The Dissertation Committee for Emmanouil Kapritsos
certifies that this is the approved version of the following dissertation:

# Replicating Multithreaded Services

Committee:

Lorenzo Alvisi, Supervisor

Mike Dahlin

Rodrigo Rodrigues

Robbert van Renesse

Emmett Witchel

# Replicating Multithreaded Services

by

**Emmanouil Kapritsos, B.S.; M.Sc.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

**THE UNIVERSITY OF TEXAS AT AUSTIN**

December 2014

Dedicated to my parents

for their unwavering support toward my academic path.

# Acknowledgments

*"As you set out for Ithaca,*
*pray that the road is long,*
*full of adventure, full of knowledge."*

Cavafy could very well be speaking of the pursuit of a Ph.D. degree, a road as full of knowledge as it is of adventure. I was fortunate not to travel this road alone. Several people along the way made the journey enjoyable, helped me overcome the many challenges I met, and graced me with their knowledge, experience and wisdom.

I consider myself lucky that my advisor, Lorenzo Alvisi, was among those people. Lorenzo went beyond the typical role of an academic advisor, to that of a friend, helping me whenever I needed it and offering guidance through several important choices, both academic and personal. Lorenzo showed me, little by little, the path to a wider world: that of scientific reasoning. His attention to detail taught me the importance of clear and precise thinking, which I consider the most valuable lesson I learned during my Ph.D.

My fellow graduate students were a constant source of help, comfort, and knowledge. In particular, I want to thank Yang for being my academic "brother": we worked very closely during all these years and he is one of the

main contributors behind the work in this thesis. Yang and I started with rather different talents on research and computer science; I think we both gained some of the other's perspective along the way. I also want to thank Allen for being my academic "older brother". Allen and I started our collaboration with rather violent clashes of disagreement.[1] Through our collaboration, I got to appreciate Allen; at first as one of the smartest people I have met, and later on as a friend. These days our clashes are less frequent, less violent and, more importantly, constructive. Finally, I want to thank Sangmin, Ed, Prince, Mirco, Harry, Nalini, Indrajit, Sebastian, Josh, Srinath, Trinabh, Chao, Chunzhi, Apurv, Eric, Mikie, Alan, and Mike for their companionship and support.

My committee members helped greatly in improving the quality of this thesis, by providing guidance and detailed feedback. I would like to thank Mike Dahlin, in particular, with whom I collaborated closely throughout these years. Mike's intuition and knowledge in systems building is remarkable, and I learned much from our collaboration. I want to also thank Peter Triantafillou for his support and guidance, and for equipping me with the tools required to pursue a Ph.D. degree.

Outside the immediate academic environment, however, four people are mainly responsible for putting me in the position to finish this thesis—indeed, to even begin writing it. My parents are the ones that first put me on the academic path. Their example and constant encouragement has always been the motivating force behind all my academic accomplishments; this thesis is dedicated to them. Later, during my undergrad years, my friend and classmate Constantinos inspired me, by his own example, to aim at nothing less than

[1] ...made even more frustrating by the fact that he was usually right.

the highest I could achieve. It is also he who first planted in my head the idea of pursuing a Ph.D. in the United States. Last, but certainly not least, my wife Nafsika made this long journey possible, by employing an extraordinary amount of patience and understanding. Like Penelope waiting for Odysseus, Nafsika endured my seven-year absence from her life with grace. She supported me through the toughest parts of graduate school and was the primary reason for my everyday happiness, despite the 10,000 miles that stood between us.

Emmanouil Kapritsos

*The University of Texas at Austin*
*December 2014*

# Replicating Multithreaded Services

Emmanouil Kapritsos, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Lorenzo Alvisi

For the last 40 years, the systems community has invested a lot of effort in designing techniques for building fault tolerant distributed systems and services. This effort has produced a massive list of results: the literature describes how to design replication protocols that tolerate a wide range of failures (from simple crashes to malicious "Byzantine" failures) in a wide range of settings (e.g. synchronous or asynchronous communication, with or without stable storage), optimizing various metrics (e.g. number of messages, latency, throughput).

These techniques have their roots in ideas, such as the abstraction of State Machine Replication and the Paxos protocol, that were conceived when computing was very different than it is today: computers had a single core; all processing was done using a single thread of control, handling requests sequentially; and a collection of 20 nodes was considered a large distributed system.

In the last decade, however, computing has gone through some major paradigm shifts, with the advent of multicore architectures and large cloud infrastructures. This dissertation explains how these profound changes impact the practical usefulness of traditional fault tolerant techniques and proposes new ways to architect these solutions to fit the new paradigms.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

For the last 40 years, the systems community has invested a lot of effort in designing techniques for building fault tolerant distributed systems and services. This effort has produced a massive list of results: the literature describes how to design replication protocols that tolerate a wide range of failures (from simple crashes to malicious "Byzantine" failures) in a wide range of settings (e.g. synchronous or asynchronous communication, with or without stable storage), optimizing various metrics (e.g. number of messages, latency, throughput) [1, 11, 14, 16, 18, 21, 43, 46–48, 65, 78].

These techniques have their roots in ideas, such as the abstraction of State Machine Replication and the Paxos protocol, that were conceived when computing was very different than it is today: computers had a single core; all processing was done using a single thread of control, handling requests sequentially; and a collection of 20 nodes was considered a large distributed system.

