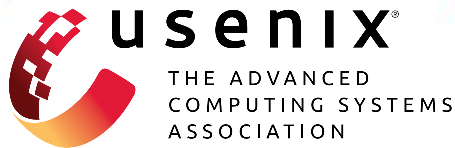


High Throughput Replication with Integrated Membership Management

Pedro Fouto, Nuno Preguiça, and João Leitão, ***NOVA LINCS & NOVA University Lisbon***

https://www.usenix.org/conference/atc22/presentation/fouto

This paper is included in the Proceedings of the
2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

**Open access to the Proceedings of the
2022 USENIX Annual Technical Conference
is sponsored by**

High Throughput Replication with Integrated Membership Management *

Pedro Fouto, Nuno Preguiça, João Leitão
NOVA LINCS & NOVA University Lisbon

Abstract

This paper introduces ChainPaxos, a new distributed consensus algorithm for high throughput replication. ChainPaxos organizes nodes in a chain, allowing for a pipeline communication pattern that maximizes throughput, by minimizing the number of messages transmitted. While other proposals have explored such patterns, ChainPaxos is the first that can execute linearizable reads in any replica with no communication overhead, relying only on information used to process updates. These techniques build on a fully specified integrated membership management solution, allowing ChainPaxos's fault-tolerance to be independent of an external coordination service, often used in other solutions, which can lead to possible safety violations in the presence of network partitions.

Our evaluation shows that, when compared with alternative Paxos variants, ChainPaxos exhibits significantly higher throughput and scalability with negligible latency impact. Compared to other solutions with similar communication patterns, besides avoiding the costs of an external coordination service, ChainPaxos's high throughput tends to increase with the ratio of read-only operations.

1 Introduction

Fault-tolerance is a key property for distributed systems, being fundamental to guarantee that they continue to operate despite failures of individual components. To achieve this, the state of the system needs to be replicated over multiple nodes.

A particularly interesting way of providing fault-tolerance is the state machine replication (SMR) [20, 30] approach, which allows to replicate any service providing strong consistency. SMR is achieved by executing the same sequence of deterministic operations on all replicas, making them transition through the same sequence of states.

*This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under the projects NG-STORAGE (PTDC/CCI-INF/32038/2017) and NOVA LINCS (grant UIDB/04516/2020).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr/).

The Paxos [15, 32] consensus protocol and its variants [6, 21, 24, 26–28] have been used as a fundamental building block for implementing SMR, by enabling replicas to agree on the order in which operations are executed. Many practical systems, such as coordination systems, scale-out, in-memory lock services and in-memory databases rely on the performance of their underlying SMR implementation, making it extremely relevant to improve the performance of consensus (or agreement) protocols.

This paper describes the design and implementation of ChainPaxos, a new consensus algorithm for high throughput replication of (deterministic) services. Our goal is to minimize the communication cost of the protocol to achieve the highest possible throughput, both for read and write operations. We achieve this by using a set of complementary techniques. For write performance, we rely on an efficient pipelined communication pattern between replicas, which has been explored and shown effective by previous approaches, notably ChainReplication [33]. This pattern allows to minimize and distribute the number of messages propagated (and therefore processed) by each node to achieve consensus, which highly contributes to maximizing the throughput of write operations. For read operations, we propose a novel scheme for linearizable reads served by a single replica, without incurring in additional communication cost (albeit at the cost of a small increase in latency), which further minimizes the communication overhead of ChainPaxos and increase its throughput.

Contrary to many recent proposals, ChainPaxos does not outsource membership management to an external coordination service (e.g., Zookeeper [12]). Instead, our system features its own integrated membership management solution that allows for the continuous execution of operations during reconfigurations, while uncoupling our system's fault-tolerance from that of an external service. As such, increasing the number of replicas in ChainPaxos effectively increases the maximum number of faults that are tolerated. On the other hand, when leveraging an external coordination service, the fault-tolerance of the replicated systems depends on the fault-tolerance of that coordination service. Additionally, as shown

recently [2], relying on an external coordination service is far from trivial as it makes a system more vulnerable to network partitions, requiring additional logic to ensure correctness.

Our design builds on the insight that it is possible to combine multiple Multi-Paxos messages and exploit a pipeline communication pattern. While this insight is not novel [19], ChainPaxos is, to the best of our knowledge, the first protocol that not only takes advantage of this insight, but also specifies an integrated membership management solution, allowing for lightweight linearizable reads to be served by any replica.

We conduct an extensive experimental evaluation of ChainPaxos where we evaluate its performance against several state-of-the-art variants of Paxos, in particular Multi-Paxos, Egalitarian Paxos, (U)-Ring Paxos, and Chain Replication. The results show that our algorithm provides higher throughput and scalability when compared with other Paxos variants, with even higher gains as both the number of the replicas of the system and the size of operations increase. When compared with other solutions that employ pipeline communication, ChainPaxos shows similar performance for writes, but improved scalability as the ratio of reads increases.

In summary, this paper makes the following main contributions: **i)** a new consensus algorithm that provides high throughput, with integrated membership management that makes it independent from external coordination services (Section 3); **ii)** a novel approach to provide linearizable read operations that distribute the load among all replicas without incurring additional communication costs (Section 4); and **iii)** an extensive experimental evaluation, showing that ChainPaxos provides better performance than state-of-the-art alternatives (Section 5).

2 Related Work

Paxos [15,32] and its variants [6,21,24,26–28] have been used in the design of replicated systems, employing diverse techniques to optimize performance aspects, such as *minimizing latency*, *reducing communication cost*, *distributing the load*, and supporting *linearizable reads*. We now discuss the most popular variants of Paxos, along with Chain Replication [33], an alternative SMR algorithm.

2.1 Minimizing latency

Multiple Paxos variants try to optimize latency. In Fast-Paxos [18], clients send Accept messages directly to acceptors, skipping the leader. Generalized Paxos [17] extends FastPaxos by allowing non-interfering requests to execute in different orders. In both cases, collisions in client requests result in additional round trips, hindering performance. Flexible Paxos [11] uses different quorum sizes for executing operations (akin to read-write quorum systems [34]), reducing the size of accept quorums and decreasing the latency of accepting operations in the fast path. While ChainPaxos only matches the fast path latency of these protocols when configured to tolerate a single fault, it requires each replica to handle only one message per operation. In contrast, the $O(n)$ message complexity of Paxos leader and learners in these solutions results in lower throughput.

2.2 Communication cost

Some variants employ chain (or ring) topologies to decrease communication cost. Ring Paxos [28] sends *Accept* messages to all replicas using IP-multicast, with responses being propagated through a ring. IP-multicast limits the operation of the protocol across data centers and negatively impacts the performance under high load when messages are lost. Chain Replication [33] is an SMR algorithm, developed for synchronous systems, where replicas are organized in a *chain* and write operations are forwarded from the head to the tail, with acknowledge messages travelling the opposite way. This approach has the advantage that all replicas send and receive the same number of messages for executing an update. Similarly, U-Ring Paxos [13] propagates messages in a ring topology, with acknowledge messages being forwarded from the tail to the head. These solutions require an external coordination service (e.g., Zookeeper [12]) to reconfigure the system when faults occur, leading to higher operational cost, slower fault-handling, potentially lower fault-tolerance (dependent on that of the external service), and vulnerability to network partitions [2]. ChainPaxos, while having a similar communication pattern, further reduces the number of messages processed by each replica, while handling reconfigurations and faults in an integrated and efficient way.

2.3 Distributing the load

Other variants of Paxos try to distribute the load across replicas. Mencius [22] pre-assigns the leader of each instance to a different node. While providing better throughput, the overall availability suffers since the failure of *any* replica will cause the system to stop until another replica takes over. In Egalitarian Paxos (EPaxos) [26], any replica can commit operations and non-conflicting operations execute in different orders. When there is no conflict, operations commit in a single communication round. Multi-Ring Paxos [24] (based on Ring Paxos) uses a similar approach, while taking advantage of the ring topology to minimize communication. When concurrent operations conflict (which is often the case in SMR), these protocols require extra rounds of communication. Atlas [8] improves on this by allowing some conflicting operations to execute in a single round. Despite trying to distribute the load across all replicas, these protocols still require nodes to send and receive $O(n)$ messages. In contrast, ChainPaxos minimizes the overall load imposed by the protocol, having $O(1)$ message complexity, while also distributing the load across replicas.

2.4 Linearizable reads

In replicated systems, where reads are more frequent than writes, it is important to reduce the cost of read operations to improve overall performance.

Synchronous systems. Chain Replication [33] proposes to execute linearizable reads by contacting a single node: the tail of the chain. When the tail fails, as detected by an external coordination service, clients fallback to the previous node in the chain to continue reading the system state. This solution, however, was designed for a synchronous model model where failures can be reliably detected. In an asynchronous system, linearizability can be violated, as the tail can become isolated and be excluded from the chain without knowing, while still serving (outdated) reads. To avoid this, for each read, either the tail or the client would need to contact the external coordination service to verify the current configuration of the chain, which is too expensive. In [33], the authors mention that the coordination mechanism needs to stop clients during reconfigurations, which is unfeasible under network partitions.

Asynchronous systems. Due to a similar reason, solutions based on Paxos cannot execute read operations by contacting only the leader, usually requiring to run a consensus instance for ordering read operations or, in special cases, to contact a quorum of replicas. However, some alternative read schemes to improve replication performance have been proposed. In Smarter [4], reads execute on a single replica, but require a special *whats_my_view* message to be sent to all replicas to gather a majority of replies confirming that no reconfiguration took place concurrently with the read operation. In [10], reads are executed on a single replica, however at the cost of requiring writes to execute in two phases. CRAQ [31] improves reads in Chain Replication by allowing to read from any replica in an asynchronous model, however it only provides per-object linearizability, and for SMR it would require all reads to contact the tail whenever there is a write executing.

