Ring Paxos: A High-Throughput Atomic Broadcast Protocol[*]

Parisa Jalili Marandi
University of Lugano
Switzerland

Marco Primi
University of Lugano
Switzerland

Nicolas Schiper
University of Lugano
Switzerland

Fernando Pedone
University of Lugano
Switzerland

Abstract

Atomic broadcast is an important communication primitive often used to implement state-machine replication. Despite the large number of atomic broadcast algorithms proposed in the literature, few papers have discussed how to turn these algorithms into efficient executable protocols. Our main contribution, Ring Paxos, is a protocol derived from Paxos. Ring Paxos inherits the reliability of Paxos and can be implemented very efficiently. We report a detailed performance analysis of Ring Paxos and compare it to other atomic broadcast protocols.

1. Introduction

State-machine replication is a fundamental approach to building fault-tolerant distributed systems [18, 24]. The idea is to replicate a service so that the failure of one or more replicas does not prevent the operational replicas from executing service requests. State-machine replication can be decomposed into two requirements regarding the dissemination of requests to replicas: (i) every nonfaulty replica receives every request and (ii) no two replicas disagree on the order of received requests. These two requirements are often encapsulated in a group communication primitive known as atomic broadcast or total-order broadcast [13].

Since atomic broadcast is at the core of state-machine replication, its performance has an important impact on the overall performance of the replicated service. As a consequence, a lot of effort has been put into designing efficient atomic broadcast algorithms [8]. Despite the large number of atomic broadcast algorithms proposed in the literature, however, few works have considered how to turn them into efficient executable protocols. In this paper, we discuss the implementation of a highly efficient atomic broadcast protocol. Although the discussion focuses on atomic broadcast in a clustered system, some of the ideas are general enough to be used as guidelines in other contexts.

We are interested in efficiency as a measure of throughput. More precisely, we define the *maximum throughput efficiency (MTE)* of an atomic broadcast protocol as the rate between its maximum achieved throughput per receiver and the nominal transmission capacity of the system per receiver. For example, a protocol that has maximum delivery throughput of 500 Mbps in a system equipped with a gigabit network has an MTE of 0.5, or 50%. An ideal protocol would have an MTE of 1. Due to inherent limitations of an algorithm, implementation details, and various overheads (e.g., added by the network layers), ideal efficiency is unlikely to be achieved.

This paper presents Ring Paxos, a highly efficient atomic broadcast protocol. Ring Paxos is based on Paxos and inherits many of its characteristics: it is safe under asynchronous assumptions, live under weak synchronous assumptions, and resiliency-optimal, that is, it requires a majority of nonfaulty processes to ensure progress. We revisit Paxos in light of a number of optimizations and from these we derive Ring Paxos. Our main design considerations result from a careful use of network-level multicast (i.e., ip-multicast) and a ring overlay.

Network-level multicast is a powerful communication primitive to propagate messages to a set of processes in a cluster. As shown in Figure 1, ip-multicast can provide high message throughput when compared to unicast communication (point-to-point). This happens for two reasons. First, ip-multicast delegates to the interconnect (i.e., ethernet switch) the work of transferring messages to each one of the destinations. Second, to propagate a message to all destinations there is only a single system call and context switch from the user process to the operating system, as opposed to one system call and context switch per destination, as with unicast. For example, with 10 receivers, ip-multicast provides almost 10 times the throughput of unicast.

However, ip-multicast communication is unreliable, i.e., subject to message losses. In modern networks, unreliability comes mostly from messages dropped due to buffer overflow. By carefully configuring maximum ip-multicast sending rates and communication buffer sizes, one can minimize such losses. The situation becomes more problematic

[*]This work was partially funded by the Swiss National Science Foundation (#200021-121931), and by the Hasler Foundation (#2316).

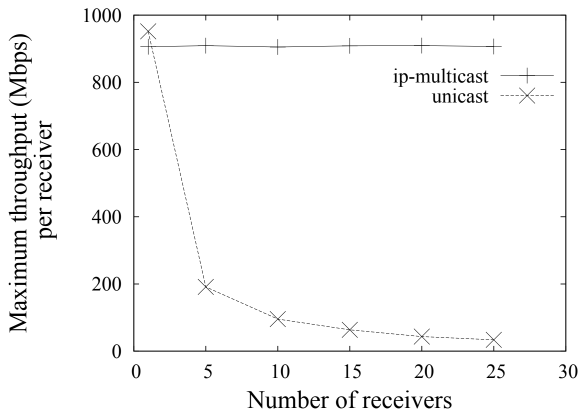


Figure 1. ip-multicast versus unicast (udp) with one sender (hardware setup c.f. Section 6)

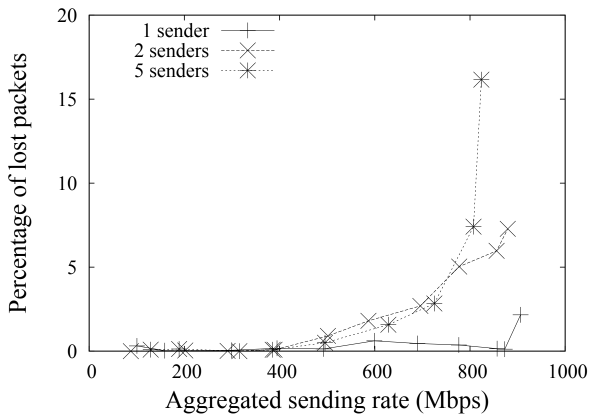


Figure 2. one versus multiple simultaneous ip-multicast senders (14 receivers)

if multiple nodes can simultaneously use ip-multicast since synchronizing distributed senders, to reduce collisions and avoid buffer overflow, is a difficult task, if possible at all.