In the last decade, however, computing has gone through some major paradigm shifts, with the advent of multicore architectures and large cloud infrastructures. This dissertation explains how these profound changes impact the practical usefulness of traditional fault tolerant techniques and proposes new ways to architect these solutions to fit the new paradigms.

**Paradigm shift 1: Multicore** With the abrupt halt in the increase of processor speeds, parallel execution is the only way to achieve high performance. The advent of multicore computers has revolutionized the way we process data; not sequentially any more, but rather in parallel. These days multicore computers and multithreaded processing are used in the overwhelming majority of computers, making single-threaded execution something of the past.

Multithreaded execution, while ubiquitous, is unfortunately not supported by most fault tolerance techniques. A popular approach to designing fault tolerant services is State Machine Replication (SMR). At the core of the SMR approach is the idea of having a number of replicas deterministically process the same sequence of requests so that correct replicas traverse the same sequence of internal states and produce the same sequence of outputs. Multithreaded execution poses a challenge to this approach. If different replicas interleave requests' instructions in different ways, the states and outputs of correct replicas may diverge even if no faults occur. As a result, today's SMR systems require servers to process requests sequentially: a replica finishes executing one request before beginning to execute the next. Of course, this approach cannot leverage the performance benefits of parallel execution, which makes it unacceptably inefficient.

**Paradigm shift 2: Large cloud infrastructures** Traditionally, distributed systems were conceived as standalone client-server pairs. The emergence of large cloud infrastructures, however, has significantly changed the way distributed systems are designed and built. Many cloud applications consist of multiple services that interact among themselves. When a client sends a request to a service $A$ (e.g. a web server), that service may need to issue a nested request to another service $B$ (e.g. a database), in order to process the client's request.

These interactions complicate our fault tolerance techniques considerably. For example, if service $A$ is replicated using a protocol that employs speculative execution [40, 43], where request execution may be rolled back,

then sending nested requests to another service $B$ means exposing $B$ to a speculative state of $A$. If speculation fails at $A$, rolling back the state of $A$ is no longer enough; one must somehow roll back the nested request to service $B$, along with its effects. Even worse, if those effects have already been observed by a client of $B$, there is no way to repair this inconsistency.

Furthermore, even when service $A$ does not employ speculative execution, but instead uses sequential execution, things are not ideal. If service $A$ employs sequential execution, issuing nested requests to other services can lead to significant performance degradation—even below the already low standards of sequential execution—since sequential execution forces service $A$ to remain idle while nested requests to other services are in flight.

In this thesis we rethink the architecture and protocols of replicated services to accommodate multithreaded execution and interaction between multiple services. In particular we make the following contributions:

**Rethink the replication architecture to support multithreading.** We show that the current reliance on sequential execution arises from the agree-execute architecture that is adopted by traditional replication systems. This architecture requires that replicas reach agreement on the order of requests and then execute requests in that order; a requirement that is at odds with the paradigm of multithreaded execution, where requests are not executed in any particular order. To address this problem, we submit that a radical architectural change is required. In Chapter 3 we propose a new *execute-verify* architecture, where replicas first speculatively execute requests in parallel, and without having agreed on their order; and then proceed to reach agreement on whether enough replicas have reached the same state and produced the same responses. Such speculation may not always succeed, however, since different thread interleavings could cause different replicas to diverge. Chapter 3 describes how we can make this new architecture efficient in practice. In particular we describe how to efficiently (a) minimize the probability of divergence, (b) detect whether a divergence has occurred, and (c) repair a divergence

when it occurs. The resulting system, Eve, achieves two properties that prior replica coordination protocols have treated as fundamentally at odds with each other: *nondeterministic interleaving of requests* and *execution independence*. Nondeterministic interleaving of requests allows Eve to achieve high-performance replication for multi-core servers. For example, in our experiments with the TPC-W benchmark, Eve achieves a 6.5x speedup over sequential execution that approaches the 7.5x speedup achieved by the original unreplicated server. Execution independence allows Eve to mask a wide range of faults, including Byzantine faults. Notably, we find that execution independence pays dividends even when Eve is configured to tolerate only crash or omission failures by offering the opportunity to mask some concurrency failures.

**Refine the protocols to accommodate service interactions.** Sequential execution and speculation, mechanisms widely used in replicated protocols, have significant shortcomings in terms of both correctness and performance when we move away from the simple client-server model to an environment where services interact. In Chapter 4 we describe Adam, a novel replication protocol that addresses these shortcomings. To address the correctness violations that can be triggered by speculation, we propose a novel technique that allows a service to employ speculative execution without exposing it to other services. In practical terms, this technique makes it safe to use multithreaded execution when replicated services interact with other services. To mitigate the performance degradation caused by sequential execution in such settings, we observe that replica convergence does not require sequential execution: any deterministic request schedule can achieve that goal. We propose a new pipelined execution scheme where replicas do not remain idle waiting for nested requests, but instead rotate execution among the available requests according to a deterministic schedule. Despite its simplicity, we show that this approach can yield significant performance benefits.