In contrast with all these solutions, ChainPaxos includes a novel technique to execute linearizable reads on a single replica, in an asynchronous environment, without ever requiring any additional communication costs. Furthermore, it allows any replica to process reads, thus distributing the read load across all replicas.

3 ChainPaxos

We assume an asynchronous distributed system with n nodes, connected by a network that can lose, duplicate, and deliver messages out of order. Nodes communicate by exchanging messages over a network with a fair loss model that allows the creation of FIFO channels between any pair of nodes. Nodes can fail by crashing, where they stop sending messages.

We follow the SMR model [30], in which each replica holds a copy of the system state and there exists a set of deterministic operations that may output a reply. Replicas start in the same initial state and apply the same sequence of operations, thus guaranteeing that all replicas transition through the same sequence of states and output the same results. We defer the processing of read operations to Section 4.

ChainPaxos is used to order the execution of operations. The system state includes the application state and the membership of the system, with `AddNode(n)` and `RemoveNode(n)` operations, respectively, adding and removing node `n` to the replica-set. These operations execute in the state machine, as other application operations, potentially impacting the quorum size of following operations. For correctness, a node can only decide a given instance strictly after knowing the decision of all previous instances (and the current membership).

3.1 Overview

This section introduces ChainPaxos, revisiting Multi-Paxos and Chain Replication to better contextualize our design.

In Multi-Paxos [15, 32], a distinguished proposer, known as *leader*, prepares multiple Paxos instances in a single step (Phase 1), followed by multiple sequential executions of Phase 2 of Paxos. In a fault-free run (Figure 1), the leader sends an *accept* message to all replicas, with each replying to all replicas with an *accept ack* message. Any replica that receives *accept ack* messages from a majority of replicas can decide and execute the request (with the replica that received the operation replying back to the client). With n replicas, the message complexity of the protocol is $O(n^2)$: each replica incurs in $O(n)$ message overhead (the leader sends/receives $2n$ messages). The reply to the client is produced after 2 communication steps between replicas. Alternatively, a replica could send the *accept ack* message only to the leader, which would then forward the decision to all replicas. In that case, the overhead of non-leader replicas decreases to $O(1)$ at the cost of an additional communication step.

Chain Replication [33] leverages a chain topology, forwarding operations from the head to the tail (Figure 2). The tail replies to clients after executing an operation, and sends *ack* messages backwards, to allow replicas to perform garbage collection. In a fault-free run, each replica incurs in $O(1)$ message overhead, with a reply being produced after $O(n)$ communication steps.

The main goals of ChainPaxos's design are: (i) minimize the number of messages each node processes in fault-free runs and make the load uniform, maximizing throughput; and (ii) integrate an efficient fault handling scheme into the algorithm, by taking advantage of Paxos messages, avoiding the need to rely on an external service. To achieve these goals, we leverage the chain topology to combine and forward multiple Multi-Paxos messages in a single ChainPaxos message. As ChainPaxos builds on Multi-Paxos, leader faults can be handled simply by falling back to the first phase of Paxos.

In ChainPaxos, in a fault-free run (Figure 3), the leader sends the *accept* message, including its *accept ack*, to the following replica in the chain. Upon receiving an *accept* message, a replica forwards the message modified to include its own *accept ack*. When the *accept* message reaches the tail of the chain, it sends a message directly to the head with the *accept ack* of all replicas, guaranteeing that the head learns about the decided value. Additionally, it is necessary to in-

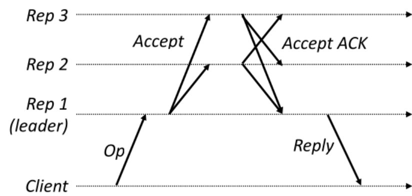


Figure 1: Multi-Paxos message flow on a fault-free run.

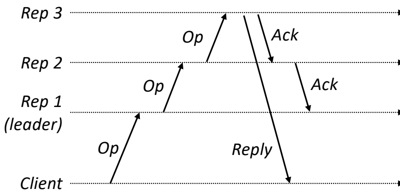


Figure 2: Chain Replication message flow on a fault-free run.

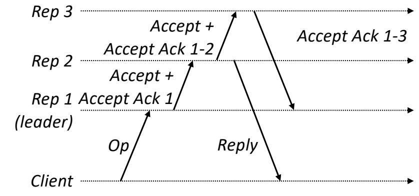


Figure 3: ChainPaxos message flow on a fault-free run.

form the replicas that have not received enough *accept ack* messages to decide the value of the instance – ChainPaxos piggybacks this information in the next *accept* message.

When an *accept* reaches the replica at the middle of the chain, it includes *accept ack* from a majority quorum. Thus, the replica knows that the received request has been decided, and can execute the request and return the result to the client. In the example of Figure 3, with three replicas, the leader and replica 2 form a quorum, with replica 2 replying to the client.

The message flow for fault-free runs achieves the first goal of minimizing the number of messages handled by each replica and keeping the load uniform: a single message is sent and received by every replica. As ChainPaxos is just using a different communication pattern to convey the messages of Multi-Paxos, it can fall back to the regular two phases of Paxos to handle faults. This is the base for achieving the second goal of integrating fault handling in the protocol.

ChainPaxos builds on these ideas to provide high throughput replication by addressing the following challenges: *i*) optimize fault-handling and integrate membership management by leveraging information about the chain topology, thus avoiding the common vulnerabilities/complexity encountered in systems that rely on external coordination services in the presence of network partitions [2]; *ii*) support efficient garbage collection of the information about decided values, which is a common challenge in many variants of Paxos, rarely addressed in the specification of algorithms; and *iii*) integrate a novel mechanism that leverages the chain topology to enable efficient linearizable read operations handled by a single replica without additional communication.

Next, we detail the operation of ChainPaxos, describing the state maintained by each replica and the operation of the protocol in fault-free runs and during reconfigurations. We present correctness arguments for our solution in Annex A.

3.2 Protocol State

Algorithm 1 presents the state of each replica. The first variable group is related with the organization of the system and includes: the members and their order in the chain (`chain`); the identity of the local node (`self`); the next node in the chain that is not marked for removal ($\texttt{c}_{\texttt{nextok}}$); the currently supported leader ($\texttt{c}_{\texttt{sleader}}$); and the replicas for which a `RemoveNode` has been received but not yet decided (`marked`).

Algorithm 1 State of ChainPaxos nodes.

`chain` : *array of nodes*	
`self` : *node*	▷ local node identifier
$\texttt{c}_{\texttt{nextok}}$: *node*	▷ next (unmarked) node in the chain
$\texttt{c}_{\texttt{sleader}}$: *node*	▷ supported chain leader
`marked` : *set of node*	▷ nodes marked for removal (init : ∅)
$\texttt{np}_{\texttt{leader}}$: *int*	▷ special prepare number of the leader
`inst` : *map int* × *PaxosInst*	▷ PaxosInst : ($\texttt{n}_\texttt{a}$, `val`, $\texttt{n}_{\texttt{acpts}}$, `decided`)
`submitted` : *set of requests*	▷ requests submitted by the client
`pending` : *set of requests*	▷ requests waiting to execute (leader only)
$\texttt{max}_{\texttt{ack}}$: *int*	▷ highest instance acknowledged
$\texttt{max}_{\texttt{acpt}}$: *int*	▷ highest leader initiated instance (leader only)
`amLeader` : *bool*	▷ true if current leader

The second group maintains the information to run Paxos instances. This includes the prepare number ($\texttt{np}_{\texttt{leader}}$) that the leader can use for bypassing the first phase of Paxos. Each replica also maintains a map (`inst`) with the information of Paxos instances including, for each instance, the highest prepare number ($\texttt{n}_\texttt{a}$) used by a leader to accept a value (`val`), the number of nodes that accepted `val` with $\texttt{n}_\texttt{a}$ ($\texttt{n}_{\texttt{acepts}}$), and a boolean indicating if the instance was decided (`decided`).

The third group is used for managing client requests. It consists of two sets: `submitted` stores requests received from clients and not yet decided, and `pending` contains the requests received by the leader (redirected from itself or other replicas) but not yet submitted for ordering.

The final group of variables is used for clarity of presentation and stores information that could be derived from other variables, including the highest instance started by the leader ($\texttt{max}_{\texttt{acpt}}$), and the highest instance known to have a decided value accepted by all nodes ($\texttt{max}_{\texttt{ack}}$). Each node also keeps track of whether it is the current leader in `amLeader`.

3.3 Fault-free execution

Algorithm 2 presents the ChainPaxos algorithm, with auxiliary functions detailed in Algorithm 3. The highlighted lines represent the logic used in faulty scenarios that require reconfiguration, which are detailed in the next section.

