Cheap and Available State Machine Replication

Rong Shi
The Ohio State University

Yang Wang
The Ohio State University

Abstract

This paper presents that, by combining on-demand instantiation and lazy recovery, we can reduce the cost of asynchronous state machine replication protocols, such as Paxos and UpRight, while maintaining their high availability. To reduce cost, we incorporate on-demand instantiation, which activates a subset of replicas first and activates backup ones when active ones fail. To solve its key limitation—the system can be halted for long when activating a backup replica, we apply lazy recovery, allowing the system to proceed while recovering backup nodes in the background. The key contribution of this paper is to identify that, when agreement nodes and execution nodes are logically separated, they each present a unique property that enables lazy recovery. We have applied this idea to Paxos and built ThriftyPaxos, which, as shown in the evaluation, can achieve higher throughput and similar availability comparing to standard Paxos, despite the fact that ThriftyPaxos activates fewer replicas.

1 Introduction

This paper presents that, by combining on-demand instantiation [35, 56] and lazy recovery [29, 30], we can reduce the cost of asynchronous state machine replication (SMR) protocols [49], such as Paxos [31, 32, 45] and UpRight [16], while maintaining their high availability.

Replication is widely used in today's storage systems to protect data against failures. In general, stronger replication protocols that can tolerate more kinds of errors usually need more replicas. For example, primary backup protocols [10, 11, 21, 54] can tolerate f machine crashes with $f+1$ replicas, which is the minimal one can expect. Paxos [31, 32, 45] needs $2f+1$ replicas to tolerate f machine crashes and asynchronous events (e.g inaccurate timeout caused by network partitions or slow machines). To further tolerate arbitrary failures, Byzantine Fault Tolerance (BFT) protocols [1, 6, 13, 16, 18, 28, 36] need at least $3f+1$ replicas.

More replicas incur a higher cost: the system needs more storage space to store data, more bandwidth to transfer data, and more processors to process data. Whether to pay such additional cost for stronger guarantees becomes a hard question for developers. Indeed, while a number of systems are using Paxos to replicate their data [3–5, 9, 12, 17], many others are still using primary backup or similar protocols, willing to take the risk of occasional data inconsistency [21, 24, 44, 50].

Existing attempts to reduce replication cost are only effective for certain applications or protocols. For example, separating agreement from execution [57] can reduce the number of execution replicas in BFT protocols to $2f+1$, but it is not effective for applications whose agreement is the bottleneck or those that are using Paxos. On-demand instantiation [35, 56] activates the minimal number of replicas first, and activates backup ones when the active ones fail. However, before a backup replica can take its responsibility, it must transfer the current state from an active replica, and the system is unavailable during state transfer: for applications with a large state, the system can be halted for long. Gnothi [55] separates data from metadata, performs partial replication for data, and performs full replication for metadata: it works effectively for applications whose data is much larger than metadata, but may not work efficiently for others.

This paper, instead, presents a general approach to reduce the replication cost of asynchronous SMR protocols, while maintaining their availability properties. To reduce replication cost, our approach incorporates the idea of on-demand instantiation, which activates a subset of replicas first and activates backup ones when active ones fail. To address its key limitation—the system may be unavailable for a long time when a backup replica is rebuilding its state, our approach incorporates lazy recovery [29, 30], which rebuilds a backup replica's state in the background without halting the system.

Neither on-demand instantiation nor lazy recovery is novel. The key contribution of this work is to identify that, for SMR protocols, *lazy recovery is possible only when agreement and execution are separated.*

SMR protocols first run an agreement protocol across replicas to decide the next request to execute and then execute the request on each replica. Therefore, a replica can be logically separated into an agreement node, which runs the agreement protocol, and an execution node, which runs the application's logic to execute the request. When separated, they each present a unique property that enables lazy recovery:

Instant activation for agreement. In principle, an agreement protocol needs to answer the question "what is the next request to execute". This question has the "memoryless" property that an agreement node does not need to know prior requests to decide the next one. This suggests that when an active agreement node fails, a blank agreement node can join the protocol instantly.

Separating critical and flexible tasks for execution. An execution node must execute requests in order, because execution of later requests may depend on information in earlier requests. Its opportunity for lazy recovery comes from a different property: critical tasks that must be performed for availability (e.g. executing a request and replying to the client) sometimes require fewer replicas than flexible tasks that can be delayed (e.g. garbage collection). For example, in Paxos, a client needs only one reply from any replica to proceed, but a garbage collection requires $f+1$ replicas to take a snapshot. While existing protocols try to ensure that, despite failures, the system has enough replicas to execute even flexible tasks, this may not be necessary: activating an appropriate number of replicas to ensure the availability of critical tasks and relying on lazy recovery for flexible tasks may achieve both low cost and high availability.

These properties enable lazy recovery for both agreement and execution, but in different ways: when an active agreement node fails, a blank backup agreement node can join the protocol instantly; when an active execution node fails, the system can proceed with remaining active execution nodes (they execute critical tasks but delay flexible tasks). In both cases, the system rebuilds the state of a backup node in the background.

This paper formally studies the number of active nodes required for availability. Here we highlight some results:

- For Paxos, we need $f+1$ active agreement nodes and active execution nodes.
- For BFT, when setting distinct bounds for omission failures (u) and commission failures (r) [16] instead of setting a unified f for both, we need $u+r+1$ active execution nodes. This is smaller than the previous $2f+1$ lower bound, in a practical setting when $u > r$.

In order for our approach to work properly, there is one additional challenge we need to address: although recovery of backup nodes can be delayed, recovery still has to be completed in a timely manner. Otherwise, long recovery can hurt the durability of the system. Furthermore, delaying flexible tasks like garbage collection for too long can eventually block the system. Recovery is further complicated by the fact that it is performed in parallel with executing new requests and they often compete for resource. To address these challenges, we introduce an *adaptive recovery* mechanism, which allows a user to specify a deadline for recovery: our mechanism makes best effort to meet the deadline while using the remaining resource to process new requests. To achieve this, it continuously monitors the progress of recovery and adaptively adjusts resource allocation.

We have applied this idea to Paxos, a popular replication protocol in today's datacenters, to build Thrifty-Paxos, which can achieve the same guarantee as Paxos with f fewer replicas. Our evaluation shows that Thrifty-Paxos can achieve higher throughput and similar availability comparing to Paxos, despite the fact that Thrifty-Paxos activates fewer replicas.