# Chapter 2

# State Machine Replication

State Machine Replication (SMR) is a powerful technique for implementing fault tolerant services. The main idea of SMR is to have a number of replicas deterministically process the same requests so that correct replicas traverse the same sequence of internal states and produce the same sequence of outputs. In this chapter, we describe the abstraction of a replicated state machine, as well as the way SMR has been implemented for the past 40 years.

## 2.1 The Replicated State Machine abstraction

A *state machine* consists of *state variables*, which encode its state, and *commands*, which transform its state. A client of the state machine makes a *request* to execute a command. The request names a state machine, names the command to be performed, and contains any information needed by the command. Processing a client request causes the state machine to produce an output and transition to a new state; or, in the case of read-only commands, to transition back to the same state. In a deterministic state machine the output and state transition caused by a command are determined only by the current state and the command.

To implement a fault tolerant state machine, one must provide the illusion of a state machine that retains both safety and liveness—collectively

called correctness—despite a (configurable) number of faults. This can be achieved by replicating that state machine, running each replica on a separate physical machine. The number of replicas required is, of course, a function of the number and type of failures that the replicated state machine should tolerate. Non-faulty replicas must start from the same initial state. The correctness of a replicated state machine consists of the following safety and liveness requirements:

**Safety** All correct state machine replicas should produce the same observable states and outputs. These states and outputs must be consistent with those produced by a single correct state machine.

**Liveness** When a client issues a request to the replicated state machine, it eventually receives a response.

The safety requirement of a Replicated State Machine, as stated above, is quite simple. It only requires that correct replicas should produce observable states and outputs that could have been produced by a single correct server, and that replicas do not diverge; convergence is crucial in maintaining the illusion of a single correct state machine.

## 2.2 Implementation

To achieve convergence, SMR implementations of the past 40 years have made a simplifying assumption: state machines process requests one at a time. This assumption of sequential execution was in fact so tightly bound with achieving replica convergence, that it appeared as part of the specification of replicated state machines, as it was described in Schneider's tutorial [65]. In this thesis, we aim to decouple the way in which requests are executed—which we view as an implementation choice—from the actual specification that we presented above.

The assumption of sequential execution allows servers to be modeled as deterministic state machines, greatly simplifying the implementation of SMR:

all that is required is that replicas agree on the order in which requests should be executed. Once an order has been agreed upon, replicas simply execute requests in that order. Since state machines are deterministic, correct replicas are guaranteed to traverse the same sequence of state transformations and produce the same sequence of responses. This *agree-execute* approach has been adopted by a large number of replication protocols [1, 14, 18, 21, 43, 46], from the original Paxos protocol [46] to more recent and advanced protocols like Zyzzyva [43]. In 2003, Yin et al. [78] described how replication protocols can be separated into an agreement and an execution phase, crystallizing the picture of this popular *agree-execute* architecture.

For all its simplicity, the underlying assumption of the agree-execute architecture has an unfortunate consequence: replicas must execute requests in a sequential order. In other words, parallel execution of requests is not easily supported by this architecture, since parallel execution can cause different replicas to execute requests' instructions in different orders, causing divergence. This drawback has become significantly more pronounced in the last ten years, during which multicore computers and multithreaded execution have become ubiquitous. Unfortunately, the current agree-execute architecture forces system designers into an undesirable choice: either give up the performance of multithreaded execution, or give up the robustness of fault-tolerant replication. Chapter 3 addresses this dilemma: we show that State Machine Replication is not, in fact, incompatible with multithreaded execution. Based on our refinement of the SMR specification, we propose a new replication architecture that preserves both safety and liveness, while allowing requests to be executed in parallel.

# Chapter 3

# Execute-Verify Replication

In this chapter, we present Eve, a novel replication architecture that aims to reconcile replication with the ability to execute requests in parallel. Our first step towards this goal was presented in Chapter 2: refining the specification of state machine replication, no longer requiring that requests are executed sequentially. Instead, Eve partitions requests in batches and, after taking lightweight measures to make conflicts within a batch unlikely, it allows different replicas to execute requests within each batch in parallel, speculating that the result of these parallel executions (i.e. the system's important state and output at each replica) will match across enough replicas.

To execute requests in parallel without violating the safety requirements of replica coordination, Eve turns on its head the established architecture of state machine replication. Traditionally, deterministic replicas first *agree* on the order in which requests are to be executed and then *execute* them [14, 45, 46, 52, 66, 78]; in Eve, replicas first speculatively *execute* requests concurrently, and then *verify* that they have agreed on the state and the output produced by a correct replica. If too many replicas diverge so that a correct state/output cannot be identified, Eve guarantees safety and liveness by rolling back and sequentially and deterministically re-executing the requests.

Critical to Eve's performance are mechanisms that ensure that, despite the nondeterminism introduced by allowing parallel execution, repli-

cas seldom diverge, and that, if they do, divergence is efficiently detected and reconciled. Eve minimizes divergence through a *mixer* stage that applies application-specific criteria to produce groups of requests that are unlikely to interfere, and it makes repair efficient through incremental state transfer and fine-grained rollbacks. Note that if the underlying program is correct under unreplicated parallel execution, then delaying agreement until after execution and, when necessary, falling back to sequential re-execution guarantees that replication remains safe and live even if the mixer allows interfering requests in the same group.