Requests from clients can be received by any replica, and are redirected to the leader (Alg. 2, line 1), which stores them in a set of pending requests (Alg. 2, line 5). The leader, upon receiving a new request, starts a new instance by increasing

Algorithm 2 ChainPaxos algorithm: message flow.

```
1:  upon receive <NEW_REQUEST,req> from client do:
2:      submitted ← submitted ∪ {req}
3:      SEND(c_sleader,<REDIRECT_REQUEST,req>)
4:
5:  upon receive <REDIRECT_REQUEST,req> from r do:
6:      if self = c_sleader then                ▷ Even if there is no quorum yet
7:          pending ← pending ∪ {req}
8:
9:  function STARTINSTANCE
10:     max_acpt ← max_acpt + 1
11:     SEND(self,<ACCEPT,max_acpt,self,np_leader,pending,0,max_ack>)
12:     pending ← ∅
13: upon receive <ACCEPT,n_i,ldr,n_a,val,n_acpts,m_ack> from r do:
14:     if np_leader ≤ n_a then                 ▷ Has not seen higher prepare
15:         UPDATELEADERINFO(ldr,n_a)           ▷ If a prepare was missed
16:         if ∄inst[n_i] ∨ inst[n_i].n_a < n_a then
17:             inst[n_i] ← (n_a,val,n_acpts + 1,false)
18:         else                                ▷ Repeated accept
19:             inst[n_i].n_acpts ← MAX(n_acpts + 1,inst[n_i].n_acpts)
20:         if inst[n_i].val = RemoveNode(node) then
21:             MARKFORREMOVAL(n_i,node)
22:         if ISQUORUM(n_acpts) ∧ ¬inst[n_i].decided then
23:             DECIDE(n_i)
24:         DECIDEANDGCUPTO(m_ack)
25:         FORWARD(n_i)
26:
27: upon receive <ACCEPT_ACK,n_i> from r do:
28:     DECIDEANDGCUPTO(n_i)
29:
```

the instance number and generating a new *accept* message (Alg. 2, line 9). The *accept* message contains the following information: (i) the instance number, which the leader tracks in $\texttt{max}_{\texttt{acpt}}$; (ii) the id of the leader; (iii) the prepare number, $\texttt{np}_{\texttt{leader}}$, used by the leader in its previous prepare message; (iv) the client request (i.e., operation); (v) the number of nodes which have accepted the value ($\texttt{n}_{\texttt{acepts}}$), initialized to 0; and (vi) the highest instance for which the decided value is known to have been accepted by all replicas ($\texttt{max}_{\texttt{ack}}$).

The leader is the first to handle the *accept* message of each instance, as it starts a new instance by sending the *accept* to itself (Alg. 2, line 11). Upon receiving an *accept* message for an instance (Alg. 2, line 13), a node stores the information for the instance, increasing the value of $\texttt{n}_{\texttt{acpts}}$ to indicate the node itself is accepting the value. If $\texttt{n}_{\texttt{acpts}}$ is greater than $n/2$, the message has already been accepted by a majority of nodes, and its value can be decided (Alg. 2, line 22). Otherwise, the value will be decided (and garbage-collected) when an *accept* message is received with $\texttt{m}_{\texttt{ack}}$ greater or equal to its instance number. This is performed in function `DecideAndGCUpTo` (called in Alg. 2, line 24 and defined in Alg. 3, line 35). This function traverses every (non-garbage-collected) instance up to instance $\texttt{max}_{\texttt{ack}}$, marking them as decided (if they were not yet), and garbage-collecting the information about them after their execution. This is safe since all instances up to $\texttt{max}_{\texttt{ack}}$ have been accepted by every node in the chain.

Finally, the node forwards the *accept* message (with the incremented $\texttt{n}_{\texttt{acpts}}$) to the next node in the chain. If the replica is the last node in the chain, it sends an *accept ack* message to the leader, signalling that every node in the chain has seen and accepted the instance. Upon receiving this message, the leader executes `DecideAndGCUpTo`, increasing its $\texttt{max}_{\texttt{ack}}$ which leads subsequent *accept* messages to trigger `DecideAndGCUpTo` in every node across the chain.

The nodes in the second half of the chain (starting from the $n/2^{th}$ node) can decide instances as soon as they receive the *accept* message, while the first $n/2$ nodes only decide (and execute an operation) after receiving an acknowledgement (the leader via an *accept ack* message, and the other nodes via the $\texttt{max}_{\texttt{ack}}$ value piggybacked in subsequent *accept* messages).

In a fault-free run, our protocol simply encodes the messages of Multi-Paxos in ChainPaxos messages. A ChainPaxos *accept* message sent by node `n` encodes the Multi-Paxos *accept* message and the *accept ack* messages of `n` and all nodes that precede it in the chain. It also encodes the *accept ack* messages of all nodes in the chain for all instances up to $\texttt{m}_{\texttt{ack}}$. A ChainPaxos *accept ack* message encodes the Multi-Paxos *accept ack* messages of all nodes in the chain.

3.4 Dealing With Faults and Reconfigurations

To describe how faults and membership reconfigurations are handled in ChainPaxos, we begin by describing the mechanisms used by replicas to suspect other nodes (i.e., fault detection) and then discuss the steps taken by ChainPaxos to reconfigure the system, either keeping the current leader or when the leader is suspected. The main challenge faced by ChainPaxos is that, when using a chain topology, the failure of a single node leads the chain to break, making it impossible for messages to keep flowing along the chain, resulting in a system halt.

Fault Detection: We have implemented two mechanisms for fault suspicion. To pinpoint faults in the chain, each replica expects to receive periodic keep-alive messages from the following node in the chain. If a node does not receive the keep-alive for a configurable period of time, it suspects the node, and requests the leader to remove it, triggering a *Reconfiguration not involving the leader*. In case the tail suspects the failure of the leader (which is its next node), it starts the process of taking leadership (Phase 1 of Paxos), and then starts the process of removing it. This effectively triggers a *Reconfiguration involving the leader*.

We note that, as we assume an asynchronous system, suspecting a node does not necessarily mean that it failed, but rather that there is a chance it might have, as it can just be temporarily slow [29]. However, since a single failed (or just slow) node can block progress in the whole chain, the keep-alive mechanism is important to allow quick removal of suspected nodes, minimizing their negative impact on the overall throughput of the chain. Incorrectly removed replicas can later rejoin the system.

The second mechanism is based on the continuous flow of *accept* messages. If a replica does not receive an *accept*

Algorithm 3 ChainPaxos algorithm: auxiliary functions.

```
 1: function MARKFORREMOVAL(n_i,node)
 2:     marked ← marked ∪ {node}
 3:     if node = c_nextok then              ▷ We marked the closest unmarked node
 4:         c_nextok = NEXTNODENOTMARKED(self, marked)
 5:         for n ← max_ack + 1, n_i − 1 do              ▷ Re-propagate accepts
 6:             FORWARD(n)
 7: function FORWARD(n_i)
 8:     if c_nextok = leader then
 9:         SEND(c_nextok,<ACCEPT_ACK, n_i>)
10:     else
11:         SEND(c_nextok,<ACCEPT, leader, n_i, inst[n_i].n_a,
             inst[n_i].val, inst[n_i].n_acpts, max_ack>)
12: function UPDATELEADERINFO(leader,n_p)
13:     if np_leader < n_p then
14:         amLeader ← false
15:         pending ← ∅
16:         c_sleader ← leader                          ▷ Set new leader
17:         np_leader ← n_p              ▷ Set the prepare number for the leader
18:         for req ∈ submitted do           ▷ Redirect requests to new leader
19:             SEND(c_sleader,<REDIRECT_REQUEST, req>)
20:         marked ← {}
21:         c_nextok = NEXTNODENOTMARKED(self, marked)
22: function DECIDE(n_i)
23:     inst[n_i].decided ← true
24:     if inst[n_i].val = RemoveNode(node) then
25:         marked ← marked \ {node}
26:         chain ← chain \ {node}
27:     else if inst[n_i].val = AddNode(node) then
28:         chain ← chain ∪ {node}
29:         c_nextok = NEXTNODENOTMARKED(self, marked)
30:         if c_nextok = node then           ▷ Was added right next to me
31:             STATETRANSFER(c_nextok, n_i)
32:     else
33:         SMREXECUTE(inst[n_i].val)
34:         pending ← pending \ {inst[n_i].val}
35: function DECIDEANDGCUPTO(n_i)
36:     for i ∈ inst ∧ i ≤ n_i do              ▷ sequential iteration up to n_i
37:         if ¬i.decided then
38:             DECIDE(n_i)
39:         inst ← inst \ {i}
40:     max_ack ← n_i
```

for a configurable period of time, it assumes that the leader is faulty and attempts to take leadership. If, during this process, the new leader could not establish a connection to some other node (to send them the *prepare* message), it suspects and starts the process of removing them. To make sure this mechanism operates correctly even if the system is subjected to a low load, the leader issues periodic *accept* messages for a special NoOP operation if there are no client requests.

Reconfiguration not involving the leader: We now explain how ChainPaxos reconfigures the chain by removing a suspected node that is not the leader.

When the leader is notified that node n is suspected, it starts an instance with RemoveNode(n) operation to remove node n from the chain. When the instance is decided, n is removed from the chain, updating the variables with the local configuration of the chain (chain and marked).

When a RemoveNode operation is being propagated, two actions need to be taken to guarantee correctness and progress: *i*) guarantee that all previous *accept* messages that might have been lost due to the failure of the node are forwarded to the next correct node (to reestablish the flow of those accept messages); and *ii*) guarantee that all subsequent *accept* messages are forwarded through the chain despite faulty nodes, until the RemoveNode operation is decided, removing the faulty node, and repairing the chain.

