High Performance State-Machine Replication

Parisa Jalili Marandi
University of Lugano
Switzerland

Marco Primi
University of Lugano
Switzerland

Fernando Pedone
University of Lugano
Switzerland

Abstract

State-machine replication is a well-established approach to fault tolerance. The idea is to replicate a service on multiple servers so that it remains available despite the failure of one or more servers. From a performance perspective, state-machine replication has two drawbacks. First, it introduces some overhead in service response time, due to the requirement to totally order commands. Second, service throughput cannot be augmented by adding replicas to the system. We address the two issues in this paper. We use speculative execution to reduce the response time and state partitioning to increase the throughput of state-machine replication. We illustrate these techniques with a highly available B-Tree service.

1 Introduction

Computer systems are usually made fault tolerant through replication. By replicating a service on multiple servers, clients have the guarantee that even if some replicas fail, the service is still available. However, once a service is replicated, consistency among the replicas must be ensured. State-machine replication is a well-known approach to replication. It achieves strong consistency by regulating how client commands must be propagated to and executed by the replicas [18, 25]. Command propagation can be decomposed into two requirements: (i) every nonfaulty replica must receive every command and (ii) no two replicas can disagree on the order of received and executed commands. Moreover, command execution must be deterministic: if two replicas execute the same sequence of commands in the same order, they must reach the same state and produce the same output.

State-machine replication is a technique to improve service availability. From a performance perspective it suffers from two shortcomings. First, it introduces some overhead in the service *response time* when compared to a single-copy client-server service. Second, the service *throughput* is limited by the throughput of a single replica. Thus, if demand augments (e.g., more clients join the system) it cannot be absorbed by adding replicas to the system. The increased response time stems from the need to order client commands before they can be executed: ordering commands is inherently more costly than sending them directly to a server, as in a client-server setup. The throughput limitation is a consequence of each replica storing a full copy of the service state and handling every command. In this paper we address each one of these issues.

To reduce the overhead in response time we rely on speculative (or optimistic) execution, a technique that has been used before in the context of replicated databases (e.g., [14, 16]). The idea is to expose servers to a command before its final order has been established. As a result, the execution of the command by the server and the execution of the protocol that orders the command can overlap in time, saving in response time. The technique is speculative because it only works if the order in which commands are executed is confirmed by the ordering protocol. If not, the commands must be rolled back and re-executed in the correct order (i.e., the order defined by the ordering protocol). We exploit this technique in the context of Ring Paxos, a high throughput consensus protocol used to implement state-machine replication. As we explain in the paper, speculative execution in Ring Paxos does not depend on network conditions (e.g., spontaneous message order), and therefore is more advantageous than previous proposals (e.g., [14, 16]).

We address the throughput limitation of state-machine replication with a state partitioning strategy. In brief, we allow applications to decompose their state into sub-states and replicate each sub-state individually. Commands are directed to and executed by the appropriate partitions only. By partitioning the state of a service, we allow to process commands in parallel. This is particularly effective for services whose state partitioning is *perfect*, that is, all commands access one sub-state or another, but no command accesses two or more sub-states. Commands that access more than one sub-state must be carefully ordered to avoid inconsistencies. We discuss how to efficiently integrate the technique into Ring Paxos.

To illustrate high performance state-machine replication, we propose, implement, and fully evaluate a highly available parallel B-Tree service. Our service implements three B-Tree operations: inserts, deletes, and range queries. We show that speculative execution can reduce response time by up to 16.2%. State partitioning allows the service to scale by adding replicas with a resulting throughput near 4 times greater than classic state-machine replication. In our largest configuration, up to three quarters of a million B-Tree commands can be executed per second with a response time below 4 milliseconds.

Summing up, the paper makes the following contributions: (1) It shows how speculative execution can be integrated into Ring Paxos to reduce the response time of state-machine replication. (2) It presents the idea of state partitioning in the context of state-machine replication. (3) It illustrates the techniques with a B-Tree service capable of executing commands very efficiently. (4) It discusses the implementation of these ideas and fully assesses them experimentally.

The remainder of the paper is structured as follows. Section 2 describes our system model and state-machine replication. Sections 3 presents speculative execution and state partitioning in detail. Section 4 illustrates the approach with a highly available parallel B-Tree service. Section 5 evaluates the performance of the B-Tree service. Section 6 comments on related work. Section 7 concludes the paper. A proof sketch of the correctness of our protocols can be found in the appendix.

2 Background

2.1 System model

We assume a distributed system composed of interconnected nodes within a single geographical location (e.g., a data center). Nodes may fail by crashing and subsequently recover, but do not experience arbitrary behavior (i.e., no Byzantine failures). The network is mostly reliable and subject to small latencies, although load unbalances (e.g., peak demand) imposed on both nodes and the network may cause variations in processing and transmission delays. Communication can be one-to-one, through the primitives *send*(p, m) and *receive*(m), and one-to-many, through the primitives *ip-multicast*(g, m) and *ip-deliver*(m), where m is a message, p is a node, and g is a group of nodes. Messages can be lost but not corrupted.

Our protocols ensure safety under both asynchronous and synchronous execution periods. The FLP impossibility result [11] states that under asynchronous assumptions consensus cannot be both safe and live. We thus assume that the system is *partially synchronous* [8], that is, it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [8], and it is unknown to the nodes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown. After GST nodes are either *correct* or *faulty*. A correct node is operational "forever" and can reliably exchange messages with other correct nodes. This assumption is only needed to prove liveness properties about the system. In practice, "forever" means long enough for one instance of consensus to terminate.

2.2 State-machine replication

State-machine replication is a fundamental approach to implementing a fault-tolerant service by replicating servers and coordinating client commands among server replicas [18, 25]. The precise way in which the technique is implemented depends on the targeted consistency criteria, which in this paper we assume to be *linearizability*.

An execution is linearizable if there is a way to reorder its commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the order of non-overlapping commands across all clients [3]. Linearizability can be contrasted with *sequential consistency*, a weaker form of consistency: An execution is sequentially consistent if there is a way to reorder the commands in a sequence that (i) respects their semantics, and (ii) respects the ordering of commands issued by the same client [3].

In the execution on top of Figure 1, client C_2 modifies the state of a read-write object x and then client C_1 reads a state of x that precedes C_2's update (e.g., by accessing a replica that has not seen C_2's changes yet). This execution is not linearizable but it is sequentially consistent. The execution on the bottom of Figure 1 is both linearizable and sequentially consistent: C_1 is allowed to see a value of x that precedes C_2's update since the two commands overlap in time.

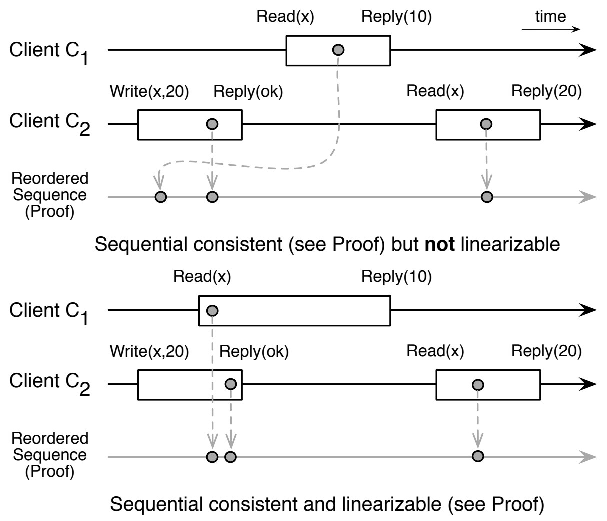

Read(x)
Reply(10)
time
Client C1
Write(x,20)
Reply(ok)
Read(x)
Reply(20)
Client C2
Reordered Sequence (Proof)
Sequential consistent (see Proof) but not linearizable
Read(x)
Reply(10)
Client C1
Write(x,20)
Reply(ok)
Read(x)
Reply(20)
Client C2
Reordered Sequence (Proof)
Sequential consistent and linearizable (see Proof)

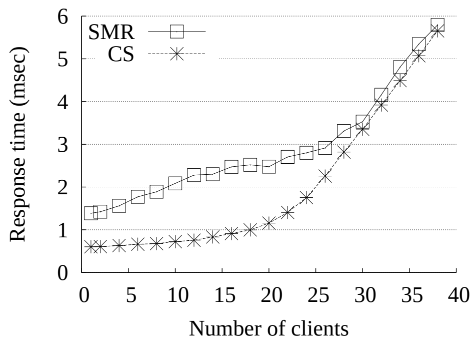
Figure 1: Linearizabiliy vs. sequential consistency.