Figure 2 illustrates the problem with concurrent ip-multicast senders. The figure shows experiments with 1, 2 and 5 senders; the aggregated sending rate is uniformly distributed among the senders. Although ip-multicast is quite reliable with a single sender, reliability decreases quickly with two and five senders (e.g., more than 15% of packets lost with five senders at high sending rates). Notice that even low percentages of packet loss can negatively impact the system since they will result in retransmissions, which will make the situation worse.

Ring Paxos is motivated by the observations above. It uses a single ip-multicast stream to disseminate messages and thus benefit from the throughput that ip-multicast can provide without falling prey to its shortcomings. To evenly balance the incoming and outgoing communication needed to totally order messages, Ring Paxos places $f + 1$ nodes in a logical ring, where f is the number of tolerated failures. Ring Paxos is not the first atomic broadcast protocol to place nodes in a logical ring (e.g., Totem [2], LCR [21] and the protocol in [10] have done it before), but it is the first to achieve very high throughput while providing low latency, almost constant with the number of receivers.

We have built a prototype of Ring Paxos and compared it to other atomic broadcast protocols. In particular, Ring Paxos can reach an MTE of 90% in a gigabit network, while keeping delivery latency below 5 msec. Moreover, both throughput and latency remain approximately constant with an increasing number of receivers (up to 25 receivers in our experiments). Previous implementations of the Paxos protocol, based either on ip-multicast only or on unicast only, have an MTE below 5%. The only other protocol that achieves a high MTE we know of is LCR [21], a pure ring based protocol. With 5 nodes in the ring LCR has an MTE of 95%, but it has a latency that increases linearly with the number of receivers and relies on stronger synchronous assumptions than Ring Paxos.

Briefly, this paper makes the following contributions: First, it proposes a novel atomic broadcast algorithm for clustered networks derived from Paxos. Second, it describes an implementation of this algorithm. Third, it analyses its performance and compares it to other atomic broadcast protocols.

The remainder of the paper is structured as follows. Section 2 describes our system model and the definition of atomic broadcast. Sections 3 and 4 review Paxos and present Ring Paxos, respectively. Section 5 comments on related work. Section 6 evaluates the performance of Ring Paxos and compares it quantitatively to a number of other protocols. Section 7 concludes the paper.

2. Model and definitions

2.1. System model

We assume a distributed crash-recovery model in which processes communicate by exchanging messages. Processes can fail by crashing but never perform incorrect actions (i.e., no Byzantine failures). Processes have access to stable storage whose state survives failures.

Communication can be one-to-one, through the primitives *send*(p, m) and *receive*(m), and one-to-many, through the primitives *ip-multicast*(g, m) and *ip-deliver*(m), where m is a message, p is a process, and g is a group of processes. Messages can be lost but not corrupted. In the text we refer sometimes to ip-multicast messages as *packets*.

Ring Paxos, like Paxos, ensures safety under both asynchronous and synchronous execution periods. The FLP impossibility result [11] states that under asynchronous assumptions consensus and atomic broadcast cannot be both safe and live. We thus assume the system is *partially synchronous* [9], that is, it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [9], and it is unknown to the processes.

Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown. Moreover, in order to prove liveness, we assume that *after* GST all remaining processes are *correct*—a process that is not correct is *faulty*. A correct process is operational "forever" and can reliably exchange messages with other correct processes. Notice that in practice, "forever" means long enough for consensus to terminate.

2.2. Consensus and atomic broadcast

Consensus and atomic broadcast are two distributed agreement problems at the core of state-machine replication. The problems are related: atomic broadcast can be implemented using a sequence of consensus executions [5]. Consensus is defined by the primitives *propose*(v) and *decide*(v), where v is an arbitrary value; atomic broadcast is defined by the primitives *broadcast*(m) and *deliver*(m), where m is a message.

Consensus guarantees that (i) if a process decides v then some process proposed v; (ii) no two processes decide different values; and (iii) if one (or more) correct process proposes a value then eventually some value is decided by all correct processes. Atomic broadcast guarantees that (i) if a process delivers m, then all correct processes deliver m; (ii) no two processes deliver any two messages in different orders; and (iii) if a correct process broadcasts m, then all correct processes deliver m.

3. Paxos

Paxos is a fault-tolerant consensus algorithm intended for state-machine replication [19]. We describe next how a value is decided in a single instance of consensus.

Paxos distinguishes three roles: *proposers*, *acceptors*, and *learners*. A process can execute any of these roles, and multiple roles simultaneously. Proposers propose a value, acceptors choose a value, and learners learn the decided value. Hereafter, N_a denotes the set of acceptors, N_l the set of learners, and Q_a a *majority quorum* of acceptors (*m-quorum*), that is, a subset of N_a of size $\lceil(|N_a|+1)/2\rceil$.

The execution of one consensus instance proceeds in a sequence of *rounds*, identified by a round number, a positive integer. For each round, one process, typically among the proposers or acceptors, plays the role of *coordinator* of the round. To propose a value, proposers send the value to the coordinator. The coordinator maintains two variables: (a) $c\text{-}rnd$ is the highest-numbered round that the coordinator has started; and (b) $c\text{-}val$ is the value that the coordinator has picked for round $c\text{-}rnd$. The first is initialized to 0 and the second to null.