The former is implemented in MarkForRemoval, executed when processing an *accept* message for a RemoveNode operation (Alg. 2, line 21). The node to be removed is added to the set of marked nodes (Alg. 3, line 2). If the node to be removed is the next node that was not previously marked, it is possible that it failed to propagate previous messages through the chain. Thus, the node sends to the next non-marked node any *accept* messages (or *accept ack* for the leader) for instances that have not yet been garbage collected (i.e., instances from m_{ack} to $n_i - 1$ (Alg. 3, line 5). This guarantees that, when healing the chain by bypassing faulty nodes, all *accept* messages will be received by all nodes that will not be removed from the chain, somewhat falling back to the pattern of Multi-Paxos[1].

The latter guarantee is provided by the Forward function (Alg. 3, line 7). This function forwards the *accept* message for a given instance to the next node. When one or more of the following nodes are marked to be removed (because a RemoveNode operation has been received, but has not yet been decided), the function forwards the *accept* message to the next non-marked node. This guarantees that a node that is to be removed in instance n_i will not vote for instances $n > n_i$.

When a leader change occurs while a RemoveNode operation for a node r is being propagated through the chain, it is possible that the operation, while observed by a minority of replicas (that add r to their marked set), is not decided. Following the regular behaviour of Multi-Paxos, the new leader might issue a different operation for that instance. Such operations should be sent to r to ensure correctness. To do so, when a replica learns about the new leader it removes all nodes from the marked set and updates the c_{nextok} variable, ensuring that messages flow across all nodes. (Alg. 3, line 12).

Reconfiguration involving the leader: ChainPaxos supports changing the leader by having a node become the leader at a given instance for that and all following instances by executing the first phase of Paxos.

This process is initiated in function TryToBecomeLeader (Alg. 4, line 1). The node selects a prepare number higher than any prepare number already seen in any instance, and sends a *prepare* message for instance $max_{ack} + 1$ directly to all nodes. Although this prepare is for a given instance, it will make the node leader of all instances from that point onward – thus, the prepare number must be larger than any

[1] Note that this might lead to nodes receiving multiple *accept* messages for the same instance with the same prepare number from different nodes. This is addressed by considering the highest observed number of acks reported in these messages. This is safe because the forward process employed during recovery never generates cycles.

Algorithm 4 ChainPaxos algorithm: leader election.

1: **function** TRYTOBECOMELEADER
2: $\quad \mathtt{n_p} = \text{NEXTPREPARENUM}(\mathtt{np_{leader}})$
3: $\quad \text{SEND}(\forall n \in chain, <\text{PREPARE}, \mathtt{max_{ack}}+1, \mathtt{n_p}>)$
4: **upon receive** $<\text{PREPARE}, \mathtt{n_i}, \mathtt{n_p}>$ **from** r **do:**
5: $\quad$ **if** $\mathtt{np_{leader}} \leq \mathtt{n_p}$ **then** ▷ Has not seen higher prepare
6: $\quad\quad \text{UPDATELEADERINFO}(\mathtt{leader}, \mathtt{n_a})$ ▷ New accepted leader
7: $\quad\quad \mathtt{insts_{accepted}} = \text{GETACCEPTEDINSTSFROM}(\mathtt{n_i})$
8: $\quad\quad \text{SEND}(\mathtt{node}, <\text{PREPARE_OK}, \mathtt{n_i}, \mathtt{n_p}, \mathtt{insts_{accepted}}>)$
9:
10: **upon receive** $<\text{PREPARE_OK}, \mathtt{n_i}, \mathtt{n_p}, \mathtt{insts_{accepted}}>$ **from** rep **do:**
11: $\quad$ **if** $\mathtt{np_{leader}} \leq \mathtt{n_p}$ **then** ▷ Has not seen higher prepare
12: $\quad\quad \text{REGISTERPREPAREOK}(\mathtt{n_i}, \mathtt{n_p}, \mathtt{insts_{accepted}})$
13: $\quad\quad$ **if** $\text{HASPREPAREOKQUORUM}(\mathtt{n_i})$ **then** ▷ Became leader
14: $\quad\quad\quad \mathtt{amLeader} \leftarrow true$ ▷ Can now start new instances
15: $\quad\quad\quad$ **for** $(\mathtt{a_{ni}}, \mathtt{a_{na}}, \mathtt{a_{val}}) \in \text{ACCEPTEDINSTSFROM}(\mathtt{n_i}, \mathtt{n_p})$ **do**
16: $\quad\quad\quad\quad \text{SEND}(\mathtt{self}, <\mathtt{ACCEPT}, \mathtt{a_{ni}}, \mathtt{self}, \mathtt{n_p}, \mathtt{req}, 0, \mathtt{max_{ack}}>)$
17: $\quad\quad\quad\quad \mathtt{max_{acpt}} \leftarrow \mathtt{a_{ni}}$
18:

previously used by any replica. We use $\mathtt{max_{ack}}+1$, since it guarantees that previous instances have already been accepted by every node, and all messages regarding those instances can be discarded. As such, nodes only need to maintain the single highest prepare number $\mathtt{np_{leader}}$ ever received (instead of keeping a prepare for each instance). Since *prepare* messages need to have unique prepare numbers, this number includes an identifier of the node which is used to make sure that no two *prepare* messages from different nodes have the same np.

A *prepare* message for a given instance is rejected if the node has already seen a higher prepare number for any instance (either on *prepare* or *accept* messages). Otherwise, the usual Paxos logic is executed for this and all higher instances, with the corresponding *prepare ok* message being returned, which includes all previously accepted values (and corresponding prepare numbers) for the instance indicated in the *prepare* message and all following instances (Alg. 4, line 8). This is necessary as a successful prepare also makes the node the leader of all future instances. From this point until a *prepare* with an higher prepare number is received, the sender of the *prepare* message will be set as the supported leader $\mathtt{c_{sleader}}$ and all pending and future client requests will be redirected to it (Alg. 3, line 12).

Upon reception of a quorum of *prepare ok* messages (Alg. 4, line 13), the node considers itself the new leader. It then executes the regular Paxos logic, but for multiple instances: for all instances for which accepted values exist, it uses the value with the highest associated prepare number as its proposal for that instance, and forwards the corresponding *accept* message over the chain. The regular protocol execution then resumes. In Annex A, we discuss in more detail the correctness of leader election and reconfigurations.

Adding a new replica: For adding a node *n* to the chain, *n* sends a request to a replica with AddNode(n) operation as its value. The leader processes this request by starting an instance that is executed as any other instance of ChainPaxos.

When the instance is decided, the node is added to the tail of the chain updating the local chain configuration (variables chain and $\mathtt{c_{nextok}}$). Once the new node is added, it requests the current state (history of operations or snapshot) from another node at the instance in which the operation to add the node was decided. While this state transfers in the background, the new node can already participate in the following instances actively forwarding messages (although it can only locally execute and garbage collect operations after the completion of the state transfer).

4 Local Linearizable Read Operations

In this section, we discuss our proposal to execute read operations. As we mentioned previously, in Chain Replication, due to the use of an external coordination service, reading from the tail does not provide linearizable reads in the presence of network partitions, as the tail might become partitioned and not be aware that it was removed from the system. Guaranteeing linearizable reads requires contacting the external configuration service, which defeats the purpose of the low overhead achieved by only contacting the tail. Due to similar issues, most SMR protocols only support linearizable reads by executing them as normal consensus operations or, in some cases [5], by contacting a quorum of replicas (and falling back to executing the read as a normal operation when conflicts occur). We now discuss how we leverage on the chain topology and our integrated membership management to provide linearizable reads without any added communication cost.

To provide linearizable reads, it is necessary to guarantee that the result of a read reflects a state that, at the moment the read is received, is at least as recent as the most recent state for which any node has returned a result (either for a read or for a write). The base intuition of our proposal is that a node can guarantee this property by waiting for a message to loop around the entire chain, making sure that the local node is as up-to-date as any node was at the moment the message started looping around the chain.

Based on this intuition, our solution for linearizable reads works as follows. Clients issue read operations to any replica in the chain. Upon receiving the operation, the replica locally registers that the operation depends on the lowest unseen consensus instance (but no information is sent to other nodes). For instance, if the highest instance that the replica has seen so far is 6 (regardless of it being decided or not), the read operation will depend on instance 7. Upon receiving the *accept ack* message to the consensus instance for which the operation depends on, the read operation is performed locally in the local committed state and the reply is sent to the client.

This protocol implements linearizable reads by enforcing the following properties: **i)** a read *r* returns a value that is at least as recent as any value outputted by the protocol at the moment the read was received. By waiting that the following consensus instance is acknowledged and executing the read in the current local state, a replica is assured that the result of any read that was returned at any replica before the reception

of r cannot be more recent than the result that will be returned for r – this follows from the properties of ChainPaxos, which guarantee that as messages loop the chain they make the state of replicas advance, so that the following replica in the chain is in a state that is at least as recent as the previous replica. As such, if some replica has already returned a value for state s_i, by waiting that the following consensus message is acknowledged, which requires a full loop of an operation through the chain, the local replica state will be at least as recent as s_i. Due to the same reason, the result of a read will also reflect the result of any committed write operations, at the moment the read was received; **ii)**, upon a reconfiguration, a node that is partitioned from the chain will not return stale values: when a replica loses connection to the others, either it is eventually removed from the chain, preventing it from ever replying to client read operations or it eventually reconnects to the other nodes, allowing it to continue responding to read requests. In the latter case, since the replica was not removed from the chain, no progress was made while it was partitioned, thus linearizability is not lost.