Eve's execute-verify architecture is general and applies to both crash tolerant and Byzantine tolerant systems. In particular, when Eve is configured to tolerate crash faults, it also provides significant protection against concurrency bugs, thus addressing a region of the design space that falls short of Byzantine fault tolerance but that strengthens guarantees compared to standard crash tolerance. Eve's robustness stems from two sources. First, Eve's *mixer* reduces the likelihood of triggering latent concurrency bugs by attempting to run only unlikely-to-interfere requests in parallel [44, 61]. Second, its *execute-verify* architecture allows Eve to detect and recover when concurrency causes executions to diverge, regardless of whether the divergence results from a concurrency bug or from distinct correct replicas making different legal choices.

In essence, Eve refines the assumptions that underlie the traditional implementation of state machine replication. In the agree-execute architecture, the safety requirement that correct replicas agree *on the same state and output* is reduced to the problem of guaranteeing that deterministic replicas process identical sequences of commands (i.e. agree *on the same inputs*). Eve continues to require replicas to eventually behave like deterministic state machines for liveness, but it no longer insists that they execute identical sequences of requests in the common case: instead of relying on agreement on inputs, Eve reverts to the actual—and weaker—safety requirement of SMR that replicas agree on state and output.

The practical consequence of this refinement is that in Eve correct repli-

cas enjoy two properties that prior replica coordination protocols have treated as fundamentally at odds with each other: *nondeterministic interleaving of requests* and *execution independence*. Indeed, it is precisely through the combination of these two properties that Eve improves the state of the art for replicating multi-core servers:

1. *Nondeterministic interleaving of requests lets Eve provide high-performance replication for multi-core servers.* Eve gains performance by avoiding the overhead of enforcing determinism. For example, in our experiments with the TPC-W benchmark, Eve achieves a 6.5x speedup over sequential execution. This speedup compares favorably with the 7.5x speedup achieved by the original unreplicated server. For the same benchmark, Eve achieves a 4.7x speedup over the Remus primary-backup system [23] by exploiting its unique ability to allow independent replicas to interleave requests nondeterministically.

2. *Independence lets Eve mask a wide range of faults.* Without independently executing replicas, it is in general impossible to tolerate arbitrary faults. Independence makes Eve's architecture fully general, as our prototype supports tunable fault tolerance [18], retaining traditional state machine replication's ability to be configured to tolerate crash, omission, or Byzantine faults. Notably, we find that execution independence pays dividends even when Eve is configured to tolerate only crash or omission failures by offering the opportunity to mask some concurrency failures. Although we do not claim that our experimental results are general, we find them promising: for the TPC-W benchmark running on the H2 database, executing requests in parallel on an unreplicated server triggered a previously undiagnosed concurrency bug in H2 73 times in a span of 750K requests. Under Eve, our mixer *eliminated* all manifestations of this bug: it classified the requests that caused the bug as conflicting and therefore did not allow them to execute in parallel. Furthermore, when we altered our mixer to occasionally allow conflicting requests to

be parallelized, Eve detected and corrected the effects of this bug 82% of the times it manifested, because Eve's independent execution allowed the bug to manifest (or not) in different ways on different replicas.

## 3.1 System model

The novel architecture for state machine replication that we propose is fully general: Eve can be applied to coordinate the execution of multithreaded replicas in both synchronous and asynchronous systems and can be configured to tolerate failures of any severity, from crashes to Byzantine faults.

In this chapter, we primarily target asynchronous environments where the network can arbitrarily delay, reorder, or lose messages without imperiling safety. For liveness, we require the existence of synchronous intervals during which the network is well-behaved and messages sent between two correct nodes are received and processed with bounded delay. Because synchronous primary-backup with reliable links is a practically interesting configuration [23], we also evaluate Eve in a server-pair configuration that—like primary-backup [12]—relies on timing assumptions for both safety and liveness.

Eve can be configured to produce systems that are *live*, i.e. provide a response to client requests, despite a total of up to $u$ failures, whether of omission or commission, and to ensure that all responses accepted by correct clients are *correct* despite up to $r$ commission failures and any number of omission failures [18]. Commission failures include all failures that are not omission failures. The union of omission and commission failures are Byzantine failures. However, we assume that failures do not break cryptographic primitives; i.e., a faulty node can never produce a correct node's MAC. We denote a message X sent by Y that includes an authenticator (a vector of MACs, one per receiving replica) as $\langle X \rangle_{\vec{\mu}_Y}$.

## 3.2 Protocol overview

Figure 3.1 shows an overview of Eve, whose "execute-verify" design departs from the "agree-execute" approach of traditional SMR [14, 46, 78].

### 3.2.1 Execution stage

Eve divides requests in batches, and lets replicas execute requests within a batch in parallel, without requiring them to agree on the order of request execution within a batch. However, Eve takes steps to make it likely that replicas will produce identical final states and outputs for each batch.

**Batching** Clients send their requests to the current primary execution replica. The primary groups requests into batches, assigns each batch a sequence number, and sends them to all execution replicas. Multiple such batches can be in flight at the same time, but they are processed in order. Along with the requests, the primary sends any data needed to consistently process any nondeterministic requests in the batch (e.g. a seed for `random()` calls or a timestamp for `gettimeofday()` calls [14, 18]). The primary however makes no effort to eliminate the nondeterminism that may arise when multithreaded replicas independently execute their batches.