State-machine replication can be implemented as a series of *consensus* instances [19]. The i-th consensus instance decides on the i-th command (or batch of commands) to be executed by the servers. Consensus is defined by the primitives *propose*(v) and *decide*(v), where v is an arbitrary value, a command to be executed by the servers. Consensus guarantees that (i) if a server decides v then some client proposed v; (ii) no two servers decide different values; and (iii) if one (or more) non-faulty client proposes a value then eventually some value is decided by all non-faulty servers.

With respect to performance, state-machine replication suffers from two shortcomings: First, totally ordering commands delays their execution and consequently the response time experienced by the clients, when compared to a non-replicated client-server setup. Second, since every replica contains a full copy of the service state and must receive every command, limited or no performance improvement can be expected from adding replicas to the system. Notice that some performance improvement can be obtained from a few optimizations. Read commands need not be executed by all replicas: upon deciding on a read command, only one server (e.g., randomly assigned) must execute the command and return the results to the client. Although all servers must execute update commands, the response from only one server is sufficient for the clients. Obviously, if the server assigned to return the results to the client fails, the client has to retransmit its request. Since fails are (hopefully) rare events, this is the design we follow in this paper.

Figure 2 compares the performance of a replicated system to a non-replicated client-server system with a workload composed of read-only commands only—more details about these experiments can be found in Sections 4 and 5. The graph on the left of Figure 2 shows the response time of the two systems as the number of clients increases. The difference between the two curves before saturation (28 clients) indicates the overhead introduced by replication. The graph on the right of Figure 2 shows the throughput of the system as replicas are added. Since the workload is composed of read operations only, replication can improve throughput up to four replicas; with eight replicas, the overhead of simply delivering and discarding read commands prevents the system from scaling further.

To conclude, we claim that these are fundamental performance limitations, not implementation specific. State-machine replication requires commands to be ordered, and ordering commands is inherently more expensive than directly sending them to a server. Moreover, the fact that all replicas must deliver all commands—although not all commands must be executed by all replicas—limits the attainable performance (see [15] for a similar argument). In the next section we describe two mechanisms that ad-



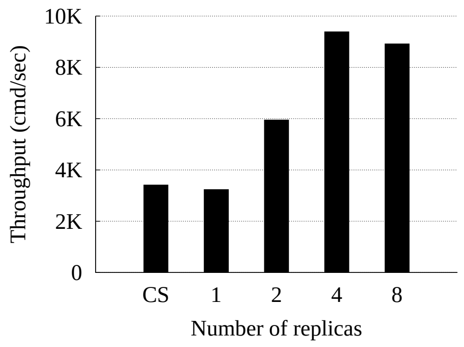


Figure 2: Client-server (CS) versus state-machine replication (SMR) executing read-only commands. (Left) Response time versus number of clients. (Right) Throughput versus number of replicas.

dress these problems.

3 High Performance SMR

Our approach to improving the performance of state-machine replication consists in addressing its two fundamental limitations: we show how to reduce the response time and how to increase the throughput of a replicated system. This work has been conducted in the context of Ring Paxos, a high throughput total order broadcast protocol. In the following, we first recall Ring Paxos (Section 3.1), and then introduce each one of our contributions (Sections 3.2 and 3.3).

3.1 Ring Paxos outline

Ring Paxos is a variation of Paxos [19], optimized for clustered systems. Paxos distinguishes three roles: *proposers*, *acceptors*, and *learners*. A node can execute one or more roles simultaneously. In a client-server setup, clients act as proposers and servers as learners. A value is a command proposed by a client to be executed by the servers; the decided value is the next command to be executed. Each instance of Paxos proceeds in two phases: During Phase 1, the coordinator selects a unique round number *c-rnd* and asks a quorum Q_a (i.e., any majority) of acceptors to promise for it. By promising, an acceptor declares that, for that instance, it will reject any request (Phase 1 or 2) with round number less than *c-rnd*. Phase 1 is completed when Q_a confirms the promise to the coordinator. Notice that Phase 1 is independent of the value, therefore it can be pre-executed by the coordinator. If any acceptor already accepted a value for the current instance, it will return this value to the coordinator, together with the round number received when the value was accepted (*v-rnd*).

Once a coordinator completes Phase 1 successfully, it can proceed to Phase 2. Phase 2 messages contain a value and the coordinator must select it with the following rule: if no acceptor in Q_a accepted a value, the coordinator can select any value (i.e., the next client-submitted value). If however any of the acceptors returned a value in Phase 1, the coordinator is forced to execute Phase 2 with the value that has the highest round number *v-rnd* associated to it. In Phase 2 the coordinator sends a message containing a round number (the same used in Phase 1). When receiving such requests, the acceptors acknowledge it, unless they have already acknowledged another message (Phase 1 or 2) with a higher round number. They update their *c-rnd* and *v-rnd* variables with the round number in the message. When a quorum of acceptors accepts the same round number (Phase 2 acknowledgement), consensus terminates: the value is permanently bound to the instance, and nothing will change this decision. It is therefore safe for learners and deliver the value. Learners learn this decision either by monitoring the acceptors or by receiving a decision message from the coordinator.

As long as a nonfaulty coordinator is eventually selected, there is a majority quorum of nonfaulty acceptors, and at least one nonfaulty proposer, every consensus instance will eventually decide on a value. A failed coordinator is detected by the other nodes, which select a new coordinator. If the coordinator does not receive a response to its Phase 1 message it can re-send it, possibly with a bigger round number. The same is true for Phase 2, as long as the same round number is used. If the coordinator wants to execute Phase 2 with a higher round number, it has to complete Phase 1 with that round number beforehand. Eventually the coordinator will receive a response or will suspect the failure of an acceptor.

Ring Paxos [21] differs from Paxos in a few aspects that make it more throughput efficient:

- Acceptors are organized in a logical ring. The coordinator is one of the acceptors. Phase 1 and 2 messages are forwarded along the ring, each acceptor appends its decision so that the coordinator, at the end of the ring, can know the outcome (Step 3 in Figure 3).
- Ring Paxos executes consensus on value IDs. That is, for each client value, a unique identification number is selected by the coordinator. Consensus is executed on IDs which are usually significantly smaller than the real values.
- The coordinator makes use of ip-multicast. It triggers Phase 2 by multicasting a packet containing the client value, the associated ID, the round number and the instance number to all acceptors and learners (Step 2 in Figure 3).
- The first acceptor in the ring creates a small message containing the round number, the ID and its own decision and forwards it along the logical ring.
- An additional acceptor check is required to guarantee safety. To accept a Phase 2 message, the acceptor must know the client value associated with the ID contained in the packet.
- Once consensus is reached, the coordinator can inform all the learners by just confirming that some value ID has been chosen. The learner will deliver the corresponding client value in the appropriate instance (Step 4 in Figure 3). This information can be piggybacked to the next ip-multicast message.

Message losses may cause learners to receive the value proposed without the notification that it was accepted, the notification without the value, or none of them. Learners recover lost messages by inquiring other nodes. Ring Paxos assigns each learner to a preferential acceptor in the ring, to which the learner can ask lost messages. Lost Phase 1 and 2 messages are handled like in Paxos. The failure of a node (acceptor or coordinator) requires a new ring to be laid out.