Our proposal trades a potentially higher latency (compared to executing a read as a normal operation) for the possibility of processing a read locally at any node, without additional consensus instances or communication steps. This leads to lower communication and processing overhead, and allows to balance the load of read operations across all replicas, leading to better overall performance. Under low load, write operations may be less frequent, which could delay read operations. We note however, that the head of the chain issues periodic *NoOP* operations if no write is received, as to show to the other replicas that the head is still correct, hence the maximum latency of reads in scenarios with a low load will be controlled by the frequency of these *NoOP* operations. Alternatively, to ensure faster read processing, a replica processing a read which has not received the message for the next consensus instance after some configurable timeout can forward the read to the leader to be executed as a normal operation (which in turn will allow other pending reads to complete).

5 Evaluation

This section reports the experimental evaluation of ChainPaxos in a broad range of scenarios. We start by assessing the performance and scalability in CPU-bound and network-bound settings (Section 5.2), and the impact of our novel read protocol (Section 5.3), using a replicated key-value store application under the YCSB workload [7], when compared with other consensus protocols. Then, we report the results of integrating ChainPaxos with ZooKeeper [12], by replacing the Zab [14] replication protocol (Section 5.4). Finally, we study how ChainPaxos behaves in a geo-replicated setting (Section 5.5) and the impact of reconfigurations in our integrated membership when compared to using an external coordination service (Section 5.6).

We have implemented a prototype of ChainPaxos in Java, using a framework for building distributed protocols, Babel [9], which relies on the Netty [1] framework for the communications module. Similarly to other authors [8, 26], to guarantee fairness in our comparisons, the other consensus protocols were implemented using the same codebase as ChainPaxos. This guarantees that the results are not influenced by specific implementation aspects, such as the programming language, client communication patterns or differences in optimizations (such as batching). Each protocol was implemented following the description presented in their respective publications as well as available code bases for EPaxos [25] and Ring Paxos [23]. For the latter, as proposed by the authors, we limit the number of concurrent instances the leader can start as a form of flow control to mitigate the loss of multicast messages and include a mechanism for recovering from lost messages.

Each replica includes: an application (either the replicated key-value store or Zookeeper), which receives client requests, submits them for ordering, and replies to the client when the operation is executed; a proxy, serving as the intermediary between the application and the consensus protocol, also redirecting operations to the consensus leader when applicable and; the consensus solution itself, which receives operations from the proxy and notifies it once their ordering is decided.

5.1 Experimental Setup and Parameters

The experiments were conducted on the Grid5000 testbed [3], using a cluster of machines with an Intel Xeon Gold 5220 CPU with 18 cores and 96 GiB DDR4 RAM. Machines are connected through a 25 Gbps Ethernet switched network. Each replica executes in its own machine, and clients (running YCSB [7]) execute on 3 independent machines (with multiple client threads per machine). Each client thread connects to a replica for executing operations in a closed-loop.

Every protocol is executed in similar conditions, with the exception of Chain Replication that uses Zookeeper as the external management service (following [33]). For all protocols, the leader is elected at the start of the experiment and the protocols run multiple consensus instances in parallel. All results are the average of 5 independent runs, discarding the start and end periods of each experiment. In all results presented, the standard deviation between runs is always below 10%.

In addition to ChainPaxos, Chain Replication [33], and Ring Paxos [28], we report as: *EPaxos*, the execution of EPaxos [26] in a workload where all operations conflict (which is the same case of other baselines); and *EPaxos-NoDep*, the execution of EPaxos in a workload where no two operations conflict, which is equivalent to running multiple independent Paxos instance in parallel. We note that this is an unrealistic workload, as it would require all operations to be independent from each other, being presented only to provide the best (theoretical) results for a protocol following the strategy of EPaxos. *MultiPaxos* refers to the variant of Multi-Paxos [16] where acceptors forward their *accept ack*

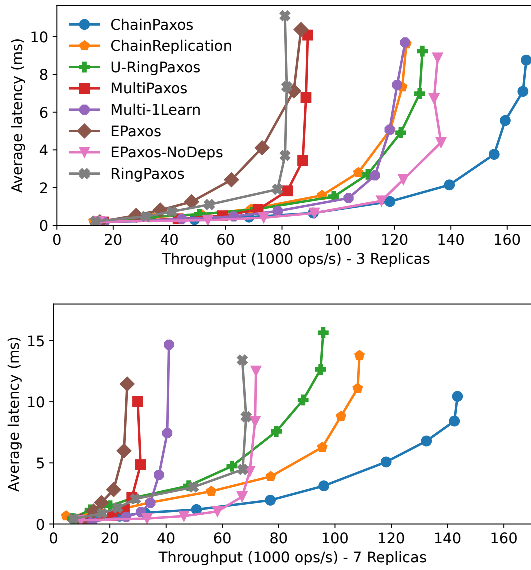


Figure 4: Performance for operations with 128 bytes (CPU bottleneck).

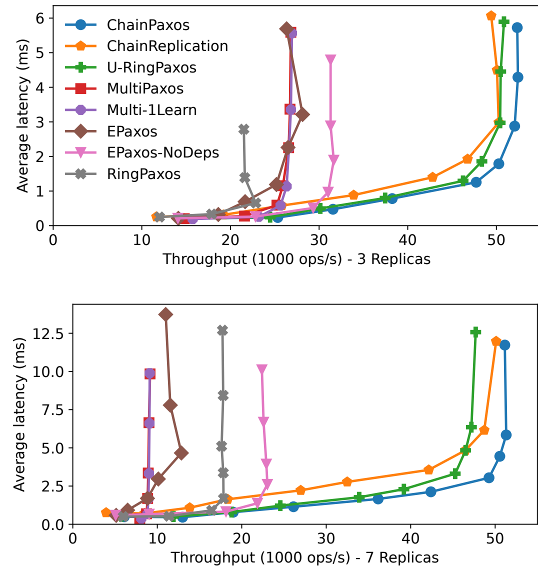


Figure 5: Performance with network bottleneck.

messages to all replicas, whereas *Multi-1Learn* represents the variant where acceptors only send the *accept ack* to the leader that, upon collecting a quorum of replies, issues a *decided* to all replicas - this protocol has a message flow equal to Raft [27] in the normal case. *U-Ring Paxos* [13] is a variation of Ring Paxos, using unicast instead of multicast, with a message flow similar to our solution and Chain Replication.

5.2 Performance in a Single Data Center

This section reports the results obtained in a single data center, running the YCSB benchmark with a replicated key-value store application. We study scenarios that attempt to saturate the CPU and the available bandwidth, by varying the size of the data stored in the key-value store.

CPU Bound. Figure 4 shows the performance of each protocol in a CPU-bound scenario. For this experiment, clients execute small (128 bytes) operations and no batching is employed (i.e., each operation is executed in an individual consensus instance). Clients connect uniformly at random to a replica, and receive a reply after the operation is executed in that replica. While this does not provide optimal latency for some solutions, it maximizes throughput by distributing the load of handling client requests as much as possible.

These results show that, by pipelining a single message per each operation through all replicas, ChainPaxos minimizes CPU usage, achieving the best performance and scalability. Chain Replication and U-RingPaxos perform worse, as they propagate some extra messages: acknowledge messages in the former, and proposals being propagated to the leader through the chain in the latter. These messages could be batched or piggybacked with a small penalty to latency. We note that the throughput of ChainPaxos with 7 replicas, which tolerates 3 faults, is higher than that of Chain Replication with both 3 and 7 replicas, which tolerate 2 and 6 faults, respectively.

For both versions of MultiPaxos, the leader (and all replicas in regular MultiPaxos) transmits and receives messages from, at least, a majority of replicas, resulting in higher CPU usage and lower performance. The impact of this effect increases with the number of replicas. For EPaxos, when all operations need to be ordered, the algorithm requires two rounds of communication, leading to an higher number of messages and lower throughput. The execution of *EPaxos-NoDeps* is similar to executing multiple parallel MultiPaxos instances, distributing the load among replicas. This leads to an higher throughput than MultiPaxos and EPaxos that, unlike ChainPaxos, also decreases with the number of replicas, as more messages need to be processed. Furthermore, we note that *EPaxos-NoDeps* is not totally ordering all operations, as other protocols do. RingPaxos is tricky to tune, as a single lost multicast message can stall the entire system. Even for our best configuration (with 150 simultaneous consensus instances), RingPaxos performance is worse than U-RingPaxos.

Overall, these results show that lowering the number of messages processed by each replica allows to achieve higher throughput with a negligible latency overhead. Furthermore, the throughput of chain-based protocols degrades very slowly when increasing the number of replicas, while the throughput of other protocols degrades quickly, as the number of messages processed by each node depends on the number of replicas in the system. This is relevant for supporting critical systems with high availability requirements.

Network Bound. Figure 5 presents the performance in a network-bound scenario. For this experiment, the bandwidth of replicas is limited to 1Gbps, with clients issuing 2048 byte operations, saturating the bandwidth of the replicas without

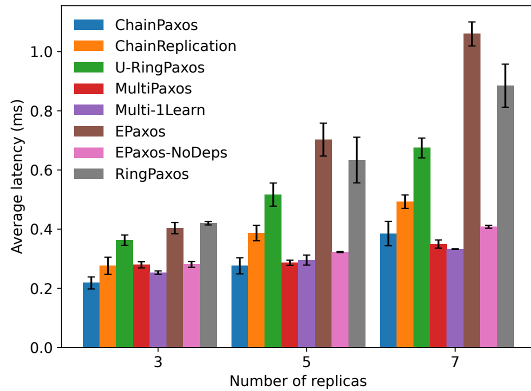


Figure 6: Latency under low load.

saturating their CPU. For saving the bandwidth consumed in redirects and maximizing the bandwidth available for the consensus protocol, all clients connect directly to the leader/head (uniformly distributed in both EPaxos variants).