2 Background

State machine replication (SMR) models an application as a deterministic state machine. For fault tolerance, SMR deploys multiple replicas of an application's state machine on different machines. To ensure all replicas are identical, SMR protocols run an agreement or consensus protocol across replicas to decide the next request to execute. These protocols can guarantee that, despite failures, all correct replicas will reach the same decision.

To tolerate network failures, replicas need to log requests during agreement, so that if a request is lost, the corresponding replica can retrieve it from the log and retransmit it. To ensure logs do not grow arbitrarily, SMR protocols periodically require application's state machines to take snapshots of their states, promising that they will never need earlier requests. The system can then garbage collect log entries before the snapshot.

Previous works exploit features of SMR protocols to reduce their cost.

On-demand instantiation. Most SMR protocols (e.g. Paxos, PBFT, UpRight, etc) are designed for an *asynchronous* environment where message delivery can be delayed arbitrarily and clocks of machines can drift arbitrarily because of asynchronous events such as network partitions or machine overloading. SMR protocols are designed to be safe (all correct replicas process the same sequence of requests) despite failures and to be live when message delivery is timely and clocks are reasonably synchronized.

While a replicated system needs a minimal number of $f+1$ replicas to tolerate f failures, an asynchronous system needs more replicas, because in an asynchronous environment, it is impossible to accurately know whether a replica has failed or not. In this case, if the system has only $f+1$ replicas, and if one of them is not responding, it is impossible for the system to decide how to proceed: it is inappropriate to proceed with remaining ones because the unresponsive replica may just be temporarily slow and in this case, the request has not been executed by sufficient number of replicas; it is also inappropriate to wait for the unresponsive replica, because the replica may have actually failed and waiting may take for ever.

To solve this problem, asynchronous SMR protocols incorporate more replicas and only expect a subset of them to respond. Such design motivates the idea of on-demand instantiation [35, 56]: since the system needs only a subset of replicas to respond, we can activate the subset first. If all of them respond in time, the system can make progress; if some of them become unresponsive, we can activate the backup ones. Since machine failures and asynchronous events are rare, this approach can reduce the replication cost in most of the time.

On-demand instantiation can achieve the same safety and liveness properties as the original approach, because asynchronous SMR protocols are designed for an environment where some of the replicas can be arbitrarily slow: in such an environment, the on-demand instantiation approach, which disables a subset of nodes, is fundamentally indistinguishable from the original approach when some nodes are slow. However, as mentioned in Section 1, previous works that adopt this idea suffer from the availability problem that a backup replica may take a long time to recover before it can function.

Separating agreement from execution. In SMR protocols, a replica can be logically separated into an agreement node, which participates in agreement, and an execution node, which runs the application's state machine.

Paxos made such separation when describing its protocol (agreement node and execution node are called acceptor and learner, respectively, in Paxos). Yin et.al. observe that for BFT protocols, the number of execution nodes can be reduced to lower system cost [57]. UpRight further refines this observation [16]. However, as mentioned in Section 1, this approach is not much helpful to light applications whose agreement is the bottleneck; it is also not effective to Paxos-like protocols.

3 Combine on-demand instantiation and lazy recovery

Our approach incorporates on-demand instantiation to reduce replication cost, and addresses its availability problem by lazy recovery: when an active replica fails, the system keeps processing requests while recovering a backup replica in the background. Such combination can achieve both low cost and high availability.

To incorporate lazy recovery, however, we must ensure that the system is able to function correctly even when part of the system is in recovery and thus only has partial state—this is the major challenge of this work. The key contribution of this paper is to identify that lazy recovery is possible only when agreement node and execution nodes are logically separated. When separated, they each present a unique property that enables lazy recovery.

3.1 Instant activation for agreement node

An agreement protocol needs to decide the next request to execute, and this task has the memoryless property that an agreement node does not need to know prior requests to decide the next one. Such memoryless property allows a blank backup agreement node to join the protocol instantly when an active one becomes unresponsive.

Number of active nodes for agreement. Suppose an SMR protocol needs a maximum number of N^A_{max} agreement nodes, in which f of them can fail. Also suppose that the SMR protocol needs N^A_{normal} agreement nodes to participate in agreement in the failure-free case. In most SMR protocols, $N^A_{normal} = N^A_{max} - f$, because there is no guarantee that more nodes can respond. However, some protocols, such as Fast Paxos [33] and Zyzzyva [28], introduce a fast path, which requires more replicas to respond, to decide the next request with less latency. When the fast path is not possible, these protocols resolve back to traditional approaches. For these protocols, N^A_{normal} could be larger than $N^A_{max} - f$.

Because a backup node can join agreement instantly, the system should activate only N^A_{normal} agreement nodes. The safety and liveness of this approach is the same as the original approach, as discussed in Section 2.

Availability. As long as the N^A_{normal} agreement nodes are correct and can communicate with each other, our system can process requests correctly. When an agreement node becomes unresponsive, our system activates a backup node. To detect failures quickly, we use aggressive techniques (Section 6), because asynchronous replication does not rely on the accuracy of failure detection for correctness. The activation only takes a few messages. Therefore, the system will not halt for long when an agreement node becomes unresponsive.

To avoid frequent conflict (different agreement nodes propose different requests), many agreement protocols elect one node as the leader to propose the next request. When the leader fails, the system may halt for a while to elect a new leader and rebuild its state. Both the original protocols and our approach have to pay such cost.

3.2 Separating critical and flexible tasks for execution node

The key observation that enables lazy recovery for execution nodes is that the number of replicas required to execute critical tasks (e.g. executing a request) is sometimes fewer than that required to execute flexible tasks (e.g. garbage collection). On the one hand, the system should activate sufficient number of nodes so that, despite failures, the system can always process critical tasks; on the other hand, it does not need to be so conservative for flexible tasks because they can be delayed.

Number of active nodes for execution. Suppose an SMR protocol needs $N^E_{critical}$ execution nodes to perform critical tasks and needs $N^E_{flexible}$ nodes to perform flexible tasks. By following the previous idea, the system should activate $max(N^E_{critical}+f, N^E_{flexible})$ execution nodes. Once again, the safety and liveness of this approach is the same as the original approach, as discussed in Section 2.

Availability. When all active execution nodes are correct and can communicate with each other, they can perform all tasks. When no more than f active nodes are unresponsive, the system still has enough replicas to perform critical tasks. Therefore, the system can rebuild backup execution nodes in the background without halting the system. Of course, the system may not be able to perform flexible tasks until backup replicas are rebuilt.

3.3 Case studies

Table 1 shows the effectiveness of our approach when applied to popular protocols.