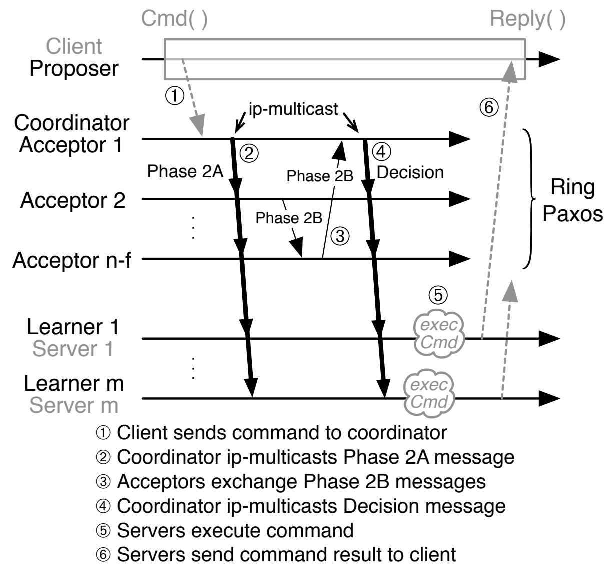

Cmd( )
Reply( )
Client
Proposer
①
ip-multicast
⑥
Coordinator
Acceptor 1
②
④
Phase 2A
Phase 2B
Decision
Acceptor 2
Phase 2B
③
Ring
Paxos
Acceptor n-f
⑤
Learner 1
Server 1
exec
Cmd
Learner m
Server m
exec
Cmd
① Client sends command to coordinator
② Coordinator ip-multicasts Phase 2A message
③ Acceptors exchange Phase 2B messages
④ Coordinator ip-multicasts Decision message
⑤ Servers execute command
⑥ Servers send command result to client

Figure 3: Ring Paxos in a client-server setup (n acceptors, up to f of which can fail, and m learners/servers)

3.2 Speculative execution

The response time experienced by a client of a replicated service can be decomposed into three activities: (a) proposing and ordering a command, (b) executing the command at the servers, and (c) transmitting the response to the client. A reduction in the duration of any of these activities will likely decrease response time. However, in the context of Ring Paxos this is not easy to do since the protocol is already highly optimized and it seems unlikely that it can be significantly improved to accommodate high throughput and lower response time. Moreover, the delay incurred by the execution of a command and the transmission of its response is mostly service specific.

We resort to a speculative (or optimistic) strategy which consists in overlapping part of the ordering protocol (i.e., Ring Paxos) with the execution of commands. In Ring Paxos, a command reaches the servers before its ordering information (Steps 2 and 4 in Figure 3, respectively). When a command arrives, it is buffered by the server and only executed once its order is known. We propose to execute the command immediately after it is received, avoiding any buffering. In doing so, servers can start processing the command before its order is confirmed, saving some time.

A server can only respond to a client after it has executed the command and its order is confirmed. The mechanism is speculative because it works as long as the order in which commands arrive at the servers (and thus the order in which they are executed) is confirmed. In rare occasions (discussed below) commands may be executed out-of-order. If the order in which one or more commands were executed is not confirmed, the server must *rollback* them and re-execute the commands in the proper order. Rolling back a command is service-specific and can be done physically (e.g., by using an undo log) or logically (e.g., by executing an action that reverses the effects of the out-of-order command) [28]. We illustrate logical rollback in Section 4.

Fortunately, in Ring Paxos the order assigned by the coordinator when a command is ip-multicast is always confirmed by the acceptors. The only situation in which the execution order of a command may change is when the coordinator is replaced by another process (e.g., due to a crash), a rare event. Lost messages do not cause commands to be executed out-of-order since each command (or batch of commands) contains a consensus instance number, which allows a server to detect missing commands.

We can estimate the improvements expected from speculative execution. Let δ be the time it takes for a client to send a command to the coordinator and for a server to respond to the client with its results (Steps 1 and 6 in Figure 3). Assume further that Δ_o is the time needed to order the command (i.e., the time difference between the first and the second ip-multicast related to the command) and Δ_e is the time needed to execute the command. Without speculative execution, the response time expected by a client in the absence of contention is $2\delta + \Delta_o + \Delta_e$. With speculative execution, it depends on the values of Δ_o and Δ_e: if $\Delta_o < \Delta_e$ then response time is $2\delta + \Delta_e$; otherwise response time is $2\delta + \Delta_o$. Thus, we can expect an improvement of the order of $\min(\Delta_o, \Delta_e)$.

3.3 State partitioning

As discussed in Section 2.2, a service implemented by means of state-machine replication has limited or no scalability at all, as a consequence of server replicas storing the full service state, and receiving and handling all client commands. To make the system scalable, we must partition the service's state into "sub-states". If the partitioning is *perfect*, that is, all commands access one sub-state or another, but no command accesses two or more sub-states, then the technique can be trivially implemented: It suffices to replicate each partition individually, using different and independent instances of Ring Paxos, and submit client commands to the appropriate partition.

Some services, however, may not allow perfect partitioning. This is the case when a service's state is partitioned into sub-states such that some of the commands access more than one partition. We illustrate this case with an example. Consider a B-Tree service with insert and query commands—Section 4 contains a detailed description of this service. We can partition the B-Tree into sub-trees by assigning to each sub-tree a non-overlapping key interval and replicate each sub-tree using state-machine replication. An insert command is directed to a single replicated sub-tree. A query command that requests a set of keys within a certain range may be addressed to a single sub-tree or to multiple sub-trees, depending on the range and the key intervals assigned to each sub-tree. If the query command addresses multiple sub-trees, then it is divided into "sub-commands", one for each sub-tree; the client builds the final response from the results received from each sub-tree. Such a service, however, cannot be implemented by independent instances of Ring Paxos, as we now explain.

To understand why, consider the execution on the left of Figure 4. Under linearizability, this execution cannot happen since client C_3 sees C_1's insert before C_2's, and C_4 sees first C_2's insert before C_1's. If we partition the B-Tree into two independent sub-trees, however, as in the execution on the right of Figure 4, then clients may observe a non-linearizable behavior. In this execution, C_3's and C_4's Query$(0, 100)$ command is composed

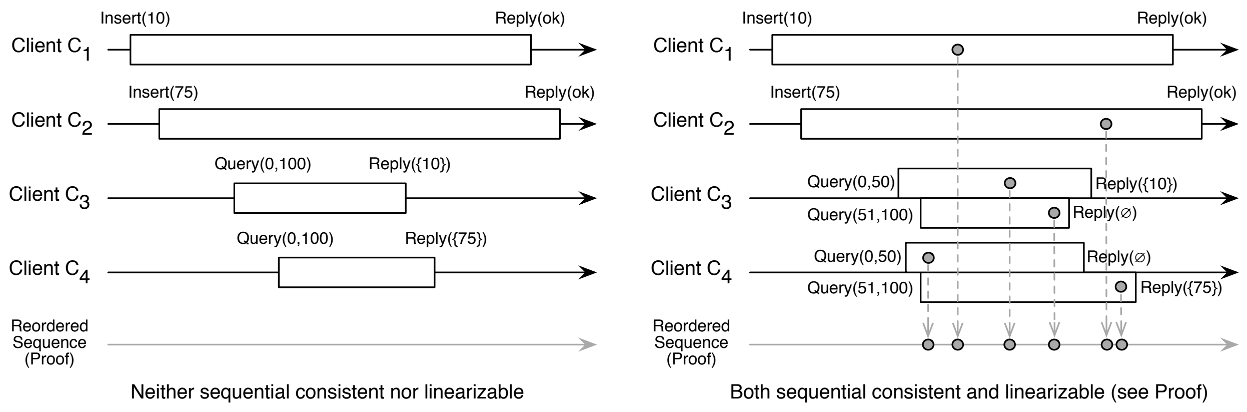


Figure 4: (Left) An non-linearizable execution that cannot happen if a B-Tree is replicated with state-machine replication. (Right) How the same execution can happen if sub-trees of the B-Tree are replicated independently.

of two subcommands, Query$(0, 50)$ and Query$(51, 100)$. The problem is that while C_3's Query$(0, 50)$ succeeds C_4's Query$(0, 50)$ in one partition, C_3's Query$(51, 100)$ precedes C_4's Query$(51, 100)$ in the other partition, and thus, C_3's Query$(0, 100)$ neither precedes nor succeeds C_4's Query$(0, 100)$. To ensure linearizability we must be able to establish a total order on all commands, not only on sub-commands. Notice that this happens in spite of the fact that the execution of each sub-tree is individually linearizable.