Results show that ChainPaxos, ChainReplication, and U-RingPaxos, by only receiving and transmitting each operation once, achieve maximum use of available bandwidth. For these solutions, the replicas were consuming approximately 900 Mb/s of both inbound and outbound bandwidth. This allows the system to maintain its performance with an increasing number of replicas. For MultiPaxos, since the leader needs to transmit each operation to all other replicas, its bandwidth usage is disproportionately higher than that of other replicas, limiting their throughput. Furthermore, the throughput decreases with the number of replicas. EPaxos versions suffer from the same issue, but since EPaxos uses multiple leaders, it distributes the load of the leader across all nodes, leading to better scalability than MultiPaxos. *EPaxos-NoDeps* requires less communication steps, having higher throughput, but still far from ChainPaxos. For RingPaxos, the higher message size results in more frequent message losses. Even configuring the number of concurrent instances to 20 as to achieve the best results, the performance is substantially lower than that of ChainPaxos and Chain Replication.

Latency with a fixed throughput. Figure 6 shows the latency with a fixed load – clients execute 9000 operations per second, using payloads of 128 bytes. In this experiment, clients are setup to minimize latency: in RingPaxos and MultiPaxos clients connect directly to the leader; in EPaxos clients connect to all replicas uniformly; in Chain Replication and U-RingPaxos clients connect to the tail; and in ChainPaxos to the replica in the position $n/2+1$. Error bars present the standard deviation of the results.

The results show that, with 3 replicas, ChainPaxos and MultiPaxos variants exhibit the lowest latency, since they can respond to client requests after a single communication step. With increasing numbers of replicas, the latency of ChainPaxos increases, while the latency of both MultiPaxos variants remains mostly unaffected. Since both U-Ring Paxos and

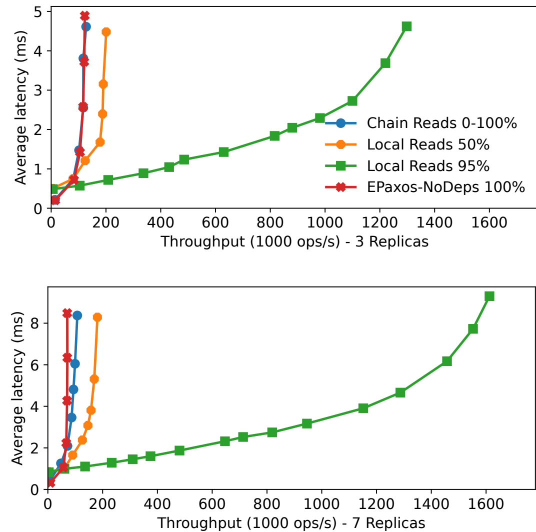


Figure 7: Performance with read operations.

Chain Replication require more communication steps until a reply is generated, their latency is always higher than ChainPaxos. We note that ChainPaxos with 5 replicas presents a similar latency to Chain Replication with 3 replicas, in which case both tolerate the failure of 2 replicas. In EPaxos, since conflicts lead to extra communication rounds, the variant where all operations conflict (*EPaxos*) naturally shows higher latency. For RingPaxos, multicast message drops (which happen even without saturation) and retransmissions lead to higher latency.

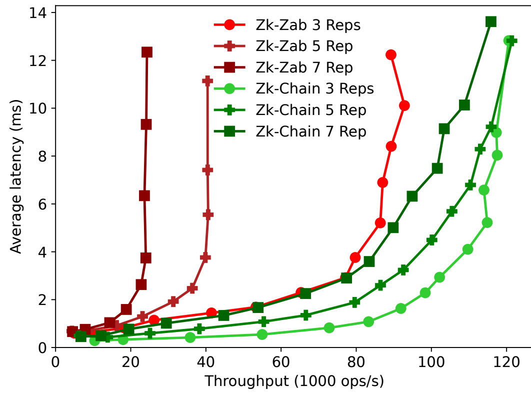
5.3 Performance of Read Operations

Figure 7 shows the impact of ChainPaxos's novel linealizable read approach in workloads with different read ratios and payloads of 128 bytes. The throughput of executing reads as normal (consensus) operations (*Chain Reads*) is constant, regardless of the ratio of read operations. For our novel approach (*Local Read*), the throughput is much higher than executing reads as normal operations, and the throughput scales both with the ratio of read operations and with the number of replicas. This is explained by the fact that as reads impose no overhead to the consensus protocol and the load is distributed evenly among replicas, more replicas can process more reads. The high throughput of ChainPaxos's local linearizable reads comes at the cost of a small additional latency under low load.

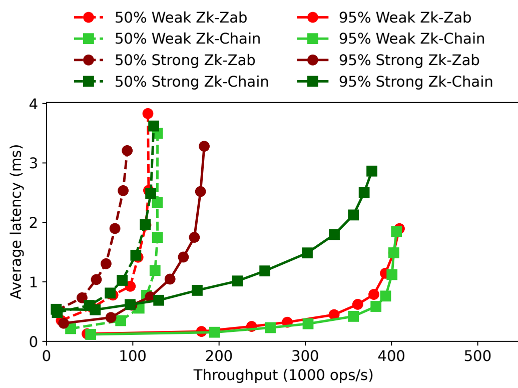
For comparison, we include the results of *EPaxos-NoDeps*, the Paxos-based protocol with best performance in the previous results. The results show that ChainPaxos achieves a significantly higher throughput than *EPaxos-NoDeps*.

5.4 Zookeeper case-study

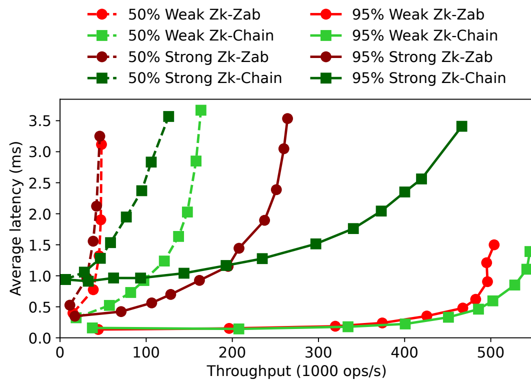
To evaluate the performance of our protocol in a more realistic scenario, we adapted ZooKeeper [12] to use ChainPaxos as its consensus protocol, instead of Zab [14]. While some features were not implemented, such as ephemeral nodes, our implementation fully supports creating, updating, and retrieving



(a) Write-only workload



(b) Mixed workload with 3 replicas



(c) Mixed workload with 7 replicas

Figure 8: Performance of Chain-based Zookeeper vs original Zookeeper.

znodes. We evaluated the performance of our implementation (*ZK-Chain*) against the original ZooKeeper using Zab (*ZK-Zab*), in a setup similar to the CPU bound scenario of Section 5.2. The results are presented in Figure 8.

For a write-only workload (Fig. 8a), the results show that ChainPaxos achieves higher throughput than the original Zookeper, with the difference increasing with the number of replicas. This is due to the lower number of messages of our protocol (Zab's message pattern is similar to *Multi-1Learn*).

Figures 8b and 8c present mixed workloads (50% and 95% of read operations), with both weak and strong reads. *Weak reads* represent the regular reads of ZooKeeper, where a replica replies with its current state, allowing for stale data to be served (e.g., with late replicas and under network partitions). *Strong reads*, in our solution, are executed using linearizable local reads. While ZooKeeper does not support linearizable reads, the authors suggest issuing a *sync* operation before a read as a close approximation of linearizability in most cases. The results show that, unlike with Zab, the strong reads with ChainPaxos scale to a throughput similar to executing weak reads. Overall, the throughput with ChainPaxos is higher than with Zab for the same setting, and the difference increases with the number of replicas.

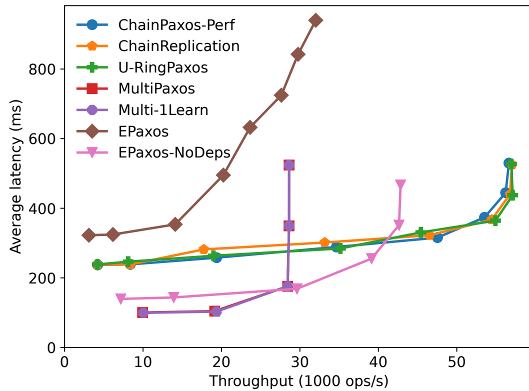
5.5 Performance in a Geo-Replicated Setting

We evaluated our protocol in a geo-replicated setting by emulating an environment with 5 sites. Using the Linux `tc` command, we limited the bandwidth to 1Gbps, and increased latency to the following values, extracted from https://cloudping.co, related to AWS EC2 data centers.

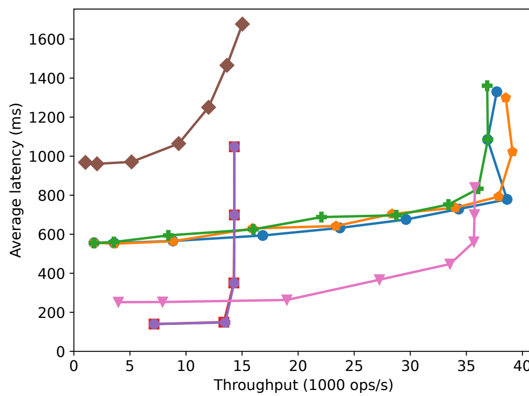
Sites	A	B	C	D	E
North Virginia (A)	-	92	127	204	186
Frankfurt (B)	88	-	210	288	279
São Paulo (C)	122	207	-	338	359
Sydney (D)	211	292	325	-	161
Seoul (E)	188	287	309	156	-

In these experiments, we did not use Ring Paxos, since IP multicast is typically unavailable across data centers. The replica in site A is always the leader/head. Experiments with 3 replicas use sites A, B, and C. Clients connect to the replica that leads to the best performance: the leader for MultiPaxos; evenly distributed for EPaxos, and; the tail for chain-based solutions.