Paxos. The standard Paxos protocol needs a maximum of $2f+1$ replicas to tolerate asynchronous events and f crash failures. Its agreement protocol sends requests to all replicas and requires $f+1$ of them to respond; its execution requires only one execution node to reply to the client (critical task) while requiring $f+1$ execution nodes to perform a snapshot for garbage collection (flexible task). By using the previous calculations, our approach needs to activate $f+1$ agreement nodes and execution nodes .

Fast Paxos and Speculative Paxos [48] introduce a fast path, which requires more than $f+1$ agreement nodes to participate[1], to commit requests with fewer rounds of message exchanges. For these protocols, our approach needs to activate more than $f+1$ agreement nodes.

Cheap Paxos [35] applies on-demand instantiation to Paxos. It requires the same number of replicas as our approach, but since it does not incorporate lazy recovery, it suffers from the availability problem.

BFT. Practical Byzantine Fault Tolerance (PBFT) [13] needs a maximum of $3f+1$ replicas to tolerate f arbitrary failures. Its agreement protocol sends requests to all replicas and requires $2f+1$ of them to respond; its execution requires $f+1$ execution nodes to reply to the client (critical task) while requiring also $f+1$ execution nodes to perform a snapshot for garbage collection (flexible task). By using the previous calculation, our approach needs to activate $2f+1$ agreement nodes and execution nodes.

Yin et. al. observe that when agreement and execution are separated, a BFT protocol needs only $2f+1$ execution nodes. Our approach can reduce its agreement cost, but cannot reduce its execution cost. Zyzzyva introduces a fast path to commit requests with less latency, but since it needs all $3f+1$ agreement nodes to respond for the fast path, our approach cannot reduce its cost.

ZZ [56] applies on-demand instantiation to BFT protocols. It requires only $f+1$ execution replicas, which is even fewer than that of our approach, but it also suffers from the availability problem. That says, being too aggressive in cutting replicas may have a cost in availability. Our approach, instead, chooses a middle ground.

UpRight. UpRight makes a distinction between failures that can cause the system to become unavailable (its number is represented by u) and failures that can cause the system to become incorrect (its number is represented by r). Its conclusion is that we need a maximum of $u+r+max(u,r)+1$ agreement nodes and $r+max(u,r)+1$ of them should respond in the agreement protocol[2]; we need a maximum of $u+max(u,r)+1$ execution nodes and $r+1$ of them should reply to the clients (critical task) while $max(u,r)+1$ of them should perform snapshots for garbage collection (flexible task). Both standard Paxos ($u=f, r=0$) and BFT ($u=r=f$) can be viewed as an instance of UpRight.

By using previous calculations ($f=u$), our approach needs to activate $r+max(u,r)+1$ agreement nodes and $u+r+1$ execution nodes. Comparing to original UpRight, our approach can always reduce its agreement cost and can reduce its execution cost when $u>r$. This conclusion shows that, when $u>r$, setting distinct u and r can reduce replication cost comparing to using a unified $f=u$. This is appealing because in most environments, the possibility of crash failures is indeed much higher than that of those bizarre failures, indicating $u>r$ is a

[1]Fast Paxos can be configured in two ways: it can be configured to have $2f+1$ agreement nodes and $f+\lfloor\frac{f}{2}\rfloor+1$ of them must respond for the fast path; it can also be configured to have $3f+1$ agreement nodes and $f+1$ of them must respond. We use the first configuration in the paper for a fair comparison with other Paxos-like protocols.

[2]UpRight further separates agreement into two phases: authentication phase needs a maximum of $u+r+max(u,r)+1$ nodes and order phase needs a maximum of $2u+r+1$ nodes. We only present the larger number in the paper for simplicity.

Protocol	Agreement		Execution			
	N^A_{max}	$N^A_{normal} = N^A_{active}$	N^E_{max}	$N^E_{critical}$	$N^E_{flexible}$	N^E_{active}
Paxos	$2f+1$	$f+1$	$2f+1$	1	$f+1$	$f+1$
Fast Paxos	$2f+1$	$f+\lfloor \frac{f}{2} \rfloor+1$	$2f+1$	1	$f+1$	$f+1$
PBFT	$3f+1$	$2f+1$	$3f+1$	$f+1$	$f+1$	$2f+1$
Yin et.al.	$3f+1$	$2f+1$	$2f+1$	$f+1$	$f+1$	$2f+1$
Zyzzyva	$3f+1$	$3f+1$	$2f+1$	$f+1$	$f+1$	$2f+1$
UpRight	$u+r+max(u,r)+1$	$r+max(u,r)+1$	$u+max(u,r)+1$	$r+1$	$max(u,r)+1$	$u+r+1$

Table 1: Required number of replicas for different protocols. N^A_{max}: max number of agreement nodes; N^A_{normal}: number of agreement nodes in failure-free case; N^A_{active}: number of active agreement nodes in our approach; N^E_{max}: max number of execution nodes; $N^E_{critical}$: number of execution nodes for critical tasks; $N^E_{flexible}$: number of execution nodes for flexible tasks; N^E_{active}: number of active execution nodes in our approach.

practical setting. Note that such opportunity to reduce execution cost by setting distinct u and r does not exist in the original UpRight protocol, because its execution cost $u+max(u,r)+1$ is equal to $2u+1$ when $u>r$ and it is not different from the $2f+1$ bound of early work.

4 Adaptive recovery

Although recovery can be delayed, it has to be performed in a timely manner for two reasons: first, data durability is determined by how frequently machines fail and how fast they can recover. Therefore, increasing recovery time may lead to higher probability of data loss; second, flexible tasks, such as garbage collection, cannot be performed until recovery is complete. If they are delayed for too long, eventually the system will be blocked.

In our approach, ensuring recovery speed is further complicated by the fact that recovery is performed in parallel with executing requests and these two tasks often compete for resource (e.g. network and disk bandwidth).

To complete recovery in a timely manner and to make a balance between recovery and executing new requests, this paper introduces an adaptive recovery mechanism. It allows the administrator to specify a deadline for recovery, which is determined by the required data durability and the frequency of machine failures. Then adaptive recovery attempts to meet the deadline with minimal resource while allocating all remaining resource to executing new requests.

In order for this approach to work, first, we need a dynamic and fine-grain mechanism to control the resource dedicated to recovery. For this purpose, we split the whole recovery into multiple recovery requests, each fetching a subset of the state, and introduce a parameter I_{rec}, which is defined as the system will execute I_{rec} client requests after it executes a recovery request. If no client requests arrive for a certain amount of time, however, this constraint can be relaxed. This parameter allows our approach to dynamically control the speed of recovery.