**Mixing** Each replica runs the same deterministic *mixer* to partition each batch received from the primary into the same ordered sequence of *parallelBatches*—groups of requests that the mixer believes can be executed in parallel with little likelihood that different interleavings will produce diverging results at distinct replicas. For example, if *conflicting* requests $\rho_1$ and $\rho_2$ both modify object $A$, the mixer will place them in different parallelBatches. Section 3.3.1 describes the mixer in more detail.

**Executing (in parallel)** Each replica executes the parallelBatches in the order specified by the deterministic mixer. After executing all parallelBatches

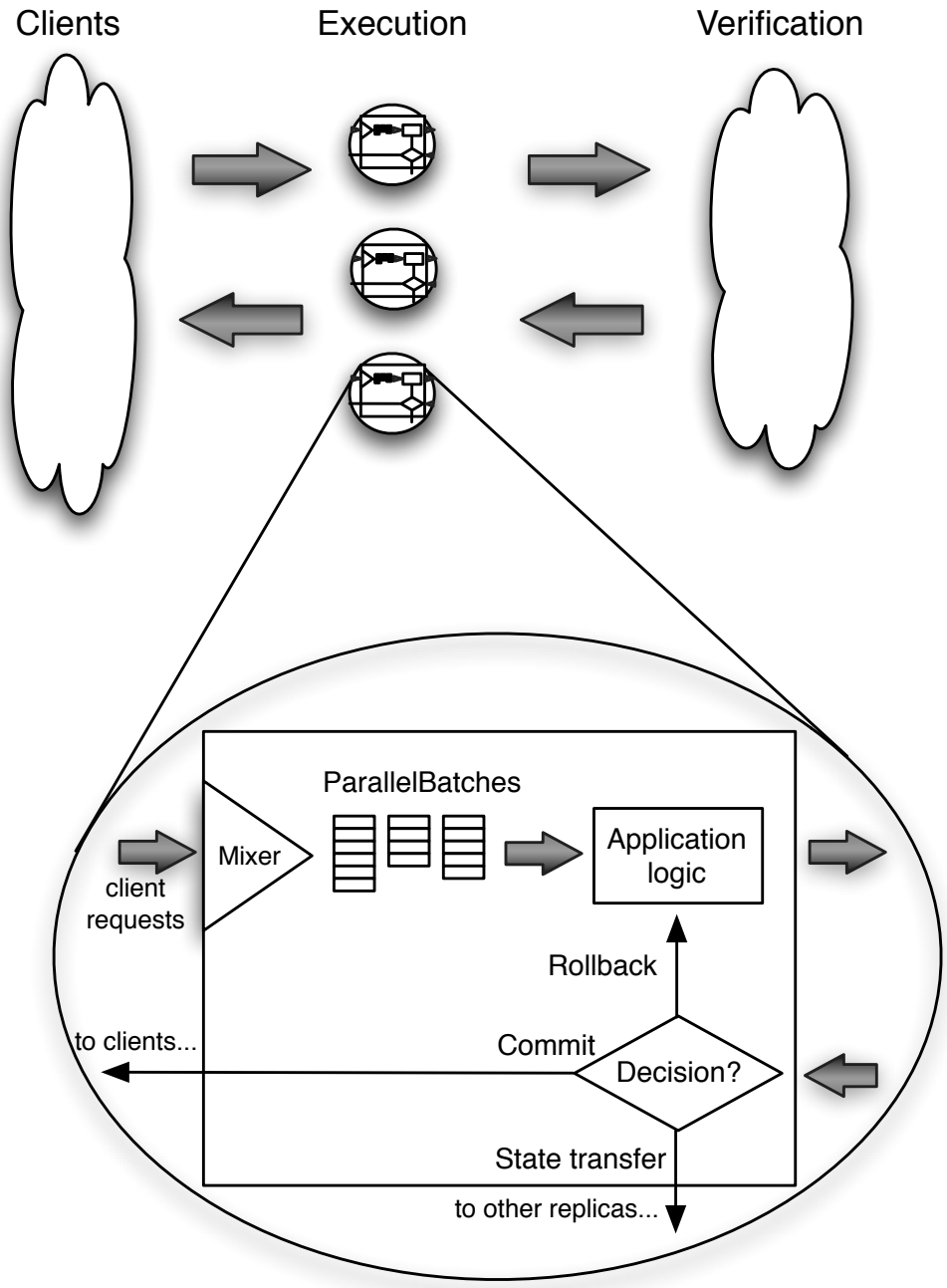


Figure 3.1: Overview of Eve.

in the $i^{th}$ batch, a replica computes a hash of its application state and of the outputs generated in response to requests in that batch.

This hash, along with the sequence number $i$ and the hash for batch $i-1$, constitute a *token* that is sent to the verification stage in order to discern whether the replicas have diverged. We include the hash for the previous batch to make sure that the system only accepts valid state transitions (see Section 3.4.1 for why this is necessary). Verification replicas will only accept a token as valid if they have already agreed that there is a committed hash for sequence number $i-1$ that matches the one in the $i^{th}$ token. Section 3.3.2 describes how we efficiently and deterministically compute the hash of the final state and outputs.