Figure 9 presents the throughput and latency when using all available bandwidth. As within a single data center, ChainPaxos, Chain Replication, and U-Ring Paxos are able to make optimal use of available bandwidth, providing higher throughput than other protocols that order all operations. The EPaxos variant without inter-operation dependencies is able to maintain its throughput with a varying number of replicas, since the cost of transmitting each operation to all nodes is divided among the multiple leaders. However, we remind the reader that this configuration of EPaxos provides weaker guarantees than the other alternatives. As for latency, the latency of the chain-based solutions degrades as the number of replicas increases, as it takes longer for the messages to traverse the chain. For a high number of replicas, MultiPaxos variants provide lower latency, as communication between the leader and other replicas proceeds in parallel, although with, at most, half the throughput of ChainPaxos.



(a) 3 Replicas



(b) 5 Replicas

Figure 9: Performance in geo-replicated setting.

5.6 Impact of Reconfiguration

In our final experiment we evaluate the impact of reconfiguration, comparing ChainPaxos that uses its own integrated management mechanism and Chain Replication that uses an external management scheme based on Zookeeper (executing on dedicated machines). We conduct these experiments in the geo-replicated scenario with independent Zookeeper instances at sites A, B, and D. This distribution minimizes latency for replicas without a local Zookeeper replica. We used 1s timeouts to suspect the failure of another node (both in ChainPaxos and in ZooKeeper).

Experiments run for 90 seconds. Every 10 seconds the following reconfiguration events occur (denoted by vertical red lines for replica failures and green lines for replica additions): 10s) the tail node fails; 30s) the middle node fails; 50s) the head/leader fails; 70s) the head and middle replicas fail simultaneously. Replicas are added at 20s, 40s, 60s, 80s, in sites where a replica had previously failed. Clients issue operations to a random active replica to distribute the load.

Figure 10 shows the throughput observed during the experiments. Despite Chain Replication using additional resources (3 extra machines executing Zookeeper), it takes more time to perform a reconfiguration than ChainPaxos, particularly when

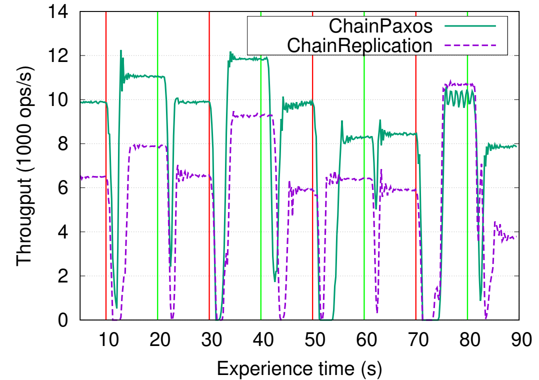


Figure 10: Reconfiguration

adding replicas to the set (green vertical lines). This happens because any reconfiguration has to be coordinated through Zookeeper. However, ChainPaxos takes longer to perform the reconfiguration when the leader fails because it resorts to the regular communication pattern of Paxos, whereas Chain Replication only fetches the new leader from Zookeeper. When the leader and middle nodes fail simultaneously (70s), both solutions take the same time to perform reconfiguration because ChainPaxos can handle both reconfigurations in parallel (albeit using two operations), whereas Chain Replication performs two sequential reconfiguration steps with Zookeeper. In general, ChainPaxos handles reconfiguration faster than Chain Replication without the cost of requiring additional machines to run the external management system, while avoiding the vulnerabilities to network partitions that can compromise the safety of the system [2].

6 Final Remarks

This paper presented ChainPaxos, a distributed consensus algorithm for high throughput replication of deterministic services. ChainPaxos exploits a pipeline communication pattern which allows to reduce the number of messages that each replica needs to send and process, while leveraging the foundations of Paxos to allow leader exchanges. Unlike previous solutions that exploit this communication pattern, ChainPaxos relies on a novel approach to execute linearizable read operations without incurring in any additional communication cost. Finally, ChainPaxos integrates membership management within the protocol. The fully specified algorithm fills an empty space in the literature and, unlike many recent proposal, decouples the fault-tolerance of ChainPaxos from that of the external coordination service. Our extensive evaluation shows that ChainPaxos achieves higher throughput and better scalability when compared to state-of-the-art solutions. Furthermore, our approach for executing linearizable reads has a huge impact on scalability. The results illustrate the benefits of our solution in the context of a key value store and the Zookeeper coordination services (where ChainPaxos leads to better performance than Zab).

References

[1] Netty framework. https://netty.io/.

[2] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 351–368, 2020.

[3] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[4] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation*, pages 141–154, 2011.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[6] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[8] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, and Joao Leitao. Babel: A framework for developing performant and dependable distributed protocols. *arXiv preprint arXiv:2205.02106*, 2022.

[10] Rachid Guerraoui, Dejan Kostic, Ron R Levy, and Vivien Quema. A high throughput atomic storage algorithm. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 19–19. IEEE, 2007.

[11] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[12] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[13] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: High-throughput atomic broadcast. *The Computer Journal*, 60(6):866–882, 2017.

[14] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

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

[16] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[17] Leslie Lamport. Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

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

[19] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. Technical report, 2009.

[20] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.

[21] Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*, pages 307–314. IEEE, 2004.

[22] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 369–384. USENIX Association, 2008.

[23] Parisa Jalili Marandi. U-Ring paxos code. https://github.com/sambenz/URingPaxos. (Accessed Oct 2019.).

[24] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[25] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Epaxos code. https://github.com/efficient/epaxos. (Accessed Mar-2019.).

[26] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[27] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[28] Parisa Jalili Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 527–536, June 2010.

[29] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–14, 2015.

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

[31] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, 2009.

[32] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42–1, 2015.

[33] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[34] Avishai Wool. Quorum systems in replicated databases: Science or fiction? *IEEE Data Eng. Bull.*, 21(4):3–11, 1998.

A Correctness

In this section, we present the correctness argument for ChainPaxos, by showing that the execution of an instance of ChainPaxos is equivalent to an execution in Multi-Paxos. We consider the following cases: (i) an instance where all nodes agree on the leader; (ii) an instance that elects a new leader; (iii) instances following a new leader election; and (iv) instances where a replica is removed or added.

A.1 All nodes agree on the leader

This case is the one described in Section 3.3. In Multi-Paxos, an equivalent run would consist in: (i) the leader sending the *accept* message to all replicas; (ii) each node replying with an *accept ack* message to the leader that learns the decided value; (iii) the leader informing all other nodes of the learned value.

In ChainPaxos, the *accept* message is also sent to all nodes, but instead of being sent directly it is sent indirectly as the *accept* message is forwarded across all nodes of the chain.

The *accept* message also encodes the *accept ack* messages of the nodes through which it passes. When the leader receives the *accept ack* message from the tail of the chain, this message is equivalent to having all nodes sending the *accept ack* message to the leader in Multi-Paxos.

Piggybacked in the following *accept* messages (even for the NOOP request), the leader sends to all other nodes information that it has received the *accept ack* from all nodes of the chain ($\mathtt{m_{ack}}$ in *accept* message), which is equivalent to forwarding the learned value to all replicas in Multi-Paxos.

As messages are used to update the local state using the same logic in both ChainPaxos and Multi-Paxos, the execution of an instance in ChainPaxos is equivalent to the execution of an instance in Multi-Paxos.

A.2 Instance with a leader election (two-phases)

We start by considering the effects of an instance executed with the two phases of the Paxos protocol. In this case, the first phase of both protocols is identical – ChainPaxos uses the same message flow as Paxos, since *prepare* and *prepare ok* messages are sent directly between the node starting the prepare and all other replicas. The logic to process the *prepare* message in ChainPaxos differs from Multi-Paxos in the following aspect: ChainPaxos only acknowledges *prepare* messages for a given instance with prepare numbers higher than any prepare number used by any other leader in any other instance. The only implication of this is that the replica executing the prepare, might time-out several times. This is not different from Multi-Paxos, where the replica will complete the prepare in the current instance, and then, if any other following instance had already been executed by a different leader (potentially using the special prepare number zero), it would have to repeat the prepare phase for each such instance. This is true because in ChainPaxos the prepare number is continually used by a leader in all subsequent instances after the one in which it complete the prepare phase successfully.

The second phase of ChainPaxos consists in sending the accept message to all nodes. As explained in the previous case, ChainPaxos flow of messages is equivalent to the Multi-Paxos flow of messages. As the logic employed to process the messages is the same as in Multi-Paxos, also in this case, the execution is equivalent to that of an instance of Multi-Paxos.

A.3 Instances following a new leader election

When a new leader is elected, by running the first phase of Multi-Paxos for instance $\mathtt{n_i}$, it becomes the leader of all instances n_j such that $\mathtt{n_j} \geq \mathtt{n_i}$, for which it will subsequently issue *accept* messages through the chain (in order). To prove the correctness of ChainPaxos, we show that this execution is equivalent to running the two phases of the Paxos protocol for all instances $\mathtt{n_j}$, such that $\mathtt{n_j} \geq \mathtt{n_i}$.