We now define *state partitioning ordering*, a guarantee needed to ensure that an execution with commands involving multiple service partitions is linearizable. Let a service state be decomposed into partitions $P_1, ..., P_k$, each one replicated and implemented as a series of consensus executions—the i-th consensus instance decides on the i-th sub-command of partition P_k. Let command C_x be composed of sub-commands $\{c_{x,i} \mid c_{x,i}$ is a subcommand of C_x in $P_i\}$. We define directed graph $G = (V, E)$ such that V contains all commands C_x in the execution and E contains directed edges $C_x \rightarrow C_y$ such that $c_{x,i}$ precedes $c_{y,i}$ in P_i. State partitioning ordering requires that G be acyclic.

A consequence of G being acyclic is that it can be topologically ordered, and therefore for any two commands C_x and C_y, if $c_{x,i}$ precedes $c_{y,i}$ in partition P_i, then in no partition P_j, $c_{y,j}$ precedes $c_{x,j}$, where $c_{x,i}, c_{x,j} \in C_x$ and $c_{y,i}, c_{y,j} \in C_y$. We state the property as an acyclic graph of commands to cover more complex cases involving relations between more than two commands (see [22] for an example).

We have integrated state partitioning order into Ring Paxos as follows. First, there is one ip-multicast address associated with each partition (corresponds to Step 2 in Figure 3) and one ip-multicast address associated with decisions (corresponds to Step 4 in Figure 3). Differently than Ring Paxos, we do not piggyback decision messages with commands. Learners (i.e., servers) listen on the partition addresses they are interested in and on the decision address. Acceptors listen on all addresses. A command contains information about the partitions it accesses. For each partition accessed by the command, the coordinator ip-multicasts one Phase 2A message (with the command) using the address associated with the partition. If a process receives the same message more than once, it simply discards the duplicates. When order is established, the coordinator ip-multicasts the decision message using the decision address. Learners may receive decision messages for partitions they are not interested in, in which case they discard the messages.

To conclude, the state partitioning technique improves the scalability of state-machine replication but it may not be applicable in some cases or it may impose restrictions on how the state of a service can be partitioned. Consider a service whose state contains variables x and y, and a command that modifies x based on the value of y. In this case, the service's state can only be partitioned such that both x and y belong to the same partition. While this constraint limits the number of services that can benefit from state partitioning, we show in the next section that the technique is general enough to allow the implementation of a high performance fault-tolerant B-Tree service.

4 Replicated parallel B-Trees

In this section we illustrate high performance state-machine replication with a B-Tree service. We define the service's interface, used by the clients, and how it was implemented and optimized using speculative execution and state partitioning.

B-Tree service The B-Tree stores $(key, value)$ tuples, where both key and $value$ are 8-byte integers. Clients can submit insert, delete and query commands. An insert command insert(k, val) checks whether an entry with key k already exists in the tree; if not, (k, val) is included in the tree. In any case the command returns an acknowledgement. A delete command delete(k) removes entry with key k, if existent, and returns an acknowledgement. A query command query(min, max) returns all entries (k, val) such that $min \leq k \leq max$.

Fully replicated B-Tree In order to tolerate server failures we replicate the B-Tree service using state-machine replication. Client commands are linearizable and submitted to the servers by means of Ring Paxos. Insert and delete commands are received and executed by all operational servers, but only one server (randomly chosen by the client) responds. A query command is received by all operational servers and executed by a single server, randomly chosen by the client. If a client does not receive the response for a command after some time it resubmits the command.

Speculative execution To reduce the response time experienced by clients we use speculative execution. Since queries do not change the state of the tree, there is no state to be rolled back in case of commands delivered out-of-order. Inserts and deletes are executed against the B-Tree as soon as they are received. To roll back a successful insert(k, val), the server executes a delete(k)—there is nothing to roll back if the insert fails because the key already exists, a fact that is taken into account by the server. A delete(k) keeps the value removed so that it can be rolled back by executing an insert.

State partitioning We divide the state of the B-Tree in partitions such that each partition is responsible for a range of keys (i.e., range partitioning). A command that accesses more than one partition is broken into sub-commands by the client (i.e., by a client replication library) and submitted to each concerned partition. Responses received from multiple partitions are merged at the client. Key ranges are of the same size, but depending on the keys included in and deleted from the B-Tree, partitions may become unbalanced. We do not currently address this problem, but it is part of our ongoing work. We are considering techniques to repartition the key space on-the-fly to keep partions balanced.

5 Performance evaluation

In this section we assess the performance of our replicated B-Tree. We consider executions in the presence of message losses and in the absence of process failures. Process failures are hopefully rare events; message losses happen relatively often because of high network traffic.

5.1 Experimental setup

We ran the experiments in a cluster of Dell SC1435 servers equipped with 2 dual-core AMD-Opteron 2.0 GHz CPUs and 4GB of main memory. The servers are interconnected through an HP ProCurve2900-48G Gigabit switch (0.1 msec of round-trip time). Each experiment (i.e., point in the graph) is obtained over a 60-second run out of which the first and the last 10 seconds are discarded. Clients and servers run in different nodes. Each client runs in a closed loop with a random think time in the range of 0–10 msec.

In all experiments the B-Tree is initialized with 12 million entries. Client commands are messages with 256 bytes; responses are 8 Kbytes for ranges and 256 bytes for inserts and deletes. We consider three workloads: (a) each client command is a query with range of 1000 keys;

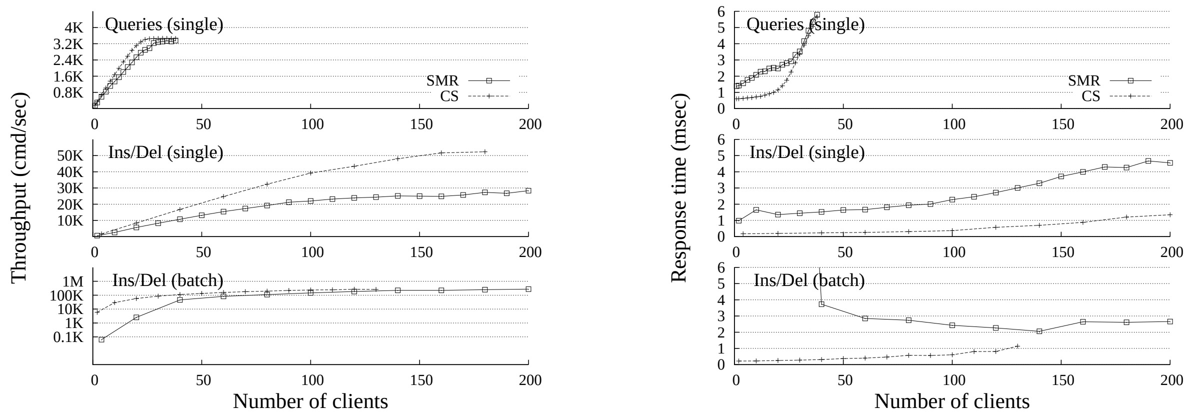


Figure 5: State-machine replication (SMR) versus client-server (CS) under three workloads. (Left) Throughput versus number of clients—notice the different throughput scales, one of which is logarithmic. (Right) Response time versus number of clients.

(b) each client command is an insert or a delete—hereafter we refer to inserts and deletes as *updates*; (c) each client command is composed of seven batched updates (which is what fits in a 256-byte message). Additionally, in workload (c) Ring Paxos batches client messages in bigger packets (8 Kbytes) to improve throughput.