In order to meet the deadline, our approach tracks the progress of recovery: during recovery, a backup node needs to fetch state from an active node. The active node knows the total size of the state to be transferred and keeps track of how much data has already been transferred. It periodically checks the progress of recovery by comparing $\frac{fetched\ data}{total\ data}$ and $\frac{elapsed\ time}{deadline}$. If the former is smaller (larger) than the latter, it increases (decreases) recovery speed by decreasing (increasing) I_{rec}.

Agreement node recovery. A backup agreement node needs to fetch missing log entries from the leader. In this case, the leader will perform the above tracking and adaptive control mechanism.

Execution node recovery. A backup execution node needs to fetch both the latest snapshot from an active execution node and the log entries afterwards from the leader. In this case, both the active execution node and the leader needs to perform the above tracking and adaptive control mechanism. Furthermore, since the system is still processing new requests in the meantime, the backup execution node will receive those requests. Since it cannot execute them until the recovery is complete, it will cache these new requests (first in memory and then on disk if memory is full). Since these procedures all run in parallel and may compete for resource, they may affect each other, but our adaptive approach should be able to achieve a balance given enough time.

Guarantees? Adaptive recovery makes best effort to meet the deadline but cannot provide any guarantees for several reasons: first, if the deadline is too close, the system may not be able to meet the deadline even if it allocates all resources to recovery. Second, the adaptive approach takes time to monitor progress and adjust recovery speed, so if the recovery time is too short, our approach may not be effective. Finally, our approach relies on the assumption that the system throughputs when processing client requests and recovery requests are reasonably stable. If they can change rapidly, because of either hardware issues (e.g. contention on network) or software issues (e.g. some requests take much longer than others to process), our mechanism may not be accurate.

5 ThriftyPaxos

To demonstrate the effectiveness of our approach, we apply it to Paxos, a popular replication protocol, to build ThriftyPaxos. In this section, we present the detailed protocol of ThriftyPaxos. As one can imagine, it bears a significant resemblance to the standard Paxos protocol. For completeness, we present the whole ThriftyPaxos protocol, but we highlight the different part.

5.1 Overview

Paxos incorporates $2f+1$ replicas. Its key idea is that when a request is agreed by $f+1$ replicas as the next request, the request becomes *stable*, which means the decision will not be changed by future failures. Then all execution nodes execute the stable request.

A ThriftyPaxos service follows the same idea. It logically separates replicas into $2f+1$ agreement nodes and execution nodes (a pair of agreement and execution nodes can be collocated on the same machine or even in a single process). By following the calculations in Section 3, ThriftyPaxos activates $f+1$ of them and reserves f of them as backup (standard Paxos activates all of them).

To order client requests, ThriftyPaxos tries to assign a unique *slot number* to each client request: a client request with a lower slot number is ordered before one with a higher slot number. The safety property requires that each slot can be assigned to at most one client request.

In most of the time, the system elects one agreement node as the *leader*, which proposes the next request to execute. When failures or asynchronous events happen, new leaders may be elected, even if the old leader is still alive. Different leaders may make different proposals for the same slot, and to distinguish them, each leader is assigned a unique *epoch number* and it attaches this epoch number to each proposal it makes. A later elected leader has a higher epoch number than an early leader.

5.2 Basic protocol

$\textcircled{1}$ A client sends a request to the leader.

A client can have multiple outstanding requests and thus the system needs a unique identifier for each request to match a reply to a request. A classic approach to generate request ID is to combine client ID with the number of requests the client has already sent.

If the client does not get a reply in time, either because some messages are lost or because some replicas fail, it resends those request to all replicas.

$\textcircled{2}$ The leader checks the request and if the check passes, the leader proposes it as the next request.

The leader needs to check whether this is the expected request from the corresponding client. In order to check that, the leader maintains the last request ID it has proposed for each client. There are four possible cases:

$\textcircled{2.1}$ If $requestID = lastRequestID[client]+1$, then this is the appropriate request to propose. The leader sends a proposal $< epoch, slot, request >$ to $f+1$ agreement nodes including the leader itself (standard Paxos sends to $2f+1$ agreement nodes). The leader then needs to update corresponding state, such as *slot*.

$\textcircled{2.2}$ If $requestID > lastRequestID[client]+1$, it means some previous client's requests are lost. In this case, the leader can either cache the request or discard it if the cache is full. In either case, the leader needs to wait for the client to resend the missing requests.

$\textcircled{2.3}$ If $requestID < lastRequestID[client]+1$ and the request is in the leader's pending proposals, it means that the leader is processing the request. In this case, the leader performs no action.

$\textcircled{2.4}$ If $requestID < lastRequestID[client]+1$ and the request is not in the leader's pending proposals, it means the leader has already finished proposing this request. This can happen if some messages were lost so that the client did not receive the reply. In this case, the leader forwards the request to execution nodes (see Step $\textcircled{6.2}$).

$\textcircled{3}$ When an agreement node receives a proposal, it decides whether to accept the proposal.

$\textcircled{3.1}$ If $proposal.epoch < this.epoch$ or if the agreement node is not active, the agreement node discards the proposal and notifies the leader.

$\textcircled{3.2}$ Otherwise, the agreement node accepts the proposal by logging it to disk and sending an acknowledgement to the leader. If $proposal.epoch > this.epoch$, the agreement node also updates its own epoch. Note that the agreement node can accept proposals out of order.

$\textcircled{4}$ The leader waits for $f+1$ acknowledgements.

If any agreement node responds that it has accepted a proposal with a higher epoch number, this leader gives up and will not process any further requests from the clients.

If any agreement node times out, the leader picks up one backup agreement node and sends an *ACTIVATE* command to it (standard Paxos, of course, does not need this command because all replicas are active). Then the leader resents all pending proposals.

$\textcircled{5}$ The leader sends the agreed request to all active execution nodes.

If there are less than $f+1$ active execution nodes, the leader will send an *ACTIVATE* command to some backup execution nodes (standard Paxos does not need this).

$\textcircled{6}$ An execution node executes the request.

An execution node needs to execute requests in order. In order to achieve that, it maintains the last slot that it has already executed. When receiving a request, it first checks the slot number of the request.

$\textcircled{6.1}$ If $request.slot = lastslot+1$, the request is the next to execute. The execution node executes the request and sends the reply to the client. It then checks whether there are any following requests in the cache (6.3) that can be executed.

