Priority-Based State Machine Replication with PRaxos

Paulo R. Pinho*, Luciana de Oliveira Rech*, Lau Cheuk Lung*, Miguel Correia†, Lásaro Jonas Camargos‡
*Departamento de Informática e Estatística, Universidade Federal de Santa Catarina - Brazil
†INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, - Portugal
‡Faculdade de Computação, Universidade Federal de Uberlândia - Brazil
prdepinho@gmail.com, luciana.rech@ufsc.br, lau.lung@ufsc.br, miguel.p.correia@ist.utl.pt, lasaro@facom.ufu.br

Abstract—**State machine replication is a form of active replication commonly used to create fault-tolerant distributed services. In a nutshell, the approach consists in ensuring that a set of replicas receive and execute the same sequence of deterministic requests, returning the same results. This approach handles all requests evenly, but for some services it is important to consider that some requests have priority over others, i.e., that whenever two or more requests are ready to be executed, the one with higher priority is executed first. Paxos is perhaps the best known protocol to order requests in asynchronous environments, but Paxos has no notion of priority. In this paper we introduce the notion of *priority-based state machine replication* and modify Paxos to take request priorities into account. The proposed algorithm, PRaxos, works in three steps and satisfies Paxos' safety properties in asynchronous systems, while enforcing priorities when the system behaves synchronously.**

I. Introduction

Computer systems are frequently subject to faults, so dependable systems have to handle such faults, avoiding that they become errors and eventually lead to failures. Fault masking is a common fault tolerance approach. The approach consists in adding redundant processes that may take the job of one another when a process fault happens.

State machine replication (SMR) is a well-known technique to design fault-tolerant distributed services [1]. In SMR a service is implemented by a set of processes called *replicas*. Replicas implement the same service and execute the same sequence of deterministic operations requested by clients, following the same sequence of consistent states and returning identical results. If a few replicas deviate from this behavior, these faults are masked, i.e., concealed from the clients.

For all replicas to execute the same sequence of requests, these requests have to be ordered using a *total order broadcast protocol*. At the heart of this protocol lies the *consensus problem*, i.e., the problem of a group of processes agreeing on a single value among a set of proposals. Total order can be trivially achieved by chaining multiple consensus instances, each one deciding on one request [2]. It has been shown that there is no deterministic fault-tolerant algorithm that solves the consensus problem under the assumption of asynchronous communication [3]. To circumvent this result, algorithms to solve the consensus problem usually guarantee that some safety properties always hold, while termination properties hold only if some extension is made to the assumption of asynchrony. A common extension is a weak timing assumption, such as the existence of some point in time from which failures can be reliably detected [4]. Many algorithms have been proposed to solve this problem for both the crash [5], [6], [7], [8] and Byzantine fault models [9], [10]. One of the best known consensus algorithms is the Synod protocol [6]. Its generalization for total ordering requests, Paxos, guarantees safety while tolerating f faults in a system with $2f+1$ replicas [5].

Dependable systems may have real-time requirements, i.e., they may need that operations are executed before certain rigid deadlines [11]. Solving this problem requires using specialized infrastructures that allow making communication and processing predictable. That is hardly the case in general distributed systems in which unreliable communication may cause messages to be delayed or not delivered. However, in such scenarios it is still possible to attain less strict – *soft* – real-time requirements [12].

Many services may benefit from soft real-time guarantees expressed in terms of different levels of request urgency, i.e., of priority. For instance, different priorities may be assigned to users with distinct service level agreements in cloud storage services [13], [14]. Some examples of services that have been made dependable using SMR and could benefit from priorities are network file systems [9], [10], cooperative backup services [15], and relational database management systems [16].

There have been a few works on the subject of priority-based total order broadcast algorithms [17], [18], [19].The subject first appeared in [17], but the algorithm that work presents is synchronous and not fault-tolerant. Other works consider an asynchronous time model, but depend on an agreement framework based on failure detectors. The algorithm of [18] uses *view atomic multicast* [20], and the algorithm of [19] relies on the *general agreement framework* [21]. These approaches result in a considerable number of messages exchanged between replicas, and as result, they have $O(n^3)$ and $O(n^2)$ message complexity, respectively.

This paper deals with the problem of extending SMR with priorities, letting higher priority be executed before lower priority ones. We call this problem *priority-based state machine replication* – PB-SMR. Regular SMR can be solved by employing a total order broadcast protocol like Paxos to deliver the same sequence of requests to all processes. Likewise, PB-

SMR can be solved using a total ordering protocol that delivers higher priority messages before lower priority ones.

In this paper we propose to modify the Paxos algorithm in order to deal with request priorities, without any assumption stronger than eventual synchrony [22]. The resulting protocol, PRaxos, enforces the ordering of requests satisfying the same safety properties of Paxos, while ensuring priority is respected when the system behaves synchronously. PRaxos aims at solving the priority based total order broadcast in the eventual asynchronous model with linear message complexity and three communication steps.