The processing of the *prepare* message ignores the prepare if the node has seen an higher prepare number employed by another leader (for any instance). If the prepare is not ignored, it runs the Paxos logic for all instances $\mathtt{n_j}$, such that $\mathtt{n_j} \geq \mathtt{n_i}$, with the *prepare ok* message including the information relative to all these instances. This information includes any accepted value for each of those instances and the prepare number associated with that accepted value. The processing of the *prepare ok* message also executes the Paxos logic for all instance $\mathtt{n_j} \geq \mathtt{n_i}$, such that the new leader will send new accept messages for all those instances (in order) where, if there was already a value accepted by any replica for that instance, the new leader will propose that value and, similar to Multi-Paxos, if more than one value had been accepted by different replicas, the leader proposes the value with the highest associated prepare number. This ensures that for every instance $\mathtt{n_j} \geq \mathtt{n_i}$, if some value had already been accepted by a majority of replicas, then the new leader will propose that value with his current propose value.

Thus, executing the first phase of the Multi-Paxos in ChainPaxos for instance n_i and issuing the accept for all instances $\mathtt{n_j}$, such that $\mathtt{n_j} \geq \mathtt{n_i}$ as described above is equivalent to executing the two phases of the Paxos protocol for all instances $\mathtt{n_j}$, such that $\mathtt{n_j} \geq \mathtt{n_i}$, thus ensuring the correctness of ChainPaxos.

A.4 Leader Conflicts

As in Paxos, it is possible that two nodes receive a quorum of *prepare ok* messages for different prepare numbers concurrently. In this case, *accept* messages from the node with the lowest prepare number will be dropped during their propagation and will never be accepted by a majority of replicas.

The processing of *accept* messages also sets the leader and prepare number being used by the leader in all replicas. This is needed because a minority of replicas might have missed the *prepare* message.

If a previous leader is incorrectly suspected of being faulty, it might send *accept* messages for instances $\mathtt{n_i} \geq \mathtt{n}$ while a new node becomes leader in instance n. In this case, just like in regular Multi-Paxos, there are two possible scenarios for each instance $\mathtt{n_i}$: **i)** the *accept* message of the previous leader reached a majority of nodes (i.e., has been decided) before the *prepare* message of the new leader, in which case at least one *prepareOk* message received by the new leader will contain the value proposed by that *accept* message, and the new leader will simply propose that same value; **ii)** the *accept* has been "cut-off" by the *prepare* message of the new leader (i.e., a node in the chain rejected the *accept* after receiving the *prepare* with an higher sequence number). In this case, the new leader may or may not receive the value proposed by that *accept* in a *prepareOk* message. Regardless, since the instance had not been decided, both alternatives are correct.

A.5 Removal and addition of a replica

We now discuss the correctness of ChainPaxos when reconfiguring the system to either remove or add a replica, which has been presented in Section 3.4. Such reconfigurations resort to special SMR operations which have to be ordered by ChainPaxos and executed by replicas. A challenge that is present in such reconfigurations is that the number of replicas that constitute a majority of the system might change due to the execution of these operations.

In ChainPaxos, we define a minimum quorum size, and then vary the size of the quorum required to decide operations to always be a majority quorum. For instance, assuming a configuration with 5 initial replicas (where a majority quorum is 3), and a minimum quorum size of 3, adding 2 replicas (to a total of 7), would increase the majority quorum to 4. If 2 replicas are then removed, the majority quorum will be back to 3. However, due to the minimum quorum size, removal of further replicas would not decrease the quorum size below 3, which also ensures that the number of replicas in the system can never be below 3. The minimum quorum size is independent from the initial configuration, serving as a threshold below which we do not wish the system to continue functioning.

Note that in ChainPaxos no replica considers an instance as decided before knowing the operations that have been ordered in all previous instances, hence a replica will never use the incorrect quorum size to decide (and execute) an operation.

We start with the addition of a replica. To ensure correctness we need to show that the replica that is added in an instance n_i will be considered towards forming the majority of *accept ack* messages necessary to decide any instance $n_j > n_i$. This derives from the agreement property of ChainPaxos, which ensures that all (correct) replicas will agree on the instance in which the new replica is added to the system. Since replicas only consider an operation as decided after learning the decided values for all previous instances, no replica will ignore the participation of the new replica when deciding the value of any instance n_j, since the replica has already been added on instance n_i (i.e., the instance where the new replica is added), affecting the size of the quorums.

Regarding removal of nodes, to ensure correctness we need to show that the removal of a replica in instance n_i, makes it impossible for that replica to affect the decided value in any instance $n_j > n_i$ (i.e., the *accept ack* messages of that replica are never considered to achieve a majority in such instances). As discussed previously, the *accept* message that is forwarded along the chain, in round n_i, to remove a replica r is forwarded by the node directly before r to both r and its successor replica in the chain. Any node that receives such a message, adds r to its `marked` set. This makes that any *accept* message for a subsequent instance is never sent to r, hence r is not able to increment the the counter for *accept ack* within those messages.

This happens, even if nodes have not yet locally decided the outcome of instance n_i. This could be problematic if a new leader is meanwhile elected before the outcome of this instance is locally decided (and executed) by every node, since that node could be continuously skipped despite the fact that he was never removed from the system. However, if a leader change happens, the contents of the `marked` set of replicas are removed. This is performed either when the node replies to the *prepare* of the new leader, or when it receives an *accept* message from the new leader (which can be identified locally since the accept message will carry a prepare number higher than the last prepare number observed by that replica). This ensures that r receives subsequent *accept* messages for the re-executions of instances $n_j > n_i$ until the leader proposes the `removeNode(`r`)` in some instance (assuming r remains suspected).

Finally, in the case of concurrent addition or removal of replicas to the system, we note that ChainPaxos executes each addition or removal as an independent operation. We note that the fault detection mechanism may lead replicas to incorrectly suspect other replicas (e.g. due to temporary network failures). In this case, if a replica is incorrectly removed, it can ask to rejoin the system (note that ChainPaxos tries to propagate the remove operation to the node to be removed, as it is still part of the system in that instance). Finally, we note that it is easy to minimize scenarios where replicas ask for the removal of correct replicas by having the leader avoid to either remove replicas that he perceives as active, or removing replicas that were suspected by replicas being currently removed.

B Artifact Appendix

Abstract

The artifact includes the implementation of ChainPaxos, along with the other consensus algorithms that are studied in the evaluation, with intructions on how to launch and test them. These implementations include a simple replicated key-value store that was used to benchmark the algorithms. Additionally, we include our implementation of ChainPaxos in ZooKeeper, which replaces Zab.

For reproducibility, our artifact includes the client-side code that was used in the paper to measure the various performance metrics of the algorithms, along with instructions on how to run it and how to parse and interpret its results.

Scope

The artifact allows executing our consensus algorithm, ChainPaxos, which supports different read execution techniques. Our prototype fully supports the operations related with the integrated membership. The artifact includes everything that was used in the paper: source-code of all solutions; client source-code; scripts to execute the experiments; scripts to generate the plots from the experiment logs and; instructions on how to reproduce all plots.

Contents

The artifact is divided in four parts, which are distributed across four repositories:

ChainPaxos: This repository contains the code for our full implementation of ChainPaxos. Additionally, it includes the key-value store application and the different consensus algorithms that we used to compare against ChainPaxos (in the Figures 4 to 7, 9 and 10). The repository also includes information on how to compile and deploy ChainPaxos.

The source-code in the repository is divided in multiple Java packages, with the following structure: the package *chainpaxos* contains our implementation of ChainPaxos; the code for the key-value store application is on package *app*; packages *frontend* and *common* contain some generic interfaces and classes to uniformize the interaction between the application and all consensus algorithms and; all other packages are named after the consensus algorithms that we used to compare against our solution in the paper.

ZooKeeper with ChainPaxos: This part contains our modified version of ZooKeeper that replaces Zab by ChainPaxos, that was used for the results of Figure 8. The majority of code modifications are contained in the package *chain* that is on the *zookeeper-server* module.

Client-side benchmark: This part contains all the client code that was used in the entire experimental evaluation, both to benchmark the various algorithms using the key-value store, and to benchmark the original ZooKeeper against our version with ChainPaxos. The source-code itself consists of YCSB drivers, one for the key-value store and another for ZooKeeper. The repository also includes the scripts used to perform our experiments, and instructions on how to use them in order to reproduce the results in the paper.

Results parser: Finally, our artifact includes a series of Python scripts that were used to parse the results of each experiment and generate the plots presented in this paper. The *client-side benchmark* repository contains instructions on how to use these scripts.

Hosting

The artifacts can be found in the following locations:

- ChainPaxos
  - `https://github.com/pfouto/chain`
  - master branch
  - commit 72cebf2
- ZooKeeper with ChainPaxos
  - `https://github.com/pfouto/chain-zoo`
  - master branch
  - commit 65a9690
- Client-side benchmark
  - `https://github.com/pfouto/chain-client`
  - master branch
  - commit ed28200
- Results parser
  - `https://github.com/pfouto/chain-results`
  - master branch
  - commit e716e4a

Requirements

While the artifact does not have special hardware requirements, all experiments were conducted in the Grid5000 testbed, using the Gros cluster. The client-side benchmark repository includes instructions on how to reproduce the experiments on this cluster. Furthermore, the same repository provides instructions on how to deploy and run the experiments in any other cluster platform (e.g. on a cloud infrastructure), which requires some additional setup, but should still allow to reproduce all the results in the paper.