>single machine</td>
<td>1</td>
</tr>
<tr>
<td>one remote server</td>
<td>2</td>
</tr>
<tr>
<td>two servers</td>
<td>4</td>
</tr>
<tr>
<td>three servers, quorum = 2</td>
<td>8</td>
</tr>
<tr>
<td>three servers, quorum = 3</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 1: Test Scenarios

3.3 Performance Results

3.3.1 Single Machine Results

Figure 1 shows the results for the single machine scenario with Zookeeper (ZK) and three version of State Paxos (SP). SP I is the algorithm as presented in subsection 2.2 and subsection 2.3, while SP II and III have been modified in order to limit to 10KHz and to 2KHz the rate of taken decisions.

Capping the decision frequency enhances requests batching and reduces the load on the network. This mechanism allows, even with a non optimal communication layer\(^1\), to scale out pretty well. The scalability provided by this mechanism is evident in Figure 1 where we can see how SP III, although being the slowest for a single client, becomes the fastest when reaching 16 clients. In addition, Figure 1 shows that, with the exception of SP III with 1 and 2 clients, ZK is always slower that any of the SP variations. Starting with a latency of 0.10 ms and ending up with a latency of 0.82 ms, SP I latency is less than half that of ZK – which ranges from 0.32ms to 2.14 ms. SP II starts with a latency of 0.30 ms, that is almost equal to ZK, and ends up taking a third (0.77 ms with respect to 2.14) of ZK time. SP III pays it’s ability to scale out with 0.55 ms for one client, which is almost the double of ZK, but it has a latency of only 0.59 ms for 32 clients – meaning less than a third of ZK. It turns out that going from 1 to 32 clients results in increases in latencies (from 0.55 to 0.59 ms).

On the other hand, SP III suffers only less than 10% penalty (from 0.32 to 2.14ms), 8x for SP I (from 0.10 to 0.82) and 6x for SP II (from 0.17 to 0.77 ms). We have taken out SP I algorithm since adding network latency would have only worsened it’s scalability issues. However we still kept SP II in order to stress the massive impact that choosing the correct frequency cap have on scalability. In fact this is a parameter that should be chosen depending on the deployment setting. Here we can see that SP II grows slowly, being the quickest algorithm taking from 0.33 to 0.53 ms for 1 to 16 clients, but has a spike reaching 2.56 ms for 32 client. As pointed out in subsubsection 3.3.1, the overload of the network subsystem causes this degradation. It turns out that limiting the decision frequency to 10KHz, and its related batching capabilities augmentation, is not sufficient to cope with the

\(^1\)Our communication layer does not exploit yet network multicas.
client growth from 16 to 32. On the other hand, SP III is really well behaved, having almost constant latency while client population grows. SP II and ZK latencies increase by more than 1.5x from 1 to 32 client, from 0.33 ms to 0.25 ms and from 0.70 ms to 1.83 ms respectively. SP III latencies grow only 2% for 16 clients, and 22% for 32 clients, starting at 0.56 ms for 1 client and reaching 0.68 ms for 32.

3.3.3 Two Servers Results
The following results show the performances while scaling with respect to servers. SP II has been dropped since it is obvious that SP III out performed it. With two servers, both algorithms have a quorum of two, meaning that they both require interactions between the two servers to advance. This costs an additional round-trip for SP in steady state: an accept and an accepted message. ZK instead requires one round trip and a half: a propose, an ACK and a commit message.

SP III displays (Figure 3) a slow growth with respect to ZK, except for a spike reaching 1.05 ms with 4 clients. SP III stepping up by less than 5% from 4 to 32 clients (1.05 ms to 1.13 ms), while ZK steps up by 10-20% (1.35 ms to 2.32 ms). Except for the spikes with 4 and 8 clients, SP III is in average two times faster than ZK.