II. Related Work

Not many works concerning priority-based agreement problems have appeared in literature. The earliest seems to be [17], which proposes two algorithms. One of these algorithms (PriTO) is a priority-based total order broadcast in which messages are delivered according to their priorities. This algorithm depends, however, in a synchronous timing assumption, and is not fault-tolerant. Differently, our approach considers an eventually synchronous model in which there is an unknown global stabilization time after which the system behaves synchronously. Moreover, even if our algorithm requires the global stabilization time to ensure progress, it never violates the safety properties and may even progress before such time arrives.

Another paper proposes an algorithm that is fault-tolerant and considers a failure detector [18]. In that protocol, low-priority client requests are added to each replica sequence at the bottom, assuming requests are ordered from top (high) to bottom (low). High priority requests are inserted in the middle of the sequence, in a position based on the progress of the fastest replica. All replicas have to execute the lower priority requests that the fastest replica had executed so far before they start executing the incoming high-priority request. For that reason, the algorithm does not preempt requests, i.e., it does not interrupt the execution of a low-priority request in order to execute a high-priority one that is ready. The algorithm assumes the existence of a virtual synchrony group communication service that takes care of message transmission, view management, fault-tolerance and atomic broadcast [20]. The absence of preemption capability, resulting in high priority incoming requests having to wait until all replicas finish lower priority requests executed by some replicas before, is the main difference from our approach.

A third work proposes another fault-tolerant algorithm [19]. To achieve this, the system depends on the *general agreement framework* of [21], which uses a $\diamond S$ failure detector [4] to solve consensus. Each time the system receives a client request c, the consensus service is called to define the new request sequence. Requests that are lower priority than c are reallocated to a position after it, and processes must interrupt c and rollback to the previous state in their execution; thus, the algorithm is preemptive. When a replica finishes the execution of a request, the consensus service is called to define the execution progress of the system. When the majority of replicas has finished the execution of a request, that request is irreversible. An irreversible request cannot be rolled back by a higher priority incoming request, and that way the algorithm may terminate.

The main difference between [19] and PRaxos, and the main motivation for this work, is not using the generic agreement framework that causes the algorithm to exchange more messages than PRaxos. In [19], if the system has to execute n requests, the consensus algorithm of the agreement framework will be invoked once for each request plus at least $f+1$ times for each request, in order for the processors to update the group execution progress and commit the executions. In PRaxos, the agreement is invoked only once for each request, and at the end requests are committed. The reason is that priority handling is done along with the agreement protocol.

III. Model and Problem

The system is composed of a set Π of processes, out of which a maximum of $f < \|\Pi\|/2$ may fail. Each process p maintain a sequence $p.S$ of requests, a current proposal number $p.n$ and a current executing request $p.\pi$. A request r in sequence number i is denoted $p.S[1] = r$.

Priority is denoted as $P(r)$, and a request r has higher priority than r' if $P(r) > P(r')$.

The following predicates express states of a request. If process p has accepted request r, then $Accepted(r, p)$ is true. If p learned r, then $Learned(r, p)$ is true, and $r \in \texttt{learned}_p$.

We assume a failure model in which failed processes do not recover. A more realistic failure model, in which process may recover after failing, could be assumed, but we do not do so in this paper to keep the presentation simple.

Inter-replica communication is performed only by message passing. Messages may be lost but if the source and the destination are correct, the message will be eventually delivered. Messages are delivered at most once and are never corrupted.

Finally, we consider an asynchronous model extended with a global stabilization time (GST). This timing assumption is called *eventual synchrony* [23], [24]. The system may behave in an asynchronous manner, in which messages may take arbitrarily long time to be transmitted. The system will eventually enter a stable period in which messages are transmitted synchronously and requests executed in a timely manner. Though progress and ordering may be achieved before the system enters GST, this model is necessary in order to ensure progress. GST periods are long enough for the algorithm to finish, and happen very often in actual systems.

We assume that the execution time of the requests are significantly long, in comparison with the execution time of the protocol. If low priority requests are long, waiting for them to execute may be counterproductive to the system when high priority requests that demand immediate attention are kept delayed.

In order to define the PB-SMR problem we have first to define Local Priority Inversion (LPI) and Global Priority Inversion (GPI), following [19]. We say that a request r is locally (priority) inverted in a process p if r is ready for execution and p is currently executing a request with lower priority than r. A request r is globally (priority) inverted if each process p among all processes that had executed and output the result for r had r locally inverted. This means that if at least one process had not r locally inverted, then r cannot be globally inverted.

In PB-SMR the goal is to broadcast requests to processes that execute them and then respond to the requesting client. To adhere to the nomenclature of the Paxos algorithm, which we use as a basis for our own protocol (see Section IV), we say that a process *broadcasts* a message m by invoking `Propose(m)` and a process p *delivers* a message by appending it to the $\texttt{learned}_p$ sequence, initially empty. The i'th learned element is denoted as $\texttt{learned}_p[i]$.