### 3.2.2 Verification stage

Eve's execution stage strives to make divergence unlikely, but offers no guarantees: for instance, despite its best effort, the mixer may inadvertently include conflicting requests in the same parallelBatch and cause distinct correct replicas to produce different final states and outputs. It is up to the verification stage to ensure that such divergences cannot affect safety, but only performance: at the end of the verification stage, all correct replicas that have executed the $i^{th}$ batch of requests are guaranteed to have reached the same final state and produced the same outputs.

**Agreement** The verification stage runs an agreement protocol to determine the final state and outputs of all correct replicas after each batch of requests. The input to the agreement protocol (see Section 3.4) are the tokens received from the execution replicas. The final decision is either *commit* (if enough tokens match) or *rollback* (if too many tokens differ). In particular, the protocol first verifies whether replicas have diverged at all: if all tokens agree, the replicas' common final state and outputs are committed. If there is divergence, the agreement protocol tallies the received tokens, trying to identify a final state and outputs pair reached by enough replicas to guarantee that the pair is the

product of a correct replica. If one such pair is found, then Eve ensures that all correct replicas commit to that state and outputs; if not, then the agreement protocol decides to roll back. Note that the actual number of matching tokens required depends on the configuration at hand. Section 3.4 discusses how this number is instantiated for two configurations: a BFT and a primary-backup configuration.

**Commit** If the result of the verification stage is *commit*, the execution replicas mark the corresponding sequence number as committed and send the responses for that parallelBatch to the clients.

**Rollback** If the result of the verification stage is *rollback*, the execution replicas roll back their state to the latest committed sequence number and re-execute the batch sequentially to guarantee progress. A rollback will also rotate the current primary, to ensure that a faulty primary cannot compromise liveness. Furthermore, to guarantee progress, the first batch created by the new primary, which typically includes some subset of the rolled back requests, is executed sequentially by all execution replicas.

A serendipitous consequence of its "execute-verify" architecture is that Eve can mask replica divergences caused by *concurrency bugs*, i.e. deviations from an application's intended behavior triggered by particular thread interleavings [31]. Some concurrency bugs may manifest as commission failures; however, because such failures are typically triggered probabilistically and are not the result of the actions of a strategic adversary, they can be often masked by configurations of Eve designed to tolerate only omission failures.

Note that Eve does not actually know what the application intended behavior is. Eve simply detects that different replicas have diverged in their state and responses, and asks them to roll back their state and re-execute the batch sequentially, thereby masking the concurrency bug. In other words, a concurrency bug is indistinguishable from a commission failure or a divergence due to inaccuracy at the mixer. As such, while Eve can be used to mask

| Transaction | Read and write keys |
|---|---|
| getBestSellers | **read**: item, author, order_line |
| getRelated | **read**: item |
| getMostRecentOrder | **read**: customer, cc_xacts, address, country, order_line |
| doCart | **read**: item<br>**write**: shopping_cart_line, shopping_cart |
| doBuyConfirm | **read**: customer, address<br>**write**: order_line, item, cc_xacts, shopping_cart_line |

Figure 3.2: The keys used for the 5 most frequent transactions of the TPC-W workload.

concurrency bugs, it is not very accurate in detecting the presence of such bugs, as it would introduce a fair amount of false positives.

Of course, like every system that uses redundancy to tolerate failures, Eve is vulnerable to *correlated failures* and cannot mask concurrency failures if too many replicas fail in exactly the same way. This said, Eve's architecture should help, both because the mixer, by trying to avoid parallelizing requests that interfere, makes concurrency bugs less likely and because concurrency bugs may manifest differently (if at all) on different replicas.

## 3.3 Execution stage

In this section we describe the execution stage in more detail. In particular, we discuss the design of the mixer and the design and implementation of the state management framework that allows Eve to perform efficient state comparison, state transfer, and rollback.

### 3.3.1 Mixer design

Parallel execution will result in better performance only if divergence is rare. The mission of the mixer is to identify requests that may productively be

executed in parallel and to do so with low false negative and false positive rates. False negatives will cause conflicting requests to be executed in parallel, creating the potential for divergence and rollback. False positives will cause requests that could have been successfully executed in parallel to be serialized, reducing the parallelism of the execution. Note however that Eve remains safe and live independent of the false negative and false positive rates of the mixer. A good mixer is just a performance optimization (albeit an important one).

The mixer we use for our experiments parses each request, trying to predict which state it will access: depending on the application, this state can vary from a single file or application-level object to higher-level objects such as database rows or tables. Two requests conflict when they access the same object in a read/write or write/write manner. To avoid putting together conflicting requests, the mixer starts with an empty parallelBatch and two (initially empty) hash tables, one for objects being read, the other for objects being written. The mixer then scans in turn each request, mapping the objects accessed in the request to a read or write key, as appropriate. Before adding a request to a parallelBatch, the mixer checks whether that request's keys have read/write or write/write conflicts with the keys already present in the two hash tables. If not, the mixer adds the request to the parallelBatch and adds its keys to the appropriate hash table; when a conflict occurs, the mixer tries to add the request to a different parallelBatch—or creates a new parallelBatch, if the request conflicts with all existing parallelBatches.