5.2 The cost of replication

Our first set of experiments evaluate the costs of state-machine replication (SMR) with respect to a non-replicated client-server (CS) setup (see Figures 5 and 6). For queries and batched updates, replication does not introduce a cost in throughput. In these cases, the executions are CPU-bound. For single updates, the replicated setting cannot reach the same throughput as a client-server configuration because the execution of the former is limited by the maximum number of instances per second that can be run by Ring Paxos. In all cases, however, replication imposes a cost in response time, as shown by the graphs in right column of Figure 5. Response time for few clients with batched updates in the replicated setting is high because with low load Ring Paxos packets are sent due to timeouts; the effect disappears as clients are added and messages are sent as soon as an 8-Kbyte packet is full.

Adding replicas can help improve the throughput of read-only commands, as shown by the left bar on the left graph in Figure 6. For update commands, no improvement in throughput is possible since all replicas must be involved in the operations, even if only to received the commands in the right order, as discussed in Section 2.2. Figure 6 also shows the corresponding response times, with the highest values for all replicated experiments.

5.3 Speculative execution

We report our assessment of speculative execution for configurations with 1, 2, 4 and 8 servers using the queries and the batched updates workloads. (see Figures 7, 8, 9 and 10). In all scenarios speculation reduces response time with respect to state-machine replication, although the results are more visible with batched updates. By reducing response time, the technique also proportionally improves throughput, a direct consequence of Little's law [13].

In our implementation the speculative server is composed of four active threads, whose tasks can be described as follows: 1) delivering the commands and their order decided by Ring Paxos, 2) tracking the commands once their order is received, 3) processing the commands, and 4) sending ackknowledgements to the clients after the commands are successfully processed and their order is

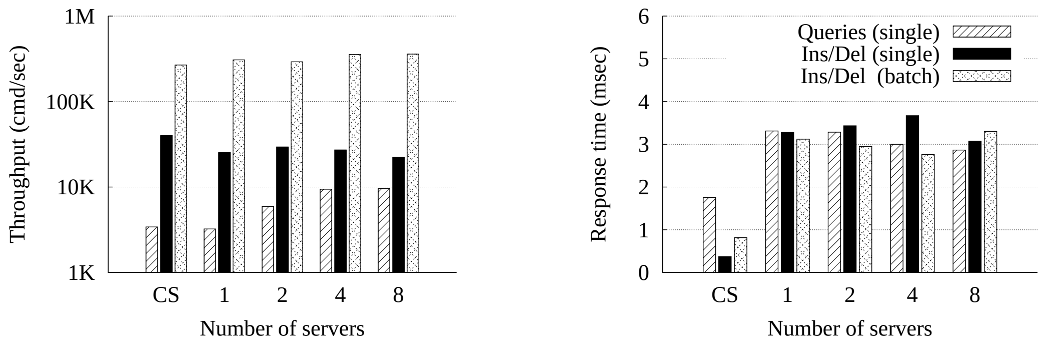


Figure 6: State-machine replication with increasing number of replicas versus client-server. (Left) Maximum throughput versus number of servers (y-axis in log scale). (Right) Response time versus number of servers.

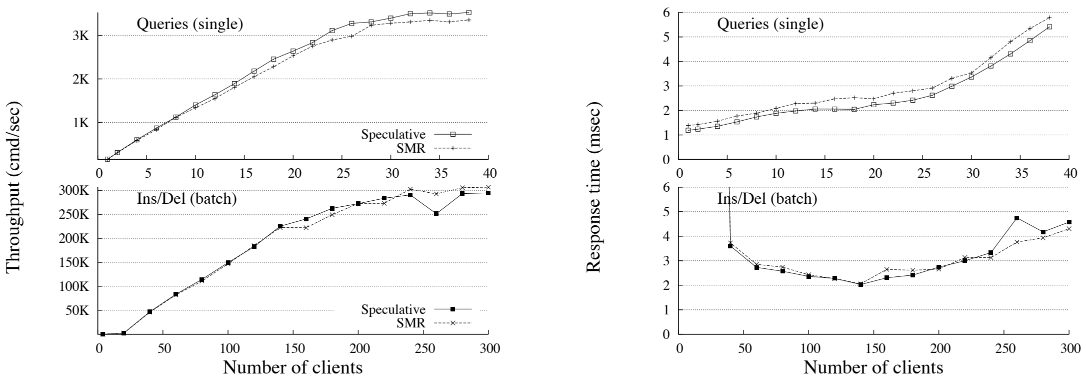


Figure 7: Speculative execution improvement on state-machine replication with 1 replicas. (Left) Throughput improvement versus number of servers. (Right) Resp. time improvement versus number of servers.

known. Thus once the first thread receives a command, inserts it in a shared buffer from which the third thread will later pick it up and process it. This implies that each command spends some time waiting in the buffer until the third thread is free to process it. Therefore in practice due to the implementation overheads a request is not processed immediately after its arrival. Assume Δ_b is the time wasted due to this overhead.

By recalling from Section 3.2 whereas Δ_e refers to the net time required to process a command the total time needed to process a command is Δ_e + Δ_b. Accordingly the expected response time improvement in practice is min$(\Delta_o, \Delta_e +, \Delta_b)$ in contrast to that of theoretical which is min(Δ_o, Δ_e).

To assess our simple analytical model we consider a configuration with one replica executing queries only.(see Figure 11)The graph on the left of the figure shows results for state-machine replication with and without speculation and the client-server setup. The area between speculative and SMR curves in this graph is depicted by Response time improvement in the graph on the right. This improvement is less than the min$(\Delta_o, \Delta_e +, \Delta_b)$ which means one could come up with a better implementation to even improve the response time further.

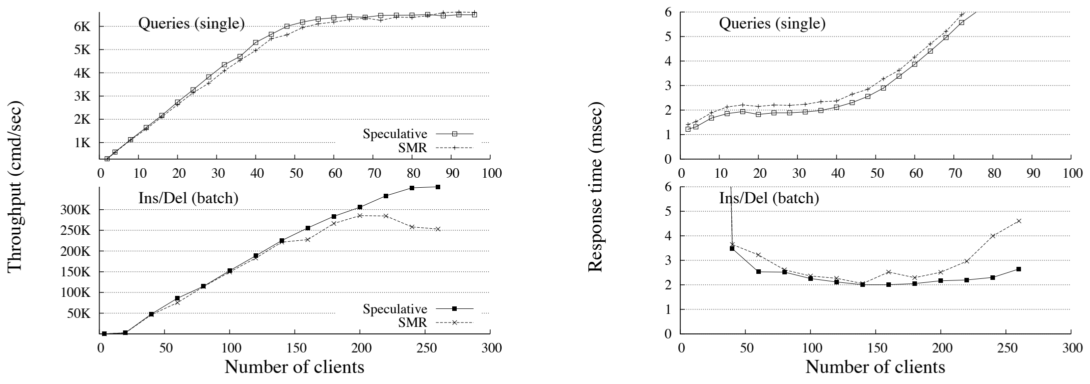


Figure 8: Speculative execution improvement on state-machine replication with 2 replicas. (Left) Throughput improvement versus number of servers. (Right) Resp. time improvement versus number of servers.

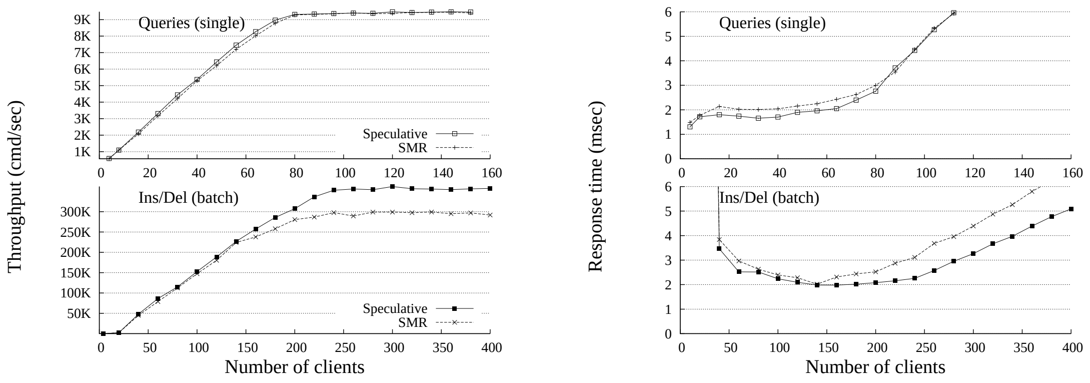


Figure 9: Speculative execution improvement on state-machine replication with 4 replicas. (Left) Throughput improvement versus number of servers. (Right) Resp. time improvement versus number of servers.