Our PB-SMR algorithm satisfies the following properties:

- Non-triviality: If request $r \in \texttt{learned}_p$ for any process $p \in \Pi$, then r has been proposed.
- Agreement: If any two processes p and q in Π learned requests in sequence number i, then $\texttt{learned}_p[i] = \texttt{learned}_q[i]$.
- Priority order: Ready requests are chosen according to their priorities.

IV. Paxos

This section presents the Paxos algorithm. The first subsection introduces the consensus algorithm, which we call Synod following [6]. The second introduces the total ordering broacast algorithm, which we call Paxos following [5]. Other authors call these algorithms respectively Paxos and Multi-Paxos, but we prefer to use Lamport's original designations.

A. The Synod protocol

The Synod protocol is a crash fault tolerant consensus protocol that is part of the total order broadcast protocol Paxos. It is defined as a set of *proposers*, *acceptors*, and *learners*. Proposers propose values, acceptors choose one proposed value, and learners learn the chosen value. The requirements are that only one value that had been proposed may be learned (non triviality), and at most one value is learned (consistency). We ensure these safety properties by requiring that the proposer propose a value only if the majority of acceptors had not previously accepted a different value.

The minimum number n of acceptors has a relation with the total number f of faults that the algorithm tolerates among them: $n = 2f+1$. So, being the majority of acceptors $f+1$, if a value v was chosen, then at least one acceptor that accepted it is guaranteed not to fail. While failure is an important aspect among acceptors, it is not so among proposers and learners, because one correct proposer and learner suffice.

The algorithm begins with a proposer p sending a $\langle Propose, n\rangle$ message to all acceptors, where n is a proposal number, used to identify different proposals.

An acceptor a compares n with its own version of $a.n$. If $a.n < n$, then a sets $a.n = n$ and responds sending a $\langle Promise, n, a.v\rangle$ to the proposer, where $a.v$ is the value that a had accepted in a previous proposal numbered $a.v.n$, or $a.v = Null$ if it has not accepted any value at all. By this message the acceptor is saying that it promises not to accept any proposal numbered less than n.

The proposer waits until it has responses from the majority of acceptors. Then it must choose the highest numbered value from the responses. It is done by setting $p.v$ as the value $a.v$, whose $a.v.n$ is greater than any other value among the responses. If all values $a.v = Null$, then the proposer is free to set $p.v$ with any value. Then the proposer then sends $\langle Accept, n, p.v\rangle$ to all acceptors.

Acceptor a receives the accept message from the proposer, and, if $a.n$ is still equal to n, then a *accepts* the value by setting $a.v = p.v$ and sends a $\langle Learn, a.v\rangle$ to all learners.

A learner that received learn messages with the same value from the majority of acceptors *learns* the value, and the algorithm is done.

B. Paxos

The atomic broadcast problem is that in which all broadcast messages need to be delivered in the same order to all processes. Paxos is an algorithm that solves this problem by assigning each message a unique sequence number agreed by all processes via a consensus protocol.

Each process p maintains a sequence $p.S$ of requests. An element $p.S[i]$ is a request r in position i, that was accepted in a proposal number $r.n$.

All processes play the role of proposer, acceptor and learner, but only one proposer may propose at a time. An elected *leader* l plays as the distinguished proposer, and as soon as l is elected, it sends a $\langle Propose, n\rangle$ message to all processes, with n as proposal number. If process p, upon receiving the proposal message, has $p.n < n$, then it sets $p.n = n$ and responds to l sending $\langle Promise, n, p.S\rangle$.

The leader gathers promise messages from the majority of processes with proposal number n. At that point, it has the requests accepted by the majority of processes for each sequence number, and for each sequence number i, l will apply the Synod protocol: it will set $l.S[i]$ as the request r that has the highest proposal number $r.n$ in sequence number i among all sequences received in promise messages. Then, it will send an $\langle Accept, n, l.S[i], i\rangle$ message to all processes.

When the processing is done at all occupied sequence numbers, then the leader starts using vacant sequence numbers to broadcast new processes that it receives from clients. For each of these new requests, the leader sends the accept message like previously described.

A process p that receives the accept message from the leader and still has $p.n = n$, *accepts* the request by setting $p.S[i] = l.S[i]$, and then sends a $\langle Learn, n, p.S[i], i\rangle$ to the leader.

When the leader receives learn messages with the same n and $p.S[i]$ from the majority of processes, it *learns* the request and sends learn messages to all processes. The process that receives this message learns the request.

In a state machine replication system, requests are executed in order. When a process p learns a request $p.S[i]$, it may execute it only if for all $j < i$, p has previously executed $p.S[j]$.