6.2 If *request.slot* $<=$ *lastslot*, then this request has already been executed. This might happen if the reply message got lost and the client resent the request. In this case, the execution node should resend the reply to the client. To achieve that, the execution node needs to maintain a reply cache for replies it has already sent. To garbage collect replies, a client can piggyback in each of its request the ID of the latest received reply.

6.3 If *request.slot* $>$ *lastslot* $+1$, the execution node caches the request or stores the request to disk if the cache is full. This could happen if a previous message was lost or the execution node is in recovery. For the former case, the execution node asks the leader to resend missing requests. For the later case, the execution node has to wait for the recovery to complete.

To garbage collect logs on the agreement nodes (3.2), ThriftyPaxos requires execution nodes to periodically take snapshots of their states, promising that they will not need earlier requests in the future. ThriftyPaxos performs garbage collection when $f+1$ execution nodes complete taking snapshots (standard Paxos asks an agreement node to garbage collect its log when its corresponding execution node takes a snapshot. This is not different from ThriftyPaxos when $f+1$ execution nodes are active, but since in ThriftyPaxos, sometimes there could be different number of agreement nodes and execution nodes, we use a more explicit rule).

5.3 Failure recovery

Recovering execution nodes. When an execution node is not responding, the leader sends an *ACTIVATE* command to a backup execution node, asking it to rebuild its state. Meanwhile, ThriftyPaxos proceeds with the remaining active execution nodes. To rebuild state, the backup execution node may need to fetch the latest snapshot from an active execution node and fetch following logs from the leader. Since the system is processing clients' requests during recovery, and the backup node cannot process them at the moment, it will cache these requests first in memory and then in a log file on disk if the memory buffer is full. After state transfer is complete, the backup node needs to load the snapshot and replay all logs afterwards.

We have applied adaptive recovery at the leader and at the active execution node to control the recovery speed of the backup execution node. When trying to meet the deadline, adaptive recovery considers recovery as complete when state transfer is done, because at that moment, the backup node has the same state as an active one: this means data durability is fully recovered and the system can garbage collect logs on agreement nodes. Therefore, later operations, including loading snapshot and replaying logs, are not considered as part of recovery.

Recovering agreement nodes. When a non-leader agreement node is not responding, the leader will send an *ACTIVATE* command to a backup agreement node, asking it to join the agreement protocol immediately. Meanwhile, to restore data durability, the backup node fetches missing logs from the leader in the background. We have applied adaptive recovery at the leader to control the speed of fetching missing logs.

When the leader is not responding, a new leader is elected. The new leader collects logs from other agreement nodes and re-proposes pending requests. ThriftyPaxos uses the same protocol as standard Paxos for leader switch (backup agreement nodes also need to participate in a leader switch). This part is out of the control of adaptive recovery, because the system is blocked anyway. The developer can make a trade-off between performance and availability by setting the snapshot interval, since only requests after the last snapshot need to be re-proposed durng a leader switch. Note that this issue exists in both ThriftyPaxos and standard Paxos.

6 Implementation

We have implemented ThriftyPaxos from scratch in Java. This section discusses some implementation details.

Failure detection. A highly available system must be able to detect failures quickly. Like standard Paxos, ThriftyPaxos does not rely on the accuracy of failure detection for safety and thus can use aggressive approaches (e.g. short timeout) for better availability. Our experiments, however, show that if timeout interval is too short, performance can be unstable, because even a minor abnormal event, such as a long disk write, can trigger an unnecessary recovery. Our implementation incorporates ideas from accurate failure detection techniques [40]: when failures can be detected for certain, our system should take actions immediately. To cover failures that cannot be accurately detected, we use a short but not too aggressive timeout (5 seconds). Our current implementation only incorporates part of the functionalities of those accurate failure detection techniques, and we leave the full incorporation as future work.

Out-of-order logging. Agreement nodes may need to log requests out of order on disks. To achieve efficient logging, ThriftyPaxos incorporates a design that is similar to SSTable in Bigtable [15]: if the slot number of the new request is larger than the last one in the current file, ThriftyPaxos appends it to the current file; otherwise, ThriftyPaxos closes the current file, creates a new one, and appends the next log entry into the new file. Such design guarantees that each log file is sequential, and thus it is simple to perform operations like log scan (merge sort) and garbage collection. Furthermore, when there are no out-of-order requests, logging is fully sequential and thus can fully utilize a hard drive.

Leader election. During leader switch, the new leader needs to collect logs from other agreement nodes and repropose pending requests (Section 5.3). To reduce I/O traffic during leader election, ThriftyPaxos gives a preference to an active agreement node as the new leader, because it has more logs compared to a backup node.

When to start recovery. Previous study shows that it may be a waste of resource to start recovering a node right after it fails, because a crashed node has is likely to come back soon. In practice, Google starts to recover a node if it cannot come back in 15 minutes [23]. Our approach follows the same idea: when a backup node fails, ThriftyPaxos waits a while to see whether it can come back before triggering recovery. We use a relatively short interval (5 minutes) to shorten the length of experiments.

7 Evaluation

Our evaluation tries to answer three questions:

- What is the performance of ThriftyPaxos, when there are no failures?
- What is the availability of ThriftyPaxos, when failures occur?
- Can adaptive recovery meet the deadline under different settings?

To address these questions, we have applied ThriftyPaxos to replicate H2 [25], a database system, and RemoteHashMap, a benchmark application built by us.

H2. H2 is a light-weight database system implemented in Java. It is often used as an embedded data store for larger projects, such as Hadoop [26]. To apply ThriftyPaxos to H2, we have modified H2 to send and receive messages through ThriftyPaxos. We configure H2 to store all tables in memory while keeping logs and snapshots on disks. To achieve quick snapshot, we configure H2 to store data in btrfs, a Linux file system that supports copy on write (COW) snapshot: whenever an execution node needs to perform a snapshot, it asks btrfs to perform a snapshot, which usually takes only several seconds. We ran TPC-C [52] over H2 to measure its performance. To avoid non-determinism, we ran H2 in single-threaded mode, which of course limits its performance. Efficient deterministic multithreading is an orthogonal topic that has been discussed in other works [2, 7, 8, 19, 20, 41] and we leave the incorporation as future work.