5.4 State partitioning

To assess the state partitioning strategy, we consider two configurations, one with the B-Tree state divided into two partitions and the other with the B-Tree state divided into four partitions (labels "2 P" and "4 P" in Figure 12, respectively); in both configurations each partition has two replicas. In executions with cross-partition query commands, a cross-partition query accesses two partions, regardless the number of existing partitions.

The graph on the left of Figure 12 shows that for queries, the throughput increases by a factor of 2.1 from SMR to two partitions, and by a factor of nearly four from SMR to four partitions. The improvement on batched updates is not as remarkable as on queries, although the system throughput increases by factors of 1.8 and 2.6 for two and four partitions, respectively. The graph on the right of the figure shows that such an increase in throughput does not incur in significant changes in response time with respect to SMR. Although these experiments were run using no cross-partition queries, as we show next, this is not the most favorable setup for state partitioning.

Figure 13 considers the effects of cross-partition queries in the state partitioning technique with two partitions in an execution with query commands whereas there are 2 replicas in each partition. The graphs show that for lower load (i.e., 100 clients) there is almost no difference in throughput and response time between different configurations. For higher loads, configurations with 50% and

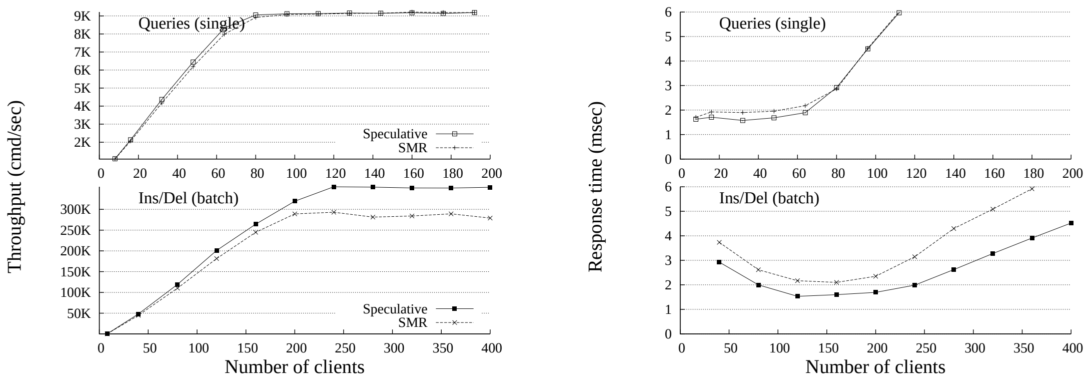


Figure 10: Speculative execution improvement on state-machine replication with 8 replicas. (Left) Throughput improvement versus number of servers. (Right) Resp. time improvement versus number of servers.

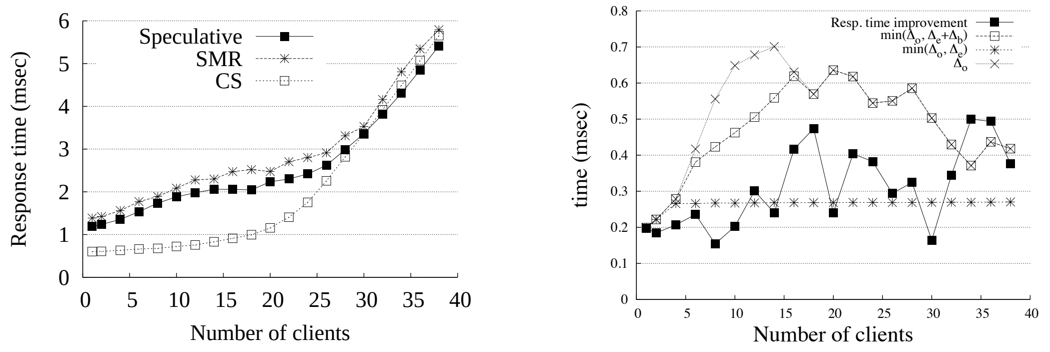


Figure 11: Speculative execution versus state-machine replication (SMR) versus client-server (CS). (Left) Response time versus number of clients. (Right) Distribution of time at the server in speculative execution versus number of clients.

75% of cross-partition queries reach higher throughputs. In fact, the lowest throughput and highest response time is obtained with a configuration without cross-partition queries. To understand why, we must look at how CPU is used in a server.

Each non-speculative server is implemented by three threads, one that receives commands, one that executes them, and one that responds to clients. Each thread is assigned a different processor. The left graph in figure 15 shows the CPU usage for threads responsible for execution and responses; the thread that receives commands has low use. While in configurations with no cross-partition queries, 98% of the processor for command execution is used, in configurations with 25% and 100% of cross-partition queries, the processor for command execution and response is 95% used. Finally, in configurations with 50% and 75% of cross-partition queries, the processors are used less than 90%. The 50% configuration has slightly higher throughput than the 75% configuration because it uses less bandwidth.

The reason for the execution processor use to decrease with the increase in the number of cross-partition queries is that a cross-partition query is “cheaper” to execute than a single-partition query since it processes fewer elements in the B-Tree. However, the response processor use increases with the number of cross-partition queries because

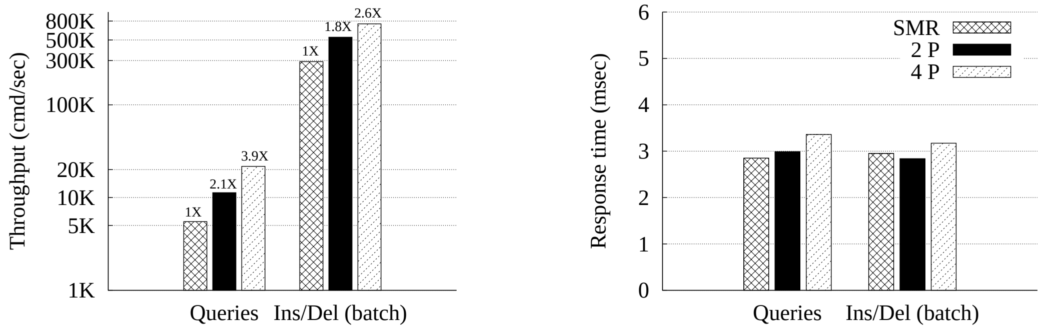


Figure 12: State partitioning (2 and 4 partitions) versus state-machine replication for queries and batched updates with no cross-partition commands. (Left) Throughput (y-axis in log scale) with improvement over SMR versus command type. (Right) Response time versus command type.