V. PRaxos

Adding priorities to the replicated state machine creates the need of adapting the protocol to deal with the characteristics of the new constraint. As high-priority requests come to the system, previously received low-priority requests need to be put behind the incoming request in the execution sequence. In PRaxos, a request is only accepted after it has been executed by the majority of processes, while other requests are *ready*, and their position may change as higher priority requests come in. Learners will eventually learn chosen values, being the leader the first to learn.

Requests are ordered according to their priority, and executed in that order by processes. When processes are executing a *ready* request, and receive a higher priority request r, then the system enters in *priority inversion*. Processes interrupt all *ready* lower priority requests and roll back to the state previous to their computing. Then they reorder the request sequence, putting r before the others, and resume operation.

As an example, suppose a process has sequence $\{\alpha_1, \beta_4, \gamma_2, \delta_2\}$. Requests α is chosen, β's and γ's execution is complete, and δ is under execution. Their priority is subscript. Then comes a new request ϵ_3. The process interrupts δ and rolls back to the state before the execution of γ. After reordering, the resulting sequence is $\{\alpha_1, \beta_4, \epsilon_3, \gamma_2, \delta_2\}$.

Fig. 1 presents how the protocol looks like when the classes are distributed to the processors. Processor p_1 is the leader and acts as distinguished proposer and learner, and all processes are acceptors and learners. The PRaxos algorithm takes three steps. Processors execute processes after receiving the $Accept$ message and the return value is sent to the requesting client by p_1, after it learned the request. Learning may be achieved by receiving $Learn$ messages from the majority of learners, or by checking the request sequences of each acceptor sent with the $Promise$ messages.

A. Algorithm

All processes in the system, including the leader, act as acceptor and learner, so the sequence $p.S$ in both are the same, but for ease of explanation, their roles shall be treated separately.

1) The Leader: Algorithm 1 describes how the leader works. The leader begins by sending $\langle Propose, n\rangle$ with a proposal number n to all acceptors. A process p with the role of acceptor that receives this message responds if the last proposal number $p.n < n$. The acceptor sets $p.n = n$ and responds by sending $\langle Promise, p.S, n\rangle$, with its requests sequence, to the leader.

The leader waits for the response of all processes (lines 3-6). When the leader receives responses from $f+1$ acceptors, it waits until a timeout expires for the responses of the rest. Then, the next phase begins.

At this point, the leader has a set Q that consists of the sequence $p.S$ from each acceptor in the quorum. It gives the leader knowledge about the requests replicas have received, accepted and learned. The next step is to determinate which requests are chosen and which are ready, which is done as follows.

Algorithm 1 Leader Proposal

```
1:  $n \leftarrow n + 1$
2:  Send $\langle Propose, n\rangle$ to acceptors
3:  repeat
4:      Receive $\langle Accept, p.S, n\rangle$ from acceptor process $p$
5:      $Q \leftarrow Q \cup \{p.S\}$
6:  until time out and $|Q| = f + 1$
7:  if $|Q| \geq f + 1$ then
8:      for $i = 0$ do
9:          if $(\exists R \subseteq Q, |R| \geq f + 1 \wedge \exists r \forall S \in R, S[i] = r) \vee (\exists p.S \in R, Learned(p.S[i], p))$ then
10:             Learn learned request in $(Q, i)$
11:             Send $\langle Learn, n, r, i\rangle$ to other learners
12:         else if $\exists R \subseteq Q, (|R| \geq f + 2 \wedge \neg(\exists S, S' \in R, S[i] = S'[i])) \vee (|R| \geq f + 1 \wedge \forall S \in R, S[i] = Null)$ then
13:             $\mu \leftarrow i$
14:             break for-loop
15:         else
16:             $r \leftarrow S[i] \in Q$ where $\neg(\exists S'[i] \in Q, S'[i].n > S[i].n)$
17:             Send $\langle Accept, n, r, i\rangle$ to acceptors
18:         end if
19:         $i \leftarrow j + 1$
20:     end for
21:     Create a sequence $Ready$ with requests in $Q$ from $\mu$ up, and requests in $Buffer$.
22:     Order $Ready$ according to request priority.
23:     for $i = \mu$ do
24:         $l.S[i] \leftarrow Ready[i]$
25:     end for
26: end if
```

For each sequence number i (line 8), a request r is chosen in i if there are $f+1$ sequences S in Q where $S[i] = r$, or one of the processes had learned r in i (line 9). This case is denoted as $Chosen(i)$. If no request was chosen in i, then one of two cases apply: either there are $f+1$ sequences S in Q where $S[i] = Null$ or there are $f+2$ sequences S in Q where $S[i]$ is set to different requests (line 12). Either way, this case is denoted as $Free(i)$. In spite of the apparent opposition between the two expressions, $\neg Chosen(i)$ is not the same as $Free(i)$, as there is a case in which a sequence number i is both $\neg Chosen(i)$ and $\neg Free(i)$.