RemoteHashMap. To test ThriftyPaxos under various workloads, we have built RemoteHashMap, an application that allows clients to get and set key value pairs on a remote server. RemoteHashMap allows us to change

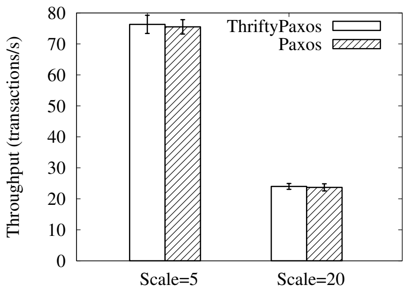


Figure 1: Throughput of TPC-C over replicated H2

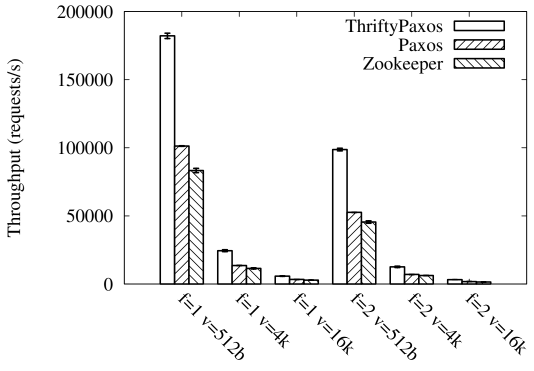


Figure 2: Throughput of writing to replicated RemoteHashMap (v is value size). We show ZooKeeper as a comparison.

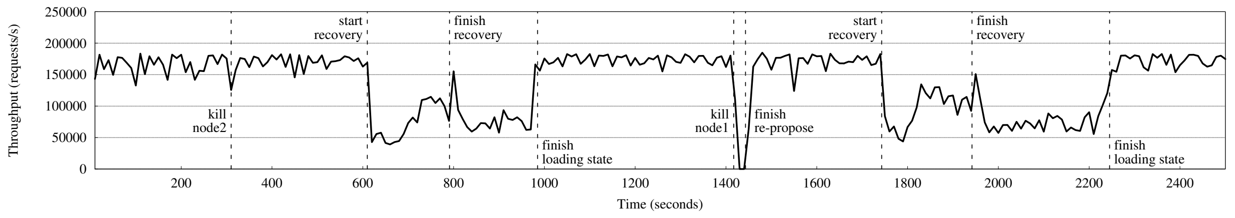
system parameters (e.g. request size, snapshot size) arbitrarily to test different aspects of ThriftyPaxos.

We run all H2 experiments on three Dell R730 machines, which are equipped with 16 cores, 64GB of memory, one SSD drive, and seven hard drivers. We store H2's data on the SSD drive and store agreement nodes' data on one hard drive. We run all RemoteHashMap experiments on 7 Dell R220 machines, which are equipped with 8 cores, 16GB of memory, and two hard drivers. We store RemoteHashMap's data on one hard drive and store agreement nodes' data on the other hard drive.

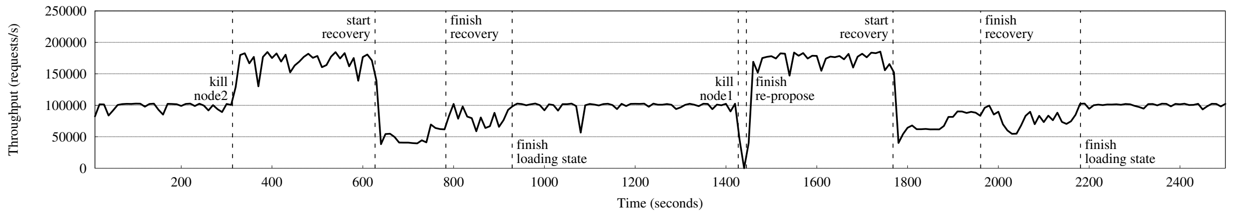
We compare ThriftyPaxos to standard Paxos and Cheap Paxos. We implement standard Paxos and Cheap Paxos by slightly modifying ThriftyPaxos: standard Paxos chooses all $2f+1$ replicas as active ones; Cheap Paxos sets I_{rec} to 0 during recovery, indicating it cannot execute client requests until recovery completes. In all experiments, we collocate an agreement node and an execution node on a same machine.

7.1 Performance

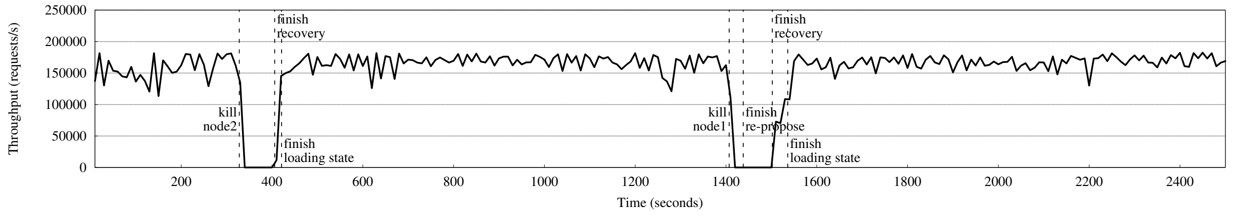
Our first set of experiments aims at comparing the performance of ThriftyPaxos to that of standard Paxos, when there are no failures. Cheap Paxos' protocol in the failure-free case is exactly the same as ThriftyPaxos, so we do not include it in the comparison.



(a) ThriftyPaxos



(b) Standard Paxos



(c) Cheap Paxos

Figure 3: Availability of ThriftyPaxos, standard Paxos, and Cheap Paxos (f=1; node1 is the leader; v=512b).

As shown in Figure 1, when running TPC-C over H2, the performance of ThriftyPaxos and standard Paxos are almost identical. This is because the bottleneck of the system lies in the execution of requests as a result of single-threaded execution: our profiling shows that one CPU core is fully utilized.

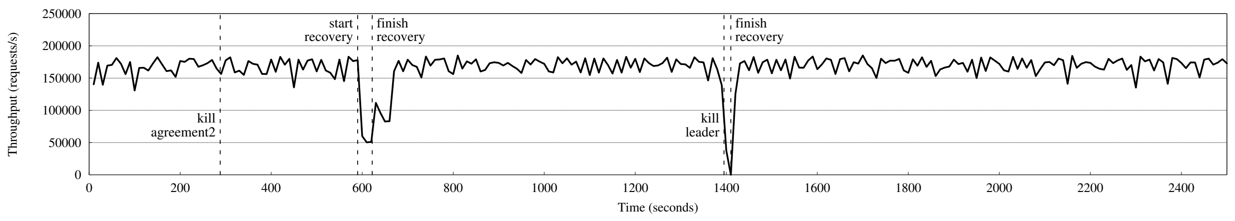
Figure 2 shows the throughput of writing key-value pairs to RemoteHashMap, whose execution is relatively light and agreement is the bottleneck. In this case, ThriftyPaxos outperforms standard Paxos by 73% to 88%, because in ThriftyPaxos, the leader, which is the bottleneck, only needs to send messages to f replicas, while in standard Paxos, the leader needs to send to $2f$. We also show the throughput of ZooKeeper [27], a mature open-source software whose update protocol is similar to standard Paxos. Since we are interested in agreement throughput, we disable the "sync" call when logging to disk for all systems. ZooKeeper cannot serve as a direct comparison to ThriftyPaxos, because they have different features and optimizations. We show ZooKeeper's throughput only to present that our implementation provides a reasonable throughput.