In our experiments with the H2 Database Engine and the TPC-W workload, we simply used the names of the tables accessed in read or write mode as read and write keys for each transaction[2] (see Table 3.2). Note that because the mixer can safely misclassify requests, we need not explicitly capture additional conflicts potentially generated through database triggers or view accesses that may be invisible to us: Eve's verification stage allows us to be safe without being perfect. Moreover, the mixer can be improved over time

[2]H2 uses coarse-grain table-level locking, so it did not make sense to implement conflict checks at a granularity finer than a table.

using feedback from the system (e.g. by logging parallelBatches that caused rollbacks).

Although implementing a perfect mixer might prove tricky for some cases, we expect that a good mixer can be written for many interesting applications and workloads with modest effort. Databases and key-value stores are examples of applications where requests typically identify the application-level objects that will be affected—tables and values respectively. Our experience so far is encouraging. Our TPC-W mixer took 10 student-hours to build, without any prior familiarity with the TPC-W code. As demonstrated in Section 3.5, this simple mixer achieves good parallelism (acceptably few false positives), and we do not observe any rollbacks (few or no false negatives).

### 3.3.2 State management

Moving from an agree-execute to an execute-verify architecture puts pressure on the implementation of state checkpointing, comparison, rollback, and transfer. For example, replicas in Eve must compute a hash of the application state reached after executing every batch of requests; in contrast, traditional SMR protocols checkpoint and compare application states much less often (e.g. when garbage collecting the request log).

To achieve efficient state comparison and fine-grained checkpointing and rollback, Eve stores the state using a copy-on-write Merkle tree, whose root is a concise representation of the entire state. The implementation borrows two ideas from BASE [62]. First, it includes only the subset of state that determines the operation of the state machine, omitting other state (such as an IP address or a TCP connection) that can vary across different replicas but has no semantic effect on the state and output produced by the application. Second, it provides an abstraction wrapper on some objects to mask variations across different replicas.

Similar to BASE [62] and other traditional SMR systems such as PBFT, Zyzzyva, and UpRight [14, 18, 42], where programmers are required to manu-

ally annotate which state is to be included in the state machine's checkpoint, our current implementation of Eve manually annotates the application code to denote the objects that should be added to the Merkle tree and to mark them as dirty when they get modified.

Compared to BASE, however, Eve faces two novel challenges: maintaining a deterministic Merkle tree structure under parallel execution and parallel hash generation as well as issues related to our choice to implement Eve in Java.

### Deterministic Merkle trees

To generate the same checksum, different replicas must put the same objects at the same location in their Merkle tree. In single-threaded execution, determinism comes easily by adding an object to the tree when it is created. Determinism is more challenging in multithreaded execution when objects can be created concurrently.

There are two intuitive ways to address the problem. The first option is to make memory allocation synchronized and deterministic. This approach not only negates efforts toward concurrent memory allocation [29, 67], but is unnecessary, since the allocation order usually does not fundamentally affect replica equivalence. The second option is to generate an ID based on object content and to use it to determine an object's location in the tree; this approach does not work, however, since many objects have the same content, especially at creation time.

Our solution is to postpone adding newly created objects to the Merkle tree until the end of the batch, when they can be added deterministically. Eve scans existing modified objects, and if one contains a reference to an object not yet in the tree, Eve adds that object into the tree's next empty slot and iteratively repeats the process for all newly added objects.

Object scanning is deterministic for two reasons. First, existing objects are already put at deterministic locations in the tree. Second, given an object, Eve can iterate all its references in a deterministic order. Usually we can use

the order in which references are defined in a class. However some classes, like *Hashtable*, do not store their references in a deterministic order; we discuss how to address these classes below.

We do not parallelize the process of scanning for new objects, since it has low overhead. We do parallelize hash generation, however: we split the Merkle tree into subtrees and compute their hashes in parallel before combining them to obtain the hash of the Merkle tree's root.

### Java Language & Runtime

The choice of implementing our prototype in Java provides us with several desirable features, including an easy way to differentiate references from other data that simplifies the implementation of deterministic scanning; at the same time, it also raises some challenges.

First, objects which the Merkle tree holds a reference to are not eligible for Java's automatic garbage collection (GC). Our solution is to periodically perform a Merkle-tree-level scan, using a mark-and-sweep algorithm similar to Java's GC, to find unused objects and remove them from the tree. This ensures that those objects can be correctly garbage collected by Java's GC. For the applications we have considered, this scan can be performed less frequently than Java's GC, since objects in the tree tend to be "important" and have a long lifetime. In our experience this scan is not a major source of overhead.

Second, several standard set-like data structures in Java, including instances of the widely-used Hashtable and HashSet classes, are not oblivious to the order in which they are populated. For example, the serialized state of a Java Hashtable object is sensitive to the order in which keys are added and removed. So, while two set-like data structures at different replicas may contain the same elements, they may generate different checksums when added to a Merkle tree: while semantically equivalent, the states of these replicas would instead be seen as having diverged, triggering unnecessary rollbacks.