The leader takes the following actions according to the state of the sequence number:

1) If $Chosen(i)$, then the leader *learns* the chosen request (line 10).
2) If both $\neg Chosen(i)$ and $\neg Free(i)$, then the leader sends $\langle Accept, n, r, i\rangle$ to all acceptors, where r is the highest numbered request in $S[i]$ among the sequences in Q, that is, with highest $S[i].n$ (lines 16, 17).

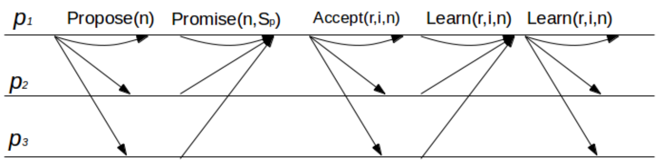

Fig. 1. PRaxos viewed as processors.

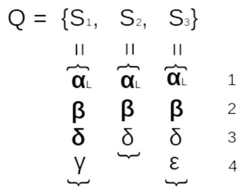

Fig. 2. The leader determines Chosen and Free requests. Bold requests are executed. The capital L besides a request means that it is learned.

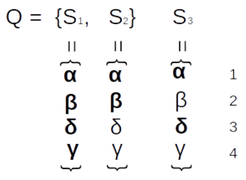

Fig. 3. Example of how Chosen and Free are not equal when not all correct replicas respond to the proposal. Bold requests are executed.

3) If $Free(i)$, then the leader stops the iteration (lines 13, 14).

When a leader *Learns* a request r in position i, then $Learned(r, i)$ is true and the leader sends the message $\langle Learn, n, r, i\rangle$ to the rest of the learners. A learner process that receives this message also learns r in i.

When the leader stops the iteration, a mark $\mu = i$ is set on the sequence (line 13). The sequence from $S[0]$ to $S[\mu - 1]$ is called the $Processed$ section, and the sequence from $S[\mu]$ onward is called the $Ready$ section. Requests in the $Processed$ section do not suffer priority inversion, since no position i in $Processed$ is $Free(i)$, and thus may be $Chosen(i)$. Since no position i in $Ready$ is $Chosen(i)$, then requests in this section may cause priority inversion.

The leader gathers all requests in the $Ready$ section of each S in Q and sequences them along with requests received from clients, ordering them according to priority (lines 21, 22). Then, the leader sends $\langle Accept, r, i, n\rangle$ to all acceptors for each of these requests, starting in position $i = \mu$ (lines 23-25).

Figure 2 shows an example of the procedure. Suppose the system has the total of three processes, which makes $f = 1$, and the leader gathered responses from all of them. In the first iteration, $i = 1$, all processes had executed and learned α, so sequence number 1 is $Chosen(1)$. Similarly, all processes had executed β in sequence number 2, which is also $Chosen(2)$. In sequence number 3, $f+1$ sequences have not executed any request, which makes it $Free(3)$, and $\mu = 3$. The leader stops the iteration here.

To get the resulting sequence, suppose that $P(\delta) = 5$, $P(\gamma) = 3$ and that the leader had buffered the client request ϵ, with priority 4. The resulting sequence would be $S = \{\alpha, \beta, \delta, \epsilon, \gamma\}$.

Figure 3 illustrates another case. Suppose that the system has three processes, and the total tolerated faults are $f = 1$. Processes p_1 and p_2 responded to the proposal by sending promise messages, whereas the response sent by p_3 was lost in transmission. According to the received sequences, α, β and δ are chosen, but δ is not considered chosen by the leader. The predicates $Chosen(1)$ and $Chosen(2)$ are true because α and β were executed and accepted by $f + 1$ replicas in Q. Requests δ and γ were not, however, and since there are no $f + 1$ null accepts or $f + 2$ different accepts in Q for either sequence numbers 3 and 4, $Free(3)$ and $Free(4)$ are false. The leader sends $\langle\delta, 3, n\rangle$ and $\langle\gamma, 4, n\rangle$ to the acceptors and considers them chosen. The $Free$ sequence number in this example is 5, and consequently $\mu = 5$.

Algorithm 2 describes how the leader treats client requests. The leader keeps buffering client requests. If the leader receives a request r whose priority is not greater than that of any request in the $Ready$ section of the sequence, then the leader sends $\langle Accept, r, i, n\rangle$ to the acceptors, where i is the first vacant position in the back of the sequence (lines 6, 7).

When the leader receives a request r that has priority higher than that of any request in $Ready$, then it inserts r in a buffer (lines 3, 4). At a timeout, the leader will start a new proposal by sending $\langle Propose, n + 1\rangle$.