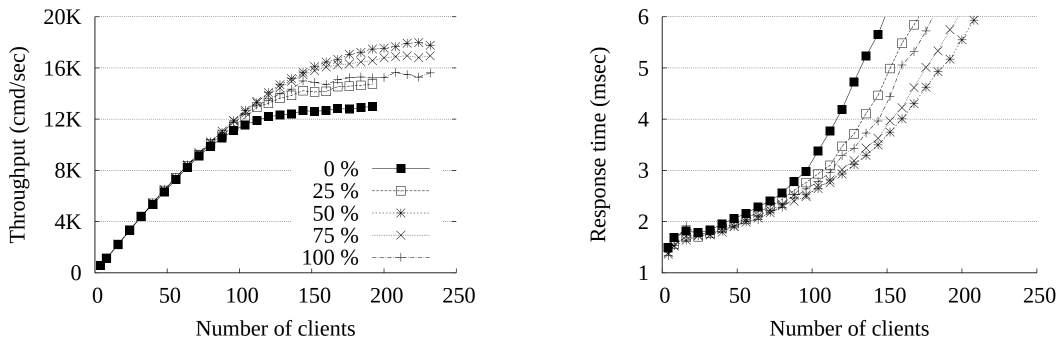
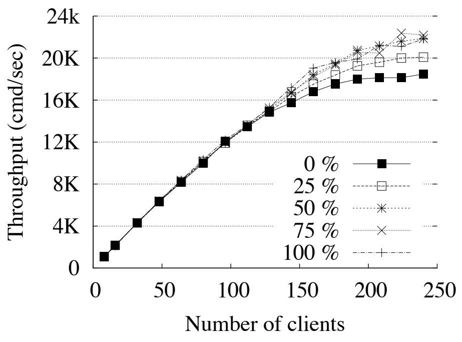


Figure 13: Effects of cross-partition queries in state partitioning with 2 replicas in each partition. (Left) Throughput versus number of clients. (Right) Response time versus number of clients.

a cross-partition query is split into two queries (and thus there are more queries) and servers respond to queries with fixed-size messages, regardless the amount of information contained in the message.

in Figure 16 the graph on left shows the outgoing bandwidth per server for the cross-partition queries. As expected by increasing the percentage of cross-partition queries the outgoing bandwidth for each server increases. However it seems that for 75% and 100% cases the bandwidth is not scaling as expected. To avoid the server's outgoing bandwidth as a bottleneck one can keep adding more replicas to each partition. The effect of 3 replicas in each partition is investigated in figure 14 where the maximum achievable throughput for all the cases is increased compared to the 2 replica case(see figure 13). The right graph in figure 15 depicts the CPU usage for threads responsible for execution and responses. As the cross-partition queries increases the responding thread consumes more CPU and the executing thread 's CPU usage decreases. Moreover the outgoing bandwidth per server is shown in the right graph of figure 16 whereas compared to the graph on the left the bandwidth is no more a bottleneck.

Our final set of experiments considers the combined effects of speculative execution and state partitioning. Figure 17 shows the relative improvements of the specula-



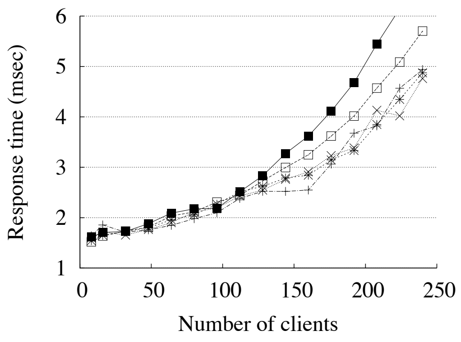
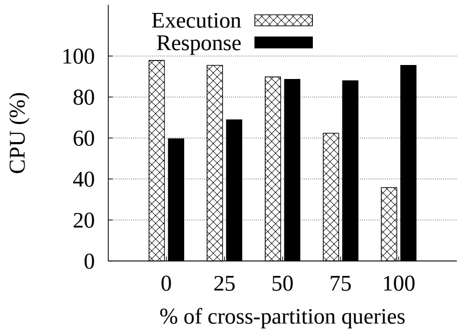


Figure 14: Effects of cross-partition queries in state partitioning with 3 replicas in each partition. (Left) Throughput versus number of clients. (Right) Response time versus number of clients.



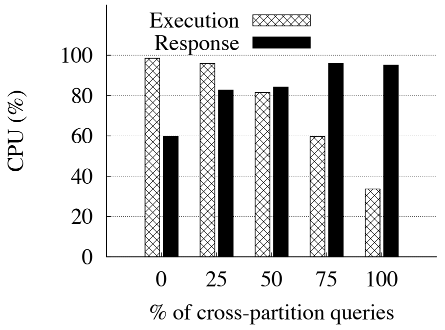


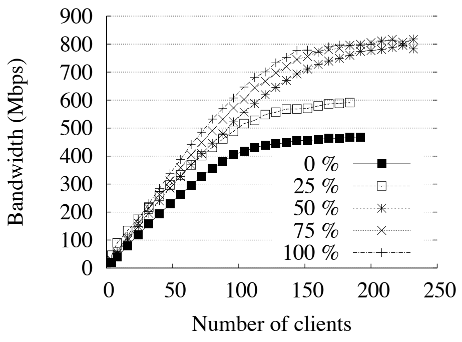
Figure 15: CPU utilization for the experiments in Figure 13 and 14. (Left) 2 replicas in each partition. (Right) 3 replicas in each partition.

tive execution technique over state-machine replication with state partitioning for different percentages of cross-partition queries. In all configurations the technique is effective in that it decreases response time, with minor improvements in throughput. The reason for the improvement to decrease with the number of cross-parition queries is that the execution time in a server of a cross-partition query is smaller than the execution time of a single-partition query, as explained above. Therefore, the window of opportunity for speculative execution is narrower (cf. last paragraph in Section 3.2).

6 Related work

State-machine replication is a well established replication technique, which has been extensively discussed in the literature. In the following we focus on work related to the two optimizations we presented, speculative execution and state partitioning, and to parallel B-Trees.

Optimistic or speculative execution has been suggested before as a mechanism to reduce the latency of agreement problems. For example, in [17, 29] clients are included in the execution of the protocol to reduce the latency of Byzantine fault-tolerant agreement. In [14, 16] the authors introduce atomic broadcast with optimistic delivery in the context of replicated databases. The motivation is



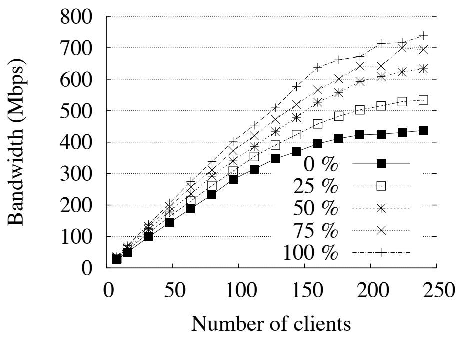
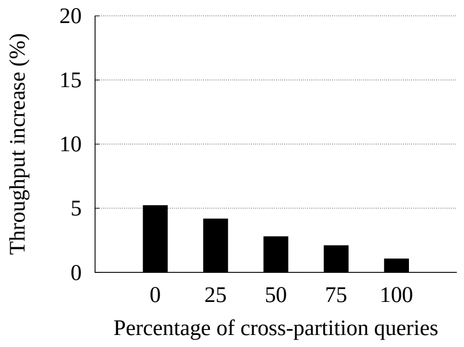


Figure 16: Outgoing bandwidth per server in cross-partition queries in state partitioning. (Left) 2 replicas in each partition. (Right) 3 replicas in each partition.



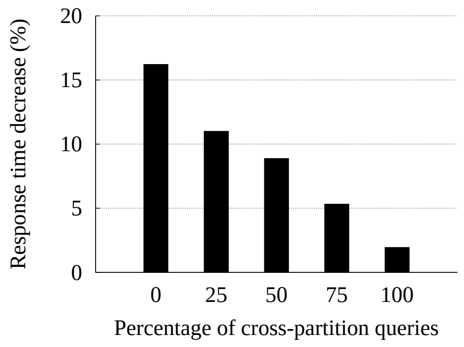


Figure 17: Improvements of combined speculative execution and state partition over state-machine replication. (Left) Throughput increase versus percentage of cross-partition queries. (Right) Response time decrease versus percentage of cross-partition queries.

similar to ours: overlapping the execution of transactions or commands with the ordering protocol. Optimistic delivery relies on spontaneous ordering of messages, typical in local-area networks. The property holds in the absence of contention. If too many commands are submitted simultaneously, then out-of-order deliveries can happen more frequently and the technique becomes less interesting. Ring Paxos can use speculative execution under high contention as it does not depend on spontaneous message ordering.