Comparing these results with the one server configuration (subsubsection 3.3.2) it seems that moving from one server to two servers has more impact on SP III than ZK. The latter increases the latency between 26% (1.83 ms to 2.32 ms for 32 client) and 82% (0.83 ms to 1.52 ms for 8 clients). SP III has almost no penalty for 1 and 2 client (less than 4%), but then grows up from 65% (0.68 ms to 1.13 ms for 32 client) to 94% (0.57 to 1.10 ms for 16 clients).

3.3.4 Three Servers Results
The results with three servers and a quorum of two (Figure 4: ZK Q2 and SP III Q2) are very similar to the previous (subsubsection 3.3.3). In fact adding a third server does not change the quorum size, thus the algorithm must not wait for the slowest server to complete the protocol. However it can be noticed that while SP III adds up only few tens of microseconds, meaning less than 3% in general, ZK adds hundreds of microseconds, which reaches a 18% increase for 32 clients (from 2.33 ms to 2.75 ms).

We forced the quorum value to three in order to simulate the behavior with five servers (Figure 4: ZK Q3 and SP III Q3). In this setting the algorithm must wait for all three servers to complete the protocol, and we would expect a performance degradation.

SP III takes 80% of ZK time for 1 client (1.11 ms to 1.38 ms), but gets down to 38% for 32 client (1.20 ms to 3.18 ms). SP III scales well with respect to clients, paying only a 8% increase in latency when going from 1 to 32 clients (1.11 ms to 1.20 ms). In contrast, ZK pays a 96% increase in latency when going from 1 to 32 clients (from 1.38 ms to 3.18 ms).

Comparing the two configuration with three servers, we notice that the slopes for both algorithms are quite similar. SP III spikes for 2 to 4 clients has moved to 1 client, and now the latency growth seems almost flat. In fact the increase with respect to the previous configuration goes down from 83% for 1 client (from 0.59 ms to 1.11 ms) to around 5% with 32 clients (from 1.14 ms to 1.20 ms). ZK results are really close to the previous one, however the costs of adding the third server is 9% for 1 client (from 1.27 ms to 1.38 ms) and 15% for 32 clients (from 2.75 ms to 3.18 ms).

3.3.5 State Paxos Speedup
Figure 5 shows the speedup of SP III with respect to ZK in the four non local configurations. SP III achieves speedups of 19% – 1 Client on 3 servers with quorum set to two – to 62% – 32 client on single machine or on 3 servers with quorum set to three. In addition, the speedup is quite similar across the four configurations for the same client population, except for 1 and 2 clients.
These results are very encouraging and show the advantages of the number of servers, SP III seems to scale better than ZK with respect to the number of clients. In other words, independently of the number of servers, SP III seems to scale better as the client population does. In other words, independently of the number of clients, SP III seems to scale better than ZK with respect to the number of clients.

In average, for all four configuration the speed up increases as the number of clients.

These results are very encouraging and show the advantages provided by State Paxos in terms of both latency and scalability.

To be fair, we should point out that Zookeeper offers a more complete service and that may add some performance penalties. At the same time it is worth to point out that we have evaluated the first version of State Paxos and through these experimentations we have learned of a few changes that we could implement to further improve its latency and throughput.

4. FUTURE WORKS
The development of State Paxos will follow two main directions: performances optimization and functionality extensions.

Performance Improvements.
Performance can be further enhanced by (1) improving the network utilization, (2) reducing the contention due to the synchronization, and (3) improving batching.

The pressure on the network can be reduced by replacing point-to-point communication with a reliable multicast protocol that leverages IP multicast. This would keep the number of data exchanges practically constant in spite of the number of nodes in the system with great benefits on scalability and latency.

The current implementation of State Paxos uses traditional lock-based synchronization to deal with concurrency. Performances could be further improved through the use of lock-free data structures in the critical path – lock-free data structures would reduce the cost of “synchronization” and improve the level of parallelism.

The natural evolution of State Paxos will follow two main directions: performances optimization and functionality extensions. As we have shown in Section 3, the frequency at which decisions are taken has a great impact on throughput. Currently, the level of batching is configured through a configuration parameter. As a performance and usability improvement we plan to automatically detect the optimal processing fre-