Algorithm 2 Leader main loop

```
1: loop
2:    Receive request $r$ from client $c$.
3:    if $P(r) > P(End(l.S))$ then
4:       $Buffer \leftarrow Buffer \cup \{r\}$
5:    else
6:       append $r$ to $l.S$
7:       $i \leftarrow$ size of $l.S$
8:       send $\langle Accept, n, r, i\rangle$ to acceptors
9:    end if
10: end loop
```

2) The Replica: Replicas receive both $Accept$ and $Learn$ messages from the leader, as they play both roles of acceptor and learner.

An acceptor replica p executes requests in its sequence $p.S$ in the order they appear. When p finishes the execution of request $r = p.S[i]$, it accepts r and sends the leader a message $\langle Learn, p.n, r, i\rangle$. At this point, the request is $Accepted(r, i, p)$. Then p starts executing the request in position $p.S[i + 1]$.

The leader buffers $Learn$ messages from the replicas. When it receives $f + 1$ $\langle Learn, r, i, n\rangle$ with the same parameters, it learns r and sends $\langle Learn, r, i, n\rangle$ to the rest of the learners.

If acceptor p reaches a point in which $p.S[i]$ has no request, but there is a $p.S[i+k]$, k positions further, that has a request, then p sends a message to the leader asking for the missing request in sequence number i.

Algorithm 3 describes how an acceptor accepts requests. When acceptor p receives an $\langle Accept, r, i, n\rangle$ from the leader, it moves all requests in $p.S[j]$ to $p.S[j + 1]$, for j starting in i, up to the end of the sequence (lines 3, 4), to open space to include r in the sequence (line 10). All moved requests that were executed and accepted roll back and become $\neg Accepted(r, i, p)$ (line 8). If p was executing any request that had been moved, then it interrupts the execution (lines 5, 6), rolls back the progress and starts the execution of $p.S[i]$ (lines 11, 12).

Algorithm 3 Acceptor acceptance

```
1:  Receive $\langle Accept, r, i, n\rangle$ from the leader
2:  if $p.n = n$ then
3:      for $j = |p.S|$ down to $i$ do
4:          $p.S[j + 1] \leftarrow p.S[j]$
5:          if Executing $p.S[j]$ then
6:              Interrupt $p.S[j]$
7:          end if
8:          Rollback $p.S[j]$
9:      end for
10:     $p.S[i] \leftarrow r$
11:     if $p.\pi >= i$ then
12:         $p.\pi \leftarrow i$
13:     end if
14: end if
```

When an acceptor p receives $\langle Learn, n, r, i\rangle$ message from the leader, it learns r in i. If $p.S[i] \neq r$, then p sends a message to the leader asking for the request. It will receive an $Accept$ message and should act as previously stated. Then it learns the request.

3) Leader election: The leader is the only process that has the buffered client requests, and if it crashes, these requests are lost. A process has a time limit to communicate with the leader. If it times out, it starts an election protocol. The new leader then starts receiving requests from clients and operates the protocol as specified.

VI. Correctness

This section sketches a proof of correctness of algorithm. We prove that PRaxos satisfies Non-triviality, Agreement and Priority Order properties.

In order to prove Non-triviality and Agreement, we demonstrate two lemmas that are related to the safety properties of the consensus problem.

Lemma 1: Only chosen requests are learned.

It is demonstrated that if a request is learned by any learner, then it was previously chosen by the acceptors. There are four ways a learner process may learn a request r in sequence number i:

1) when it receives $\langle Learn, r, i, n\rangle$ messages from $f + 1$ acceptors;
2) when it receives a $\langle Learn, r, i, n\rangle$ message from the leader;
3) when it is a leader that just proposed and either request r was accepted in i in the sequence of $f + 1$ processes that responded with promise messages,
4) or there was a learned request r in i in the sequence of one process that responded with the promise message.

In the first case, a request executes and accepts r before sending the learn message to the leader. If the leader received this message from $f + 1$ acceptors, then $f + 1$ acceptors had previously executed and accepted it, thus r is chosen.

In the second case, the leader sends $Learn(r, i, n)$ message after it has learned r, and, as demonstrated in the first case, this implies that r is chosen.

In the third case, r is accepted by $f + 1$ acceptors, and is thus chosen.

In the fourth case, r was learned by the majority of processes by the means of the three previous cases, and is therefore chosen.□

Lemma 2: For each sequence position, only one proposed request is chosen.

It is demonstrated by showing that no requests in $Free(i)$ sequence numbers are chosen, and that requests in $\neg Free(i)$ sequence numbers are guaranteed not to be substituted by other requests.

As in Paxos, if the majority of acceptors accept r, the r is *chosen*. Before sending any proposal, the proposer has to ask the majority of acceptor processes for the requests they had accepted so far. Since at least one acceptor that has accepted a chosen request is included in the majority, the proposer is guaranteed to receive all chosen requests in the promise messages, and must thus propose the chosen requests in their respective sequence numbers. In order to propose a different request in sequence number i, the majority gathered by the proposer must not include any acceptor that had previously accepted a request in i. This guarantees that only one request is chosen.