Our solution is to create wrappers [62] that abstract away semantically irrelevant differences between instances of set-like classes kept at different repli-

cas. The wrappers generate, for each set-like data structure, a deterministic list of all the elements it contains, and, if necessary, a corresponding iterator. If the elements' type is one for which Java already provides a comparator (e.g. Integer, Long, String, etc.), this is easy to do. Otherwise, the elements are sorted using an ordered pair *(requestId, count)* that Eve assigns to each element before adding it to the data structure. Here, *requestId* is the unique identifier of the request responsible for adding the element, and *count* is the number of elements added so far to the data structure by request *requestId*. In practice, we only found the need to generate two wrappers, one for each of the two interfaces (Set and Map) commonly used by Java's set-like data structures.

## 3.4 Verification stage

The goal of the verification stage is to determine whether enough execution replicas agree on their state and responses after executing a batch of requests. Given that the tokens produced by the execution replicas reflect their current state as well as the state transition they underwent, all the verification stage has to decide is whether enough of these tokens match.

To come to that decision, the verification replicas use an agreement protocol [14, 46] whose details depend largely on the system model. We present the protocol for two extreme cases: an asynchronous Byzantine fault tolerant system, and a synchronous primary-backup system. We then discuss how the verification stage can offer some defense against concurrency bugs and how it can be tuned to maximize the number of tolerated concurrency bugs.

### 3.4.1 Asynchronous BFT

In this section we describe the verification protocol for an asynchronous Byzantine fault tolerant system with $n_E = u + max(u, r) + 1$ execution replicas and $n_V = 2u + r + 1$ verification replicas [17, 18], which allows the system to remain

live despite $u$ failures (whether of omission or commission), and safe despite $r$ commission failures and any number of omission failures. Readers familiar with PBFT [14] will find many similarities between these two protocols; this is not surprising, since both protocols attempt to perform agreement among $2u + r + 1$ replicas ($3f + 1$ in PBFT terminology). The main differences between these protocols stem from two factors. First, in PBFT the replicas try to agree on the output of a single node—the primary. In Eve the object of agreement is the behavior of a collection of replicas—the execution replicas. Therefore, in Eve verification replicas use a quorum of $max(u, r) + 1$ matching tokens from the execution replicas, if available, as their "proposed" value. Second, in PBFT the replicas try to agree on the inputs to the state machine (the incoming requests and their order). Instead, in Eve replicas try to agree on the outputs of the state machine (the application state and the responses to the clients). Hence, in the view change protocol (described below) the existence of a certificate for a given sequence number is enough to commit that sequence number to the next view—a prefix of committed sequence numbers is no longer required.

**Why $u+max(u,r)+1$?** At first glance, it might seem like Eve would require $u+r+1$ execution replicas, since one must receive at least $Q \geq r+1$ matching responses (for safety) and cannot wait for more than $Q \leq n_E - u$ responses (for liveness); so $n_E \geq u + r + 1$. There is, however, an additional, rather subtle, requirement: that the application state is maintained despite failures. For example, consider a system where $u = 2$ and $r = 1$. In this system committing requests based on $r + 1(= 2)$ responses could cause the application state to be lost if those two replicas were to fail permanently.

Most previous replication protocols implicitly satisfy this requirement by using $3f+1$ execution replicas that are co-located with the agreement replicas. Since Eve separates agreement—or rather verification—from execution, it can afford to use fewer execution replicas. The price of this reduction in the replication factor is that we must take care that the application state is

not permanently lost. In particular, we must ensure that updates to the application state are performed in at least $Q \geq u + 1$ replicas. Taking the safety requirement into consideration, we have that $Q \geq max(u, r) + 1$. Adding the liveness requirement, we get $n_E \geq u + max(u, r) + 1$.

**The protocol** When an execution replica executes a batch of requests (i.e., a sequence of parallelBatches), it sends a $\langle \text{VERIFY}, view, n, T, e \rangle_{\vec{\mu}_e}$ message to all verification replicas, where *view* is the current view number, $n$ is the batch sequence number, $T$ is the computed token for that batch, and $e$ is the sending execution replica. Recall that $T$ contains both the hash of batch $n$ and the hash of batch $n-1$: a verification replica accepts a VERIFY message for batch $n$ only if it has previously committed a hash for batch $n-1$ that matches the one stored in $T$. This mechanism is necessary to ensure that the verification replicas only commit states that correspond to valid state transitions.

Without this mechanism in place, it would be possible for a malicious replica to coerce the verification replicas to commit two states $s_{n-1}$ and $s_n$ for sequence numbers $n-1$ and $n$, respectively, that do not correspond to a valid state transition taken by a correct replica. To understand how this is possible, consider a simple scenario where $u = r = 1$. This means there are 3 execution replicas—call them $A$, $B$, and $C$. In this scenario $B$ is malicious. In batch $n-1$, $A$ and $C$ diverge, and $B$ produces the same token as $A$, causing the state of $A$ to be committed for this batch. In the next batch ($n$), $C$ is still diverged from the committed state, but does not know it yet, so it proceeds to execute this batch and produce a token. This time, $B$ "sides" with $C$: it produces the same token as $C$, causing this token to be committed. That way, $B$ managed to commit two states that belong to diverged executions and may not correspond to any valid state transition of a correct replica.

**Case 1: Replicas reach agreement** When a verification replica $v$ receives $max(u, r) + 1$ VERIFY messages with matching tokens, it marks this sequence number as *preprepared* and sends a $\langle \text{PREPARE}, view, n, T, v \rangle