7.2 Availability

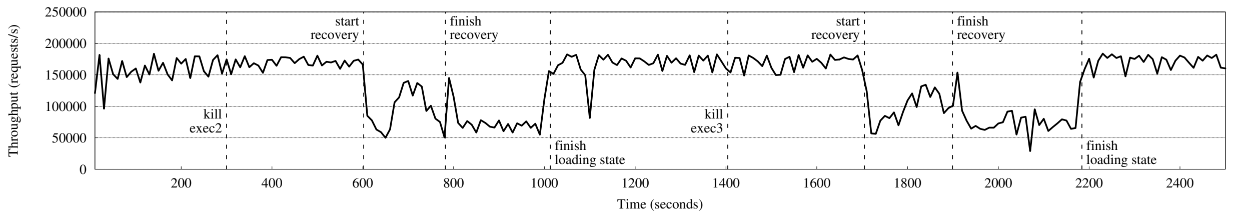
Our second set of experiments compare the availability of ThriftyPaxos to that of standard Paxos and Cheap Paxos, when failures occur.

We use RemoteHashMap in this set of experiments, because RemoteHashMap's much higher throughput creates a higher pressure on network and disk I/Os, incurring more contention against recovery. To measure availability, we kill a non-leader replica at 300 seconds, and kill the leader at 1400 seconds, for all three systems. Since an agreement node and an execution node are collocated on the same machine, both of them will be killed.

As shown in Figure 3, the behavior of ThriftyPaxos and standard Paxos are quite similar: when a non-leader replica fails, the system can continue processing requests; when the leader fails, the system needs to elect a new leader, which may block the system for a short while. In either case, since the failed replica does not come back in five minutes, the system needs to rebuild its state on another replica: rebuilding incurs network and disk I/Os, degrading system performance. The behaviors of ThriftyPaxos and standard Paxos are different in two ways: first, after a replica fails in standard Paxos, system throughput jumps. This is because standard Paxos needs to send fewer messages after a failure. ThriftyPaxos, on the other hand, has already exploited this opportunity in the failure-free case and thus its throughput remains stable after a failure. Second, ThriftyPaxos' loading time after recovery is slightly longer than that of standard Paxos. This is because ThriftyPaxos' throughput during



(a) Impact of recovering agreement node



(b) Impact of recovering execution node

Figure 4: Impact of recovering agreement node and recovering execution node

recovery is higher than that of standard Paxos (because, once again, ThriftyPaxos sends fewer messages for each request), and thus ThriftyPaxos needs to replay more log entries afterwards.

Cheap Paxos shows a different behavior: when either replica fails, Cheap Paxos needs to rebuild its state before it can process new requests. Therefore, the system is halted during node recovery. While its recovery takes 78-96 seconds in our experiments to transfer about 8GBs of state (including both snapshot and following logs), it may halt longer when application's state is larger.

We further decouple recovery of agreement node and recovery of execution node to understand their individual impact. As shown in Figure 4, recovery of execution node certainly has a bigger impact on performance, because it needs to transfer more state. Recovery of agreement node is lighter in general, but the failure of the leader can halt the system for a short while.

In both figures, a higher throughput during recovery increases the time to load state afterwards. This is simply because higher throughput during recovery increases the number of log entries to be replayed after recovery. Faster disks or a larger memory buffer may reduce such replaying time.

7.3 Adaptive recovery

Our last set of experiments measures whether adaptive recovery can meet the deadline. We set different deadlines and sizes of snapshots for this set of experiments.

Figure 5 shows the impact of different recovery deadlines. As one can observe, closer deadline, which means the system must dedicate more resource to recovery, can cause a sharper degradation of performance during recovery: the average throughput of deadline 100, 200, and 300 are 22240, 74690, and 99339, respectively.

Figure 6 shows the impact of different snapshot sizes. As one can observe, bigger snapshot, which means the system must dedicate more resource to recovery, can cause a sharper degradation of performance during recovery. Once again, our adaptive recovery mechanism successfully meets the deadline in all experiments.

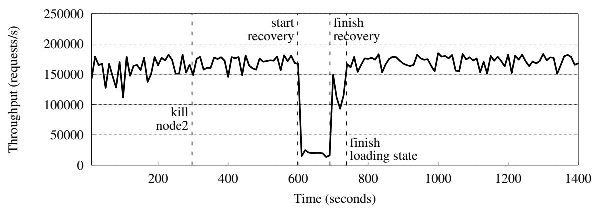
Our adaptive recovery mechanism successfully meets the deadline in all experiments. Although these results may not be extended to other applications, as discussed in Section 4, they demonstrate the efficacy of adaptive recovery for applications that can provide a stable throughput when processing client requests and recovery requests. Furthermore, in all experiments, the recovery completion time is close to the deadline. This demonstrates that adaptive recovery achieves its goal: to meet the deadline with minimal resource while using all remaining resource to process new requests.

8 Related work

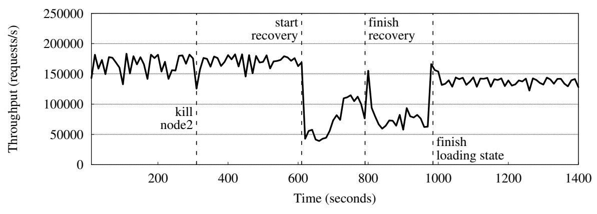
While Section 2 already presents important related works in detail, this section presents a broader survey.

State machine replication. State machine replication (SMR) [49] replicates an application's state machine on multiple machines to tolerate failures. Based on the types of failures it can tolerate, SMR protocols can be broadly classified into two categories: the Paxos family [31, 32, 45] that can tolerate machine crashes and timing errors, and the Byzantine Fault Tolerance (BFT) family [1, 6, 13, 16, 18, 28, 36] that can tolerate arbitrary errors.