In PRaxos, the leader sends $Accept$ messages for each new request when the request does not suffer priority inversion from any ready request, behaving exactly as Paxos. However, if a request that suffers priority inversion is received, the leader starts a new proposal by sending $Propose$ to the acceptors. In that case, for each sequence number i in the sequence that was $Chosen(i)$, at least one acceptor in the quorum has accepted the chosen request, and the leader has to propose that request in sequence number i. Ready requests are only proposed in sequence number i if $Free(i)$. Thus, requests in $Free(i)$ are not chosen.

There are instances in which the proposer gathered responses from the majority of acceptors in which less than the majority of the acceptors have accepted r in sequence number i, which means that r is not chosen in i. The proposer could make the distinction whether r is chosen or not if:

1) the majority included $f+1$ acceptors that had accepted no request,
2) or if the majority included $f+2$ acceptors that accepted different requests.

In the first case, since the system is composed of $2f+1$ acceptors, if $f+1$ acceptors had accepted no request, the other f would not constitute a majority, and thus no request could have been chosen in i.

In the second case, if $f+2$ acceptors accepted different requests, being one of them r, then even if the rest $f-1$ acceptors had accepted r, the sum f would not constitute a majority, and thus no request had been chosen in i.

These two cases are the definition of $Free(i)$. As a corollary, for each position i that is $\neg Free(i)$ there is an acceptor that accepted a request. In PRaxos, all chosen requests in a sequence number $\neg Free(i)$ are guaranteed not to be substituted by another request, and, thus, for each sequence number, only one proposed request is chosen.□

Lemma 3: Property of Non-triviality: if a request $r \in learned_p$, for any process $p \in \Pi$, then r has been proposed. The proof of this property is derived from the previous lemmas. If a request is learned by a learner, then it was chosen, and if a request was chosen, then it was proposed. Suppose that the sequence of learned requests of process p is $learned_p$, then, if there is a process p and a request r, where request $r \in learned_p$, then p has been previously proposed. □

Lemma 4: Property of Agreement: If two processes p and q in Π learned request r in sequence number i, then $r = \texttt{learned}_p[i] = \texttt{learned}_q[i]$.

The leader is the first to learn a request, and the other processes learn it by receiving learn messages from the leader.

Since request r was chosen in a sequence number i, then it is part of the $Processed$ section of the sequence, and is guaranteed not to change sequence number. So every process that learn r will learn it int i: $\texttt{learned}_p[i] = r$. □

Lemma 5: Property of Priority Order: Ready requests are chosen in order according to their priorities.

The order in which requests are executed and accepted by the acceptors is defined by the leader. Each time a priority inversion occurs because of a high priority request, the leader reorders the $Ready$ section of the sequence, where sequence numbers are $Free(i)$, along with incoming requests, in their appropriate order. Each acceptor process p executes and accepts requests one at a time, from $p.S[0]$ onward. This way, if a request r was chosen in i, then $p.S[i]$ was executed and accepted by $f+1$ acceptors, and before that, $p.S[i-1]$, and so on, until the base case $p.S[0]$. Thus ready requests are chosen in order according to their priorities.

However, not all positions $\neg Free(i)$ are $Chosen(i)$. This means that there may be $Ready$ requests, in the $Chosen$ section of the sequence, that are executed and chosen before higher priority $Ready$ requests, in the $Ready$ section. This case may happen if the leader gathers a majority of exactly $f+1$ acceptors, one of which has accepted request r in position i. Position i does not qualify as $Free(i)$, but request r may not have been chosen, because it may or may not have been accepted by all f replicas that did not participate in the majority, either because of message loss or delay. PRaxos treats these requests as chosen because doing otherwise multiple requests could be chosen in a position, which breaks one of the safety requirements.

Figure 3 illustrates this problem. To the leader, δ and γ are in a similar situation, both have only one accept, but actually, δ is chosen, and γ is not. The leader cannot consider them Free, for then δ could cause a priority inversion and change position, which should not happen to a chosen request. Consequently, the leader considers them $Chosen$, but this may lead γ to cause priority inversion.

Should the system behave in a synchronous way, that is, with a limit on message delivery time, then all correct replicas would receive the $Propose$ message from the leader and respond with $Promise$, successfully delivering their sequences. In that case, all $Chosen(i)$ positions will be $\neg Free(i)$, and ready requests are chosen in order according to their priorities.□

The correctness of the algorithm comes from Lemmata 3, 4 and 5.

VII. Evaluation

This section evaluates PRaxos and compares it with the related works. The analytic evaluation of distributed protocols is usually made in terms of two metrics: number of communication steps and messages sent. In relation to the latter, we count a broadcast as $n-1$ messages, where n is the number of processes. Moreover, we disregard the messages used by the client to send a request to the replicas and the messages sent back by the servers with the replies, as some of the algorithms do not solve SMR and their cost is always the same ($2n$ messages, 2 communication steps).