Partitioning the state of a replicated service is conceptually similar to partial replication of databases [22]. Partial database replication addresses scalability issues identified in fully replicated databases. Several partial database replication protocols have been proposed, some optimized for local-area networks (e.g., [5, 6, 9, 23]) and some topology-agnostic (e.g., [12, 24, 26, 27]). Partitioning the state of a replicated service differs from partially replicating a database with respect to the granularity of the data and the consistency criterion. Databases are usually organized as collections of data items. Partitioning such a state is simpler than partitioning the state of a service, which may not have been designed with partitioning as a goal. With respect to consistency, the two main consistency criteria used in replicated databases are one-copy serializability [4] and a generalized form of snapshot isolation [10, 20]. These criteria do not take real-time dependencies between operations into account and therefore admit more efficient implementations than linearizability. To a certain extent, making a partially replicated database scale is "easier" than scaling a linearizable replicated service.

Ring Paxos equipped to implement the state partitioning technique resembles an atomic multicast protocol [7]. In fact, our state partitioning ordering is inspired by the acyclic order property of atomic multicast [22]. To the best of our knowledge, however, no previous work has explored multicast communication in the Paxos family of protocols, and no speculative or optimistic multicast protocol has been proposed so far.

The closest work to our B-Tree service is [1], where the authors implement and evaluate a distributed B+Tree build on top of Sinfonia [2]. Sinfonia is a distributed, fault-tolerant storage engine that offers a low-level address space in which application processes can store their data. Sinfonia offers a minitransaction interface to its clients. Minitransactions are short-lived operations similar to a generalized compare-and-swap operation. The authors exploit the flexibility offered by Sinfonia to implement a scalable B+Tree. As an optimization, inner nodes are replicated on all Sinfonia client nodes. On the one hand this allows nodes to traverse a tree locally, without contacting any other node; on the other hand, all nodes must be involved in the update of inner nodes. Sinfonia relies on stronger system assumptions than the ones assumed in this paper. This is due to the use a two-phase commit protocol to terminate minitransactions.

7 Conclusions

This paper revisits state-machine replication from a performance perspective. State-machine replication is a well-known approach to rendering services fault tolerant. The idea is to fully replicate the service state on several servers and execute every client command in every nonfaulty server in the same order. Although some optimizations for performance are possible, inherently the technique introduces an overhead in service response time and is limited by the throughput of a single server. To mitigate these drawbacks, we have considered speculative execution and state partitioning.

Our experiments with speculative execution show that while the technique can reduce the response time of a replicated service, the improvement is limited in that the resulting service's response time remains quite larger than the response time of a client-server setup. One question for further investigation is whether there are other ways to exploit speculation to reduce response time further. For example, currently, a server waits until the order in which a command was executed is confirmed to reply to the client. Servers could respond to a client immediately after a command is executed, even if its order confirmation has not been received, and notify the client later with a short message once order is established. This mechanism would overlap both the command execution and its response with the ordering protocol.

State partitioning has shown remarkable results. In some cases, the throughput of a service improved by a factor or nearly four after partitioning its state. Experiments have also shown that the two techniques can be combined with improvements on both throughput and response time. Our plans for the future are to investigate the generality of the state partitioning technique and better characterize the space in which it can be used. We also intend to investigate mechanisms to guarantee (quasi)-balanced B-Trees.

References

[1] M. K. Aguilera, W. M. Golab, and M. A. Shah. A practical scalable distributed B-tree. *PVLDB*, 1(1):598–609, 2008.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, 2007.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high performance database cluster. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (11th ICPADS'05)*, volume 1, pages 809–815, Fuduoka, Japan, July 2005. IEEE Computer Society.

[6] A. L. P. F. de Sousa, R. C. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *NCA*, pages 298–309. IEEE Computer Society, 2001.

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

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

[9] E.Cecchet, J.Marguerite, and W.Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proc. of USENIX Annual Technical Conference, Freenix track*, 2004.

[10] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Symposium on Reliable Distributed Systems (SRDS'2005)*, Orlando, USA, 2005.

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

[12] U. Fritzke and P. Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 284–, Washington, DC, USA, 2001. IEEE Computer Society.

[13] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, Inc., New York, 1991.

[14] R. Jiménez-Peris, M. Patiño Martínez, K. B., and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 477–, Washington, DC, USA, 2002.

[15] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003.

[16] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, Austin (USA), 1999.

[17] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

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

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

[20] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. E. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2), 2009.

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

[22] N. Schiper. *On Multicast Primitives in Large Networks and Partial Replication Protocols*. PhD thesis, University of Lugano, 2009.

[23] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings*, volume 4305 of *Lecture Notes in Computer Science*, pages 81–93. Springer, 2006.

[24] N. Schiper, P. Sutra, and F.Pedone. P-store: Genuine partial replication in wide area networks. In *Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2010.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[26] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*, pages 290–297. IEEE Computer Society, 2007.

[27] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. An autonomic approach for replication of internet-based services. In *Symposium on Reliable Distributed Systems (SRDS'2008)*, pages 127–136. IEEE, 2008.

[28] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[29] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 245–260, Berkeley, CA, USA, 2009. USENIX Association.

Appendix

In the following we argue that our B-Tree algorithm is linearizable. Notice that speculative delivery alone does not change the correctness of the original state-machine replication algorithm: every command is propagated to every replica and executed in their final order, just like state-machine replication [3]. What we must show is that state partitioning is also linearizable.

Assume a B-Tree whose state is divided into partitions $\Pi = \{P_1, P_2, ...\}$. A command C is composed of one or more sub-commands $C(k)$, one for each partition P_k it addresses. In particular, C can insert, delete or query items in the B-Tree. Each B-Tree partition is replicated and implemented as a series of consensus executions such that the i-th consensus instance decides on the i-th sub-command of partition P_k. Sub-commands in a partition are executed in the order in which they are decided, that is, the i-th sub-command only starts after the $(i-1)$-th sub-command has finished.

We recall that $G = (V, E)$ is a directed graph where V contains all commands C_x in the execution and E contains a directed edge $C_x \rightarrow C_y$ iff a sub-command of C_x is executed before a sub-command of C_y in some partition P_k. State partitioning ordering states that G is acyclic.

In order to show that any execution of the B-Tree implemented using state-machine replication and state partitioning is linearizable, we must show that there is a way to reorder the commands in a sequence S such that (i) S respects the order of non-overlapping commands across all clients, and (ii) S respects the semantics of the commands, as defined in their sequential specifications.

We initially show that there is a sequence S that respects the order of non-overlapping commands across all clients. To do so, we consider two conditions: (a) If C_x precedes C_y in G, then C_x precedes C_y in S. (b) If C_x finishes before C_y starts (i.e., they are non-overlappping), then we order C_x before C_y in S. We claim that conditions (a) and (b) can always be accommodated. To see why, assume for a contradiction that C_x precedes C_y in G and C_y finishes before C_x starts. From the fact that C_x precedes C_y in G, both C_x and C_y access some partition P_k and $C_x(k)$ is executed before $C_y(k)$ at P_k. Thus, $C_x(k)$ is delivered before $C_y(k)$, and it follows that C_y cannot finish before C_x starts.

We now show that sequence S respects the semantics of B-Tree commands. We must show that any command in S takes into account all commands that precede it, and in the order in which they appear in S. Let C_x be a command in S. For every sub-command $C_x(k)$ of C_x, only commands on P_k can affect $C_x(k)$, thus, we can focus on sub-commands $C_y(k)$ only, that is, sub-commands of some command C_y on the same partition P_k. Since C_x and C_y are composed of sub-commands on a common partition, from the definition of G and the fact that it is acyclic, we can totally order them. Thus, sub-commands $C_x(k)$ and $C_y(k)$ will be executed on partition P_k according to the order of C_x and C_y in S. Moreover, from the implementation of each partition, a sub-command in a partition is only executed after the sub-command that precedes it is completed. Thus, sub-commands take into account their preceding sub-commands, in the order they are executed.