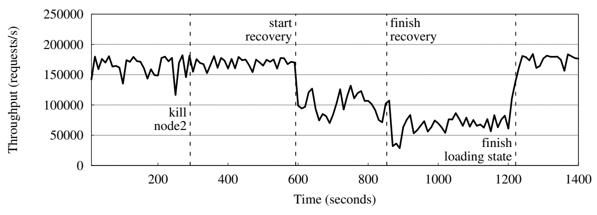
Separating agreement from execution. Paxos separates agreement (acceptors) from execution (learners) to clarify its protocol [14, 31, 32, 37, 46, 53]. Yin et. al. [57] observe that for BFT protocols, the number of required



(a) Deadline=100 seconds

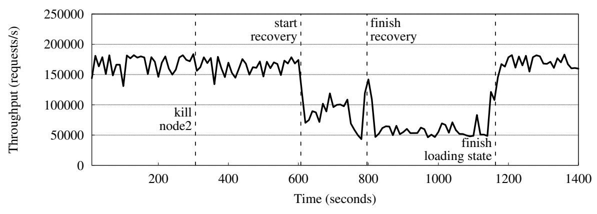


(b) Deadline=200 seconds

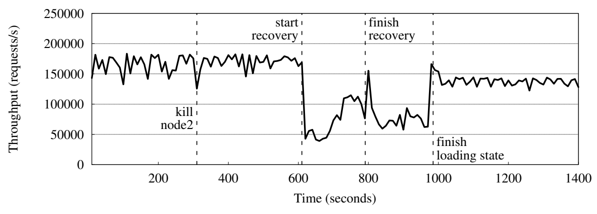


(c) Deadline=300 seconds

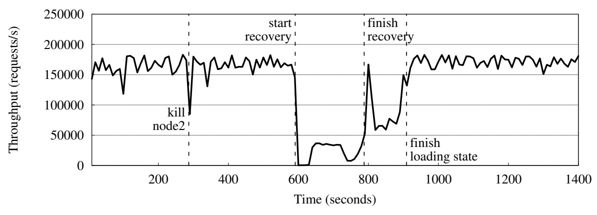
Figure 5: Recovery with different deadlines (snapshot size=5G; node2 is a non-leader replica)



(a) Snapshot size=1G



(b) Snapshot size=5G



(c) Snapshot size=15G

Figure 6: Recovery with different snapshot sizes (deadline=200 seconds; node2 is a non-leader replica)

execution replicas could be fewer than that of the agreement nodes. This observation is inherited in later works, such as UpRight [16], Zyzzyva [28], and ZZ [56].

On-demand instantiation. Cheap Paxos [35] and ZZ [35] activate minimal number of replicas first and activate backup ones when active ones are unresponsive. However, they require a backup node to rebuild its state before it can process requests, suffering from the availability problem. Distler et al. [22] proposes to split application state into multiple objects and perform on-demand instantiation for each object: this approach can alleviate the availability problem, but since it needs to maintain a log for each object, the space overhead is significantly magnified. Parallel recovery techniques [15, 47] can effectively reduce recovery time and thus can alleviate the availability problem of on-demand instantiation, but recovering a node right after it fails is a waste of resource, since most crashed nodes can come back soon [23].

Lazy recovery. Lazy recovery [29, 30] is widely used in many systems. For example, Google File System [24] starts recovering a failed node if it cannot come back in 15 minutes. Silberstein et.al. [51] applies lazy recovery to erasure-coding systems to reduce recovery overhead.

Other optimizations. Vertical Paxos [34] proposes to strengthen primary backup protocols with the help of a centralized Paxos service. Our approach provides a more general approach that does not require an additional Paxos service and that is applicable to BFT protocols.

Falcon [40] and its following works [38, 39] attempt to build accurate failure detectors, which make Paxos unnecessary. However, they rely on routers to monitor the status of machines and thus can become unavailable when routers fail, which can happen in today's datacenters [23]. Furthermore, it is inapplicable to BFT systems.

Gaios [9] and ZooKeeper [27] execute read-only requests on only one replica. Fast Paxos [33], Speculative Paxos [48], and Zyzzyva [28] introduce a fast path to reduce latency. Mencius [42] and EPaxos [43] allow multiple leaders to propose requests in parallel to achieve load balancing. These optimizations are orthogonal to our approach, although they may affect the effectiveness of our approach, as discussed in Section 3.

9 Conclusion

Whether to pay the cost of a strong replication protocol has been a painful question for developers. Instead of inventing new protocols, this paper presents a general approach to reason about the necessary conditions for correctness and availability in existing protocols. It shows that, with a deeper understanding of existing protocols, we can reduce their replication cost while maintaining their correctness and availability properties.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *SOSP*, 2005.

[2] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. NSDI'12, 2012.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. SOSP '13.

[6] Rida A. Bazzi. Synchronous Byzantine quorum systems. *Distributed Computing*, 13(1):45–52, 2000.

[7] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 2010.

[8] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[9] 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 *NSDI*, 2011.

[10] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transanctions on Computer Systems*, 14(1):80–107, February 1996.

[11] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[12] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[13] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transanctions on Computer Systems*, 20(4):398–461, November 2002.

[14] T. Chandra, R. Griesmer, and J. Redstone. Paxos made live – an engineering perspective. In *Proc. 26th PODC*, June 2007.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[16] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. UpRight Cluster Services. In *SOSP*, 2009.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[18] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, 2006.

[19] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos Made Transparent. In *SOSP*, 2015.

[20] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[21] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[22] Tobias Distler and Rüdiger Kapitza. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Eurosys*, 2011.

[23] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[25] H2. The H2 home page. http://www.h2database.com.

[26] Hadoop. `http://hadoop.apache.org/core/`.

[27] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, 2007.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, 1992.

[30] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 43–57, New York, NY, USA, 1990. ACM.

[31] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[32] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

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

[34] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, 2009.

[35] Leslie Lamport and Mike Massa. Cheap Paxos. In *DSN*, 2004.

[36] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[37] Butler Lampson. The abcds of paxos. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, page 13, New York, NY, USA, 2001. ACM.

[38] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, 2013. USENIX.

[39] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, New York, NY, USA, 2015. ACM.

[40] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP*, 2011.

[41] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[42] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *OSDI*, 2008.

[43] 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.

[44] MySQL. `http://www.mysql.com`.

[45] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proc. 7th PODC*, 1988.

[46] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[47] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[48] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[49] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[51] Mark Silberstein, Lakshmi Ganesh, Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 15:1–15:7, New York, NY, USA, 2014. ACM.

[52] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc/.

[53] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

[54] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[55] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *USENIX ATC*, 2012.

[56] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the Art of Practical BFT. In *Eurosys*, 2011.

[57] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.