We evaluate the normal case, i.e., the case in a message is sent, agreed upon, and delivered, without being delayed due to the arrival of higher priority messages, messages being lost or failure suspicions in the algorithms that use failure detectors.

PRaxos follows Paxos' number of communication steps and number of messages. Messages counted are $Accept$, and the two $Learn$ messages. In Paxos, $Propose$ and $Promise$ messages are exchanged by processes only once after the election of the leader. This is enough for the leader to know what requests processes have accepted. The most recurrent messages are, thus, the three last, which adds to $3(n-1)$

TABLE I
COMPARISON OF RELATED ALGORITHMS

Algorithm	Ref	Time model	# communication steps	# messages	Message complexity
PritTO	[17]	synchronous	3	$(n-1)^2$	$O(n^2)$
Rodrigues et al.	[18], [20]	failure detector	2	$(n-1)+n(n-1)^2$	$O(n^3)$
Wang et al.	[19], [21]	failure detector	4	$(2n^3+7n^2+5n-2)/2$	$O(n^3)$
Paxos	[6]	asynchronous	3	$3(n-1)$	$O(n)$
PRaxos	this paper	eventual synchronous	3	$3(n-1)$	$O(n)$

total messages, where n is the total number of processes, and message complexity is linear $O(n)$.

PRaxos follows similarly. $Propose$ and $Promise$ messages are exchanged after leader election and when a priority inversion occurs as a high priority request is sent by a client. Still, the most recurrent messages are the same as in Paxos, and message complexity remains linear.

Table I shows the comparison between PRaxos and the related algorithms. PRaxos has better message complexity than all three compared algorithms. This is consequence of the design, that aimed at being close to Paxos. The algorithm in [18] uses *view atomic multicast* [20] that executes with $(n-1)^2$ messages, and [19] uses the *general agreement framework* [21] that exchanges $n+2n^2$ messages.

VIII. CONCLUSION

We propose a priority-based consensus algorithm for state machine replication that does not require a consensus service and thus works with fewer messages than available algorithms. This was achieved by adapting the Paxos algorithm to take requests priority handling by itself, while maintaining safety requirements from Paxos, in addition to the new constraint. PRaxos is crash fault-tolerant, with resiliency of $2f+1$. This protocol can be used in practice to support priorities in replicated services such as cloud storage services, network file systems, cooperative backup services, and relational database management systems.

Acknowledgment: This work was supported in part by Brazilian National Research Council (CNPq) through process 401364/2014-3, CNPq 455303/2014-2 and by Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013. Miguel Correia is bolsista CAPES/Brasil (project LEAD CLOUDS).

REFERENCES

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[2] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, S. Mullender, Ed. ACM Press/Addison-Wesley, 1993, pp. 97–145.

[3] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[4] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[5] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[6] ——, "The part-time parliament," *ACM Transactions Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[7] B. M. Oki and B. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, 1988, pp. 8–17.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference*, 2014, pp. 305–319.

[9] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999, pp. 173–186.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 45–58.

[11] P. Verissimo and L. Rodrigues, *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[12] C. D. Locke, "Best-effort decision-making for real-time scheduling," Ph.D. dissertation, 1986.

[13] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, 2008, pp. 5–13.

[14] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," in *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*. Saltzburg, Austria: ACM, New York, NY, April 2011, pp. 31–46.

[15] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth, "BAR fault tolerance for cooperative services," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[16] A. F. Luiz, L. C. Lung, and M. Correia, "Byzantine fault-tolerant transaction processing for replicated databases," in *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications*, 2011, pp. 83–90.

[17] A. Nakamura and M. Takizawa, "Priority-based total and semi-total ordering broadcast protocols." in *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, pp. 178–185.

[18] L. Rodrigues, P. Verssimo, and A. Casimiro, "Priority-based totally ordered multicast," in *3rd IFIP/IFAC Workshop on Algorithms and Architectures for Real-Time Control*, 1995.

[19] Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin, "Solving the group priority inversion problem in a timed asynchronous system," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 900–915, 2002.

[20] A. Schiper and A. Ricciardi, "Virtually-synchronous communication based on a weak failure suspector." in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, 1993, pp. 534–543.

[21] M. Hurfin, R. A. Macedo, M. Raynal, and F. Tronel, "A general framework to solve agreement problems," in *Proceeding of the 18th International Symposium on Reliable Distributed Systems*, 1999, pp. 56–65.

[22] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[23] F. Cristian and C. Fetzer, "The timed asynchronous system model," in *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, 1998, pp. 140–149.

[24] C. Fetzer and F. Cristian, "On the possibility of consensus in asynchronous systems," in *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, 1995, pp. 86–91.