Acceptors maintain three variables: (a) rnd is the highest-numbered round in which the acceptor has participated, initially 0; (b) $v\text{-}rnd$ is the highest-numbered round in which the acceptor has cast a vote, initially 0—it follows that $rnd \leq v\text{-}rnd$ always holds; and (c) $v\text{-}val$ is the value voted by the acceptor in round $v\text{-}rnd$, initially null.

```
1:  **Algorithm 1: Paxos**
2:  Task 1 (coordinator)
3:  upon receiving value $v$ from proposer
4:      increase $c\text{-}rnd$ to an arbitrary unique value
5:      for all $p \in N_a$ do send ($p$, (PHASE 1A, $c\text{-}rnd$))
6:  Task 2 (acceptor)
7:  upon receiving (PHASE 1A, $c\text{-}rnd$) from coordinator
8:      if $c\text{-}rnd > rnd$ then
9:          let $rnd$ be $c\text{-}rnd$
10:         send (coordinator, (PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$))
11: Task 3 (coordinator)
12: upon receiving (PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$) from $Q_a$
                                              such that $c\text{-}rnd = rnd$
13:     let $k$ be the largest $v\text{-}rnd$ value received
14:     let $V$ be the set of ($v\text{-}rnd$,$v\text{-}val$) received with $v\text{-}rnd = k$
15:     if $k = 0$ then let $c\text{-}val$ be $v$
16:     else let $c\text{-}val$ be the only $v\text{-}val$ in $V$
17:     for all $p \in N_a$ do send (p, (PHASE 2A, $c\text{-}rnd$, $c\text{-}val$))
18: Task 4 (acceptor)
19: upon receiving (PHASE 2A, $c\text{-}rnd$, $c\text{-}val$) from coordinator
20:     if $c\text{-}rnd \geq rnd$ then
21:         let $v\text{-}rnd$ be $c\text{-}rnd$
22:         let $v\text{-}val$ be $c\text{-}val$
23:         send (coordinator, (PHASE 2B, $v\text{-}rnd$, $v\text{-}val$))
24: Task 5 (coordinator)
25: upon receiving (PHASE 2B, $v\text{-}rnd$, $v\text{-}val$) from $Q_a$
26:     if for all received messages: $v\text{-}rnd = c\text{-}rnd$ then
27:         for all $p \in N_l$ do send (p, (DECISION, $v\text{-}val$))
```

Algorithm 1 provides an overview of Paxos. The algorithm has two phases. To execute Phase 1, the coordinator picks a round number $c\text{-}rnd$ greater than any value it has picked so far, and sends it to the acceptors (Task 1). Upon receiving such a message (Task 2), an acceptor checks whether the round proposed by the coordinator is greater than any round it has received so far; if so, the acceptor "promises" not to accept any future message with a round smaller than $c\text{-}rnd$. The acceptor then replies to the coordinator with the highest-numbered round in which it has cast a vote, if any, and the value it voted for. Notice that the coordinator does not send any proposal in Phase 1.

The coordinator starts Phase 2 after receiving a reply from an m-quorum (Task 3). Before proposing a value in Phase 2, the coordinator checks whether some acceptor has already cast a vote in a previous round. This mechanism guarantees that only one value can be chosen in an instance of consensus. If an acceptor has voted for a value in a previous round, then the coordinator will propose this value; otherwise, if no acceptor has cast a vote in a previous round, then the coordinator can propose the value received from the proposer. In some cases it may happen that more than one acceptor have cast a vote in a previous round. In this case, the coordinator picks the value that was voted for in the highest-numbered round. From the algorithm, two acceptors cannot cast votes for different values in the same round.

An acceptor will vote for a value $c\text{-}val$ with corresponding round $c\text{-}rnd$ in Phase 2 if the acceptor has not received any Phase 1 message for a higher round (Task 4). Voting for a value means setting the acceptor's variables $v\text{-}rnd$ and $v\text{-}val$ to the values sent by the coordinator. If the acceptor votes for the value received, it replies to the coordinator. When the coordinator receives replies from an m-quorum (Task 5), it knows that a value has been decided and sends the decision to the learners.

In order to know whether their values have been decided, proposers are typically also learners. If a proposer does not learn its proposed value after a certain time (e.g., because its message to the coordinator was lost), it proposes the value again. As long as a nonfaulty coordinator is eventually selected, there is a majority quorum of nonfaulty acceptors, and at least one nonfaulty proposer, every consensus instance will eventually decide on a value.

Algorithm 1 can be optimized in a number of ways [19]. The coordinator can execute Phase 1 before a value is received from a proposer. In doing so, once the coordinator receives a value from a proposer, consensus can be reached in four communication steps, as opposed to six. Moreover, if acceptors send Phase 2B messages directly to the learners, the number of communication steps for a decision is further reduced to three (see Figure 3(a)).

4. Ring Paxos

Ring Paxos is a variation of Paxos, optimized for clustered systems. In Section 4.1 we explain Ring Paxos assuming a fixed coordinator, no process crashes, and no message losses. In Section 4.2 we revisit these assumptions, and in Section 4.3 we describe a number of optimizations. A proof of correctness sketch is presented in the Appendix.

4.1. Normal operation

Algorithm 2 presents Ring Paxos; statements in gray are the same for Paxos and Ring Paxos. As in Paxos, the execution is divided in two phases. Moreover, the mechanism to ensure that only one value can be decided in an instance of consensus is the same as in Paxos.

```
1:  Algorithm 2: Ring Paxos
2:  Task 1 (coordinator)
3:  upon receiving value $v$ from proposer
4:    increase $c\text{-}rnd$ to an arbitrary unique value
5:    let $c\text{-}ring$ be an overlay ring with processes in $Q_a$
6:    for all $p \in Q_a$ do send (p, (PHASE 1A, $c\text{-}rnd$, $c\text{-}ring$))
7:  Task 2 (acceptor)
8:  upon receiving (PHASE 1A,$c\text{-}rnd$,$c\text{-}ring$) from coordinator
9:    if $c\text{-}rnd > rnd$ then
10:     let $rnd$ be $c\text{-}rnd$
11:     let $ring$ be $c\text{-}ring$
12:     send (coordinator, (PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$))
13: Task 3 (coordinator)
14: upon receiving (PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$) from $Q_a$
                                        such that $rnd = c\text{-}rnd$
15:   let $k$ be the largest $v\text{-}rnd$ value received
16:   let $V$ be the set of ($v\text{-}rnd$,$v\text{-}val$) received with $v\text{-}rnd = k$
17:   if $k = 0$ then let $c\text{-}val$ be $v$
18:   else let $c\text{-}val$ be the only $v\text{-}val$ in $V$
19:   let $c\text{-}vid$ be a unique identifier for $c\text{-}val$
20:   ip-multicast ($Q_a \cup N_l$, (PHASE 2A, $c\text{-}rnd$, $c\text{-}val$, $c\text{-}vid$))
21: Task 4 (acceptor)
22: upon ip-delivering (PHASE 2A, $c\text{-}rnd$, $c\text{-}val$, $c\text{-}vid$)
23:   if $c\text{-}rnd \geq rnd$ then
24:     let $v\text{-}rnd$ be $c\text{-}rnd$
25:     let $v\text{-}val$ be $c\text{-}val$
26:     let $v\text{-}vid$ be $c\text{-}vid$
27:     if $first(ring)$ then
28:       send (successor, (PHASE 2B, $c\text{-}rnd$, $c\text{-}vid$))
29: Task 5 (coordinator and acceptors)
30: upon receiving (PHASE 2B,$c\text{-}rnd$,$c\text{-}vid$) from predecessor
31:   if $v\text{-}vid = c\text{-}vid$ then
32:     if not $last(ring)$ then
33:       send (successor, (PHASE 2B, $c\text{-}rnd$, $c\text{-}vid$))
34:     else
35:       ip-multicast ($Q_a \cup N_l$, (DECISION, $c\text{-}vid$))
```

Differently than Paxos, Ring Paxos disposes a majority quorum of acceptors in a *logical directed ring* (see Figure 3(b)(c)). The coordinator also plays the role of acceptor in Ring Paxos, and it is the last process in the ring. Placing acceptors in a ring reduces the number of incoming messages at the coordinator and balances the communication among acceptors. When the coordinator starts Phase 1 (Task 1), it proposes the ring to be used in Phase 2. The proposed ring is stored by the coordinator in variable $c\text{-}ring$.

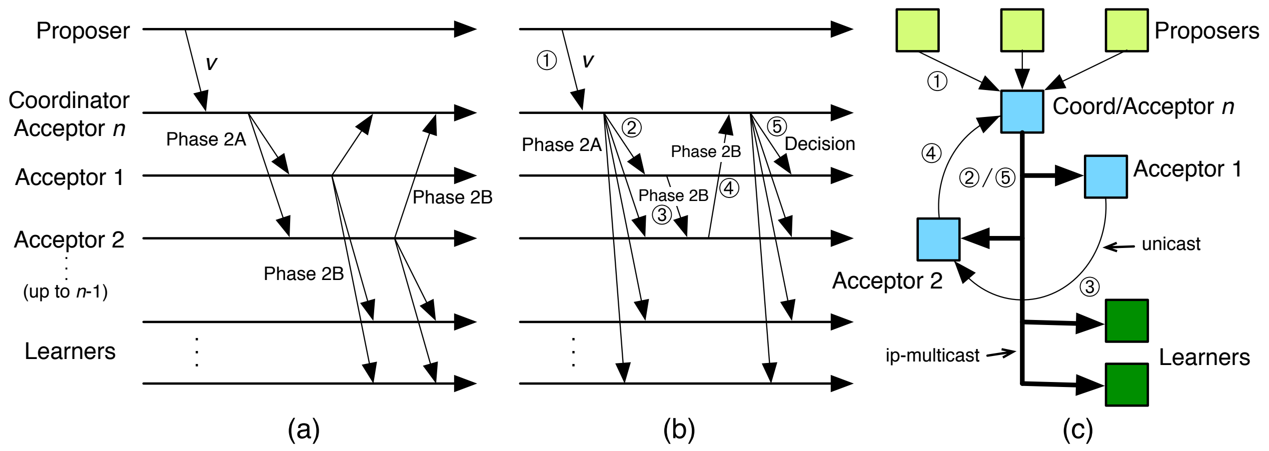


Figure 3. Optimized Paxos (a) and Ring Paxos (b,c)

By replying to the coordinator, the acceptors implicitly acknowledge that they abide by the proposed ring (Task 2).

In addition to checking what value can be proposed in Phase 2 (Task 3), the coordinator also creates a unique identifier for the value to be proposed. Ring Paxos executes consensus on value ids [10, 20]; proposed values are disseminated to the m-quorum and to the learners in Phase 2A messages using ip-multicast. Upon ip-delivering a Phase 2A message (Task 4), an acceptor checks that it can vote for the proposed value. If so, it updates its $v\text{-}rnd$ and $v\text{-}val$ variables, as in Paxos, and its $v\text{-}vid$ variable. Variable $v\text{-}vid$ contains the unique identifier of the proposed value; it is initialized with null. The first acceptor in the ring sends a Phase 2B message to its successor in the ring. Although learners also ip-deliver the proposed value, they do not learn it since it has not been accepted yet.

The next acceptor in the ring to receive a Phase 2B message (Task 5) checks whether it has ip-delivered the value proposed by the coordinator in a Phase 2A message. This is done by comparing the acceptor's $v\text{-}vid$ variable to the value's identifier calculated by the coordinator. If the condition holds, then there are two possibilities: either the acceptor is not the last process in the ring (i.e., it is not the coordinator), in which case it sends a Phase 2B message to its successor in the ring, or it is the coordinator and then it ip-multicasts a decision message including the identifier of the chosen value. Once a learner ip-delivers this message, it can learn the value received previously from the coordinator in the Phase 2A message.

4.2. Handling abnormal cases

A failed coordinator is detected by the other processes, which select a new coordinator. Before GST (see Section 2.1) it is possible that multiple coordinators co-exist. As Paxos, Ring Paxos guarantees safety even when multiple coordinators execute at the same time, but it may not guarantee liveness. After GST, eventually a single correct coordinator is selected.

Lost messages are retransmitted. If the coordinator does not receive a response to its Phase 1A and Phase 2A messages, it re-sends them, possibly with a bigger round number. Eventually the coordinator will receive a response or will suspect the failure of an acceptor. To recover from an acceptor failure, the coordinator re-executes Phase 1 and lays out a new ring, excluding the failed acceptor.

When an acceptor replies to a Phase 1A or to a Phase 2A message, it must not forget its state (i.e., variables rnd, $ring$, $v\text{-}rnd$, $v\text{-}val$, and $v\text{-}vid$) despite failures. There are two ways to ensure this. First, by requiring a majority of acceptors to never fail. Second, by requiring acceptors to keep their state on stable storage before replying to Phases 1A and 2A messages.

Message losses may cause learners to receive only the value proposed and not the notification that it was accepted, only the notification without the value, or none of them. Learners can recover lost messages by inquiring other processes. Ring Paxos assigns each learner to a preferencial acceptor in the ring, which the learner can ask for lost messages.

4.3. Optimizations

We introduce a few optimizations in Ring Paxos, most of which have been described previously in the literature: when a new coordinator is elected, it executes Phase 1 for a number of consensus instances [19]; Phase 2 is executed for a batch of proposed values, and not for a single value (e.g.,

[16]); one consensus instance can be started before the previous one has finished [19].

Placing an m-quorum in the ring reduces the number of communication steps to reach a decision. The remaining acceptors are spares, used only when an acceptor in the ring fails.[1] Finally, although ip-multicast is used by the coordinator in Tasks 3 and 5, this can be implemented more efficiently by overlapping consecutive consensus instances, such that the message sent by Task 5 of consensus instance i is ip-multicast together with the message sent by Task 3 of consensus instance $i + 1$.

5. Related work

In this section we review atomic broadcast algorithms and compare them analytically to Ring Paxos.

Several papers argued that Paxos is not an easy algorithm to implement [4, 16]. Essentially, this is because Paxos is a subtle algorithm that leaves many non-trivial design decisions open. Besides providing insight into these matters, these two papers present performance results of their Paxos implementations. In contrast to Ring Paxos, these prototypes implement Paxos as specified in [19]; no algorithmic modifications are considered.

Paxos is not the only algorithm to implement atomic broadcast. In fact, the literature on the subject is abundant. In [8], five classes of atomic broadcast algorithms have been identified: fixed sequencer, moving sequencer, destination agreement, communication history-based, and privilege-based.

In fixed sequencer algorithms (e.g., [3, 14]), broadcast messages are sent to a distinguished process, called the sequencer, who is responsible for ordering these messages. The role of sequencer is unique and only transferred to another process in case of failure of the current sequencer. In this class of algorithms, the sequencer may eventually become the system bottleneck.

Moving sequencer protocols are based on the observation that rotating the role of the sequencer distributes the load associated with ordering messages among processes. The ability to order messages is passed from process to process using a *token*. The majority of moving sequencer algorithms are optimizations of [6]. These protocols differ in the way the token circulates in the system: in some protocols the token is propagated along a ring [6, 7], in others, the token is passed to the least loaded process [15]. All the moving sequencer protocols we are aware of are based on the broadcast-broadcast communication pattern. According to this pattern, to atomically broadcast a message m, m is broadcast to all processes in the system; the token holder process then replies by broadcasting a unique global sequence number for m. As argued in Section 1, allowing multiple processes to broadcast at the same time leads to message loss, which hurts performance.

Protocols falling in the destination agreement class compute the message order in a distributed fashion (e.g., [5, 12]). These protocols typically exchange a quadratic number of messages for each message broadcast, and thus are not good candidates for high throughput.

In communication history-based algorithms, the message ordering is determined by the message sender, that is, the process that broadcasts the message (e.g., [17, 23]). Message ordering is usually provided using logical or physical time. Of special interest is LCR, which arranges processes along a ring and uses vector clocks for message ordering [21]. This protocol has slightly better throughput than Ring Paxos but exhibits a higher latency, which increases linearly with the number of processes in the ring, and requires *perfect failure detection*: erroneously suspecting a process to have crashed is not tolerated. Perfect failure detection implies strong synchrony assumptions about processing and message transmission times.

The last class of atomic broadcast algorithms, denoted as privilege-based, allows a single process to broadcast messages at a time; the message order is thus defined by the broadcaster. Similarly to moving sequencer algorithms, the privilege to order messages circulates among broadcasters in the form of a token; Differently from moving sequencer algorithms, message ordering is provided by the broadcasters and not by the sequencers. In [2], the authors propose Totem, a protocol based on a group membership service, which is responsible for reconstructing the ring and regenerating the token in case of process or network failures. In [10], fault-tolerance is provided by relying on a failure detector. However, tolerating f process failures requires a quadratic number of processes. A general drawback of privilege-based protocols is their high latency: before a process p can totally order a message m, p must receive the token, which delays m's delivery.

Ring Paxos combines ideas from several broadcast protocols to provide high throughput and low latency. In this sense, it fits multiple classes, as defined above. To ensure high throughput, Ring Paxos decouples message dissemination from ordering. The former is accomplished using ip-multicast; the latter is done using consensus on message identifiers. To use the network efficiently, processes executing consensus communicate using a ring, similarly to the majority of privilege-based protocols. To provide reasonable latency, the ring is composed of only $f + 1$ processes, and is reconfigured in the case of failure.

In Table 1, we compare algorithms that are closest to Ring Paxos in terms of throughput efficiency. All these protocols use a logical ring for process communication, which

[1]This idea is conceptually similar to Cheap Paxos [20], although Cheap Paxos uses a reduced set of acceptors in order to save hardware resources, and not to reduce latency.

Algorithm	Class	Communication steps	Number of processes	Synchrony assumption
LCR [21]	comm. history	$2f$	$f+1$	strong
Totem [2]	privilege	$(4f+3)$	$2f+1$	weak
Ring+FD [10]	privilege	(f^2+2f)	$f(f+1)+1$	weak
Ring Paxos	—	$(f+3)$	$2f+1$	weak

Table 1. Comparison of atomic broadcast algorithms (f: number of tolerated failures)

appears to be the best communication pattern when optimizing for throughput. For each algorithm, we report its class, the minimum number of communication steps required by the last process to deliver a message, the number of processes required as a function of f, and the synchrony assumption needed for correctness.

With Ring-Paxos, message delivery occurs as soon as messages make one revolution around the ring. Its latency is $f+3$ message delays since each message is first sent to the coordinator, circulates around the ring of $f+1$ processes, and is delivered after the final ip-multicast is received. In contrast, LCR requires two revolutions and thus presents a two-fold increase in latency. In Totem, each message must also rotate twice along the ring to guarantee "safe-delivery", a property equivalent to uniform agreement: if a process (correct or not) delivers a message m then all correct processes eventually deliver m. Moreover, Totem puts twice as many processes in the ring as Ring-Paxos, its latency is thus multiplied by a factor of four. The atomic broadcast protocol in [10] has a latency that is quadratic in f since a ring requires more than f^2 nodes.

6. Performance evaluation

In this section we comment on our Ring Paxos prototype, and then detail its experimental evaluation. We consider the performance of Ring Paxos in the presence of message losses and in the absence of process failures. Process failures are hopefully rare events; message losses happen relatively often because of high network traffic.

We ran the experiments in a cluster of Dell SC1435 servers equipped with 2 dual-core AMD-Opteron 2.0 GHz CPUs and 4GB of main memory. The servers were interconnected through an HP ProCurve2900-48G Gigabit switch (0.1 msec of round-trip time). Each experiment (i.e., point in the graph) was repeated 3 to 10 times, with a few million messages broadcast in each one.

6.1. Implementation

Each process maintains a circular buffer of packets; each packet is 8 kbytes long and the buffer is 160 Mbytes long. The coordinator uses this buffer to re-transmit lost ip-multicast messages; the acceptors and the learners use it to match proposal ids to proposal contents, as these are decomposed by the coordinator. Messages received out of sequence (e.g., because of transmission losses) are stored in the buffer until they can be delivered (i.e., learned) in order.

Each packet ip-multicast by the coordinator is composed of two parts. In one part the coordinator stores the ids of decided values, and in the second part it stores new proposed values with their unique ids. The coordinator implements a flow control mechanism, which depends on its buffer size. New values are proposed as long as there is space in the buffer. A buffer entry is freed after the coordinator has received the entry's corresponding Phase 2B message from its neighbor and ip-multicast a decision message related to the entry. The coordinator can re-use a free buffer entry if the entries that succeed it in the buffer are also free (i.e., we avoid holes, which would render garbage collection more complex).

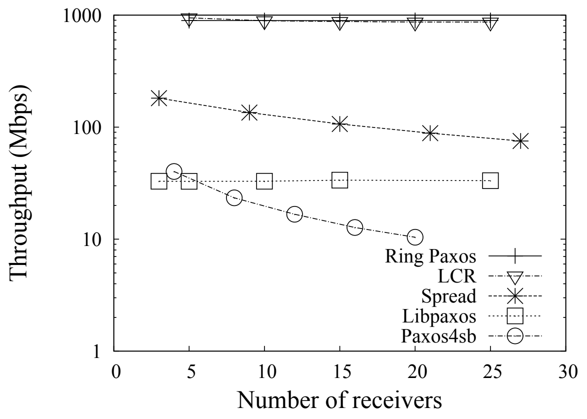
6.2. Ring Paxos versus other protocols

We experimentally compare Ring Paxos to other four atomic broadcast protocols: LCR [21], Spread [1], Libpaxos [22], and the protocol presented in [16], which hereafter we refer to as Paxos4sb. LCR is a ring-based protocol that achieves very high throughput (see also Section 5). Spread is one of the most-used group communication toolkits. It is based on Totem [2]. Libpaxos and Paxos4sb are implementations of Paxos. The first is entirely based on ip-multicast; the second is based on unicast.

We implemented all protocols, except for Spread and Paxos4sb. We tuned Spread for the best performance we could achieve after varying the number of daemons, number of readers and writers and their locations in the network, and the message size. In the experiments that we report we used a configuration with 3 daemons in the same segment, one writer per daemon, and a number of readers evenly distributed among the daemons. The performance data of Paxos4sb was taken from [16]. The setup reported in [16] has slightly more powerful processors than the ones used in our experiments, but both setups use a gigabit switch. Libpaxos is an open-source Paxos implementation developed by our research group.

Figure 4 shows the throughput in megabits per second (left graph) and the number of messages delivered per second (right graph) as the number of receivers increases. In both graphs the y-axis is in log scale. For all protocols, with the exception of Paxos4sb, we explored the space of message sizes and selected the value corresponding to the best throughput (c.f. Table 2).

The graph on the left of Figure 4 roughly places protocols into two distinct groups, one group at the top of



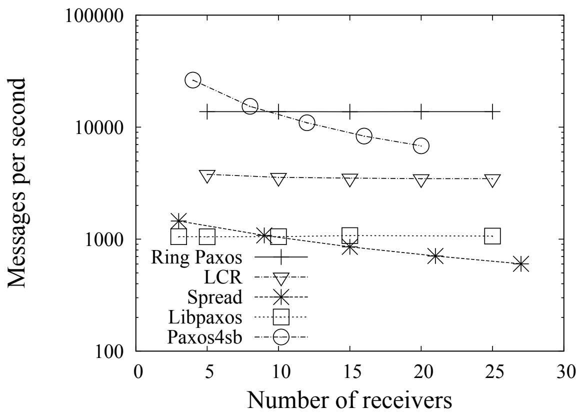


Figure 4. Ring Paxos and other atomic broadcast protocols (message sizes c.f. Table 2)

the graph and the other group at the middle of the graph. The difference in throughput between protocols in the two groups is about one order of magnitude. Notice that protocols based on a ring only (LCR), on ip-multicast (Libpaxos), and on both (Ring Paxos) present throughput approximately constant with the number of receivers. Because it relies on multiple ip-multicast streams, however, Libpaxos has lower throughput (c.f. Section 1).

Protocol	MTE	Message size
LCR	95%	32 kbytes
Ring Paxos	**90%**	**8 kbytes**
Spread	18%	16 kbytes
Paxos4sb	4%	200 bytes
Libpaxos	3%	4 kbytes

Table 2. Atomic broadcast protocols (MTE is the Maximum Throughput Efficiency)

6.3. Impact of processes in the ring

We consider now how the number of processes affects the throughput and latency of Ring Paxos and LCR (Figure 5). LCR does not distinguish process roles: all processes must be in the ring. Ring Paxos places a majority of acceptors in the ring; the remaining acceptors are spare, used in case of failure. The x-axis of the graph shows the total number of processes in the ring of LCR and Ring Paxos.

Ring Paxos has constant throughput with the number of processes in the ring. LCR's throughput slightly decreases as processes are added. With respect to latency, LCR degrades linearly with the number of processes. Ring Paxos also presents a slight increase in latency as more acceptors

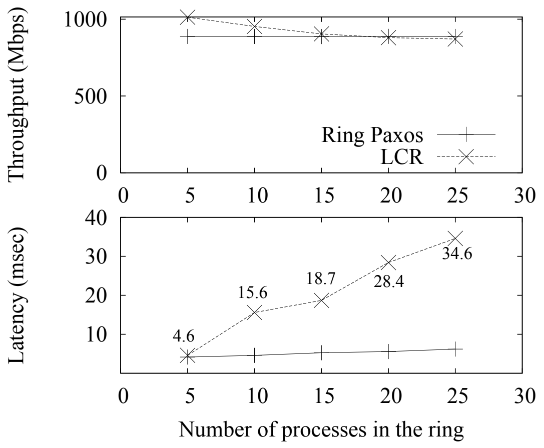


Figure 5. Varying number of processes

are placed in the ring, however, latency values remain low (i.e., in the range of 4.2 and 6.2 msec). The main difference between the rings in the two protocols is that the amount of information that circulates in the ring of Ring Paxos is minimal. In LCR, the content of each message is sent $n - 1$ times, where n is the number of processes in the ring. Message content is propagated only once in Ring Paxos (using ip-multicast). The average CPU usage per process in LCR is in the range of 65%–70% (see Section 6.6 for Ring Paxos).

6.4. Impact of message size

Figure 6 quantifies the effects of application message size (payload) on the performance of Ring Paxos. Throughput (top left graph) increases with the size of application messages, up to 8 kbyte messages, after which it decreases. We attribute this to the fact that in our prototype ip-multicast packets are 8 kbytes long, but datagrams are fragmented since the maximum transmission unit (MTU) in our network is 1500 bytes. Latency is less sensitive to application message size (top right graph). Figure 6 also shows the number of application messages delivered as a function of their size (bottom left graph). Many small application messages can fit a single ip-multicast packet (from Section 4.3, Phase 2 is executed for a batch of proposed values), and as a consequence, many of them can be delivered per time unit (left-most bar). Small messages, however, do not lead to high throughput since they result in high overhead, leading to a low rate of ip-multicast packets per second (bottom right graph).

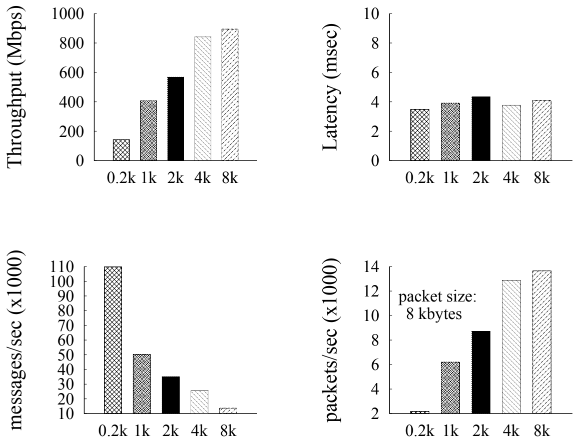


Figure 6. Impact of application message size

6.5. Impact of socket buffer sizes

The reliability of unicast and ip-multicast depends on the size of the buffers allocated to communication. Lost messages have a negative impact on Ring Paxos, as they result in retransmissions. Figure 7 shows the effect of socket buffer sizes on the maximum throughput (left graph) and latency (right graph) of Ring Paxos. We have used 16 Mbytes as socket buffer sizes in all previous experiments, as they provide the best tradeoff between throughput and latency.

6.6. CPU and memory usage

Table 3 shows the CPU and memory usage of Ring Paxos for peak throughput. In the experiments we isolated the process running Ring Paxos in a single processor and measured its usage. Not surprisingly, the coordinator is the process with the maximum load since it should both receive a large stream of values from the proposers and ip-multicast these values. Memory consumption at coordinator, acceptors and learners is mostly used by the circular buffer of proposed values. For simplicity, in our prototype the buffer is statically allocated.

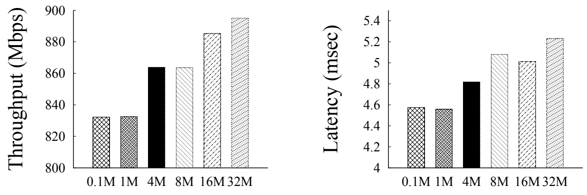


Figure 7. Impact of socket buffer size

Role	CPU	Memory	
Proposer	37.2%	2.2%	(90 Mbytes)
Coordinator	88.0%	4.1%	(168 Mbytes)
Acceptor	24.0%	4.1%	(168 Mbytes)
Learner	21.3%	4.1%	(168 Mbytes)

Table 3. CPU and memory usage

7. Conclusions

This paper presents Ring Paxos, a Paxos-like algorithm designed for high throughput. Ring Paxos is based on characteristics of modern interconnects. In order to show that Ring Paxos can be effective, we implemented it and compared it to other atomic broadcast protocols. Our selection of protocols includes a variety of techniques typically used to implement atomic broadcast. It revealed that both Ring based and ip-multicast-based protocols have the property of providing constant throughput with the numer of receivers, an important feature in clustered environments. It points out the tradeoffs with pure Ring based protocols, which result in increasing latency, and possibly additional synchronous assumptions. Protocols based on unicast only or ip-multicast only have low latency, but poor throughput. The study suggests that a combination of techniques, Ring Paxos, can lead to the best of both: high throughput and low latency, with weak synchronous assumptions. Ring Paxos is available for download at [22].

8. Acknowledgements

The authors would like to thank Antonio Carzaniga, Leslie Lamport, and Robbert van Renesse for the valuable discussions about Ring Paxos; and the reviewers for their suggestions to improve the paper.

References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread toolkit: Architecture and performance. Technical report, Johns Hopkins University, 2004. CNDS-2004-1.

[2] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, 1995.

[3] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[4] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on principles of distributed computing (PODC)*, pages 398–407, 2007.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[6] J.-M. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.

[7] F. Cristian and S. Mishra. The Pinwheel asynchronous atomic broadcast protocols. In *International Symposium on Autonomous Decentralized Systems (ISADS)*, Phoenix, Arizona, USA, 1995.

[8] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys,*, 36(4):372–421, Dec. 2004.

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[10] R. Ekwall, A. Schiper, and P. Urbán. Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 52–65, 2004.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *J. ACM*, 32(2):374–382, 1985.

[12] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of International Symposium on Reliable Distributed Systems (SRDS)*, pages 578–585, 1998.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[14] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *11th International Conference on Distributed Computing Systems (ICDCS)*, pages 222–230, Washington, USA, 1991.

[15] J. Kim and C. Kim. A total ordering protocol using a dynamic token-passing scheme. *Distributed Systems Engineering*, 4(2):87–95, 1997.

[16] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, pages 1–6, 2008.

[17] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[19] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[20] L. Lamport and M. Massa. Cheap Paxos. In *International Conference on Dependable Systems and Networks (DSN)*, pages 307–314, 2004.

[21] R. Levy. *The complexity of reliable distributed storage*. PhD thesis, EPFL, 2008.

[22] http://libpaxos.sourceforge.net.

[23] T. Ng. Ordered broadcasts for large applications. In *Symposium on Reliable Distributed Systems (SRDS)*, pages 188–197, 1991.

[24] F. B. Schneider. What good are models and what models are good? In S. Mullender, editor, *Distributed Systems*, chapter 2. Addison-Wesley, 2nd edition, 1993.

A. Correctness Proof (Sketch)

We provide a proof sketch of the correctness of Ring Paxos. We focus on properties (ii) and (iii) of consensus. Property (i) holds trivially from the algorithm.

Proposition 1 *(ii) No two processes decide different values.*
Proof sketch: Let v and v' be two decided values, and $v\text{-}id$ and $v'\text{-}id$ their unique identifiers. We prove that $v\text{-}id = v'\text{-}id$. Let r (r') be the round in which some coordinator c (c') ip-multicast a decision message with $v\text{-}id$ ($v'\text{-}id$).

In Ring Paxos, c ip-multicasts a decision message with $v\text{-}id$ after: (a) c receives f+1 messages of the form (Phase 1B, r, *, *); (b) c selects the value $v_{val} = v$ with the highest round number v_{rnd} among the set M_{1B} of phase 1B messages received, or picking a value v if $v_{rnd} = 0$; (c) c ip-multicasts (Phase 2A, r, v, $v\text{-}id$); and (d) c receives (Phase2B, r, $v\text{-}id$) from the second last process in the ring, say q. When c receives this message from q, it is equivalent to c receiving f+1 (Phase 2B, r, $v\text{-}id$) messages directly because the ring is composed of f+1 acceptors. Let M_{2B} be the set of f+1 phase 2B messages. Now consider that coordinator c received the same set of messages M_{1B} and M_{2B} in a system where all processes ran Paxos on value identifiers. In this case, c would send a decide message with $v\text{-}id$ as well. Since the same reasoning can be applied to coordinator c', and Paxos implements consensus, $v\text{-}id = v'\text{-}id$. □

Proposition 2 *(iii) If one (or more) process proposes a value and does not crash then eventually some value is decided by all correct processes.*
Proof sketch: After GST, processes eventually select a correct coordinator c. c considers a ring c-ring composed entirely of correct acceptors, and c sends a message of the form (Phase 1A, *, c-ring) to the acceptors in c-ring. Because after GST, all processes are correct and all messages exchanged between correct processes are received, all correct processes eventually decide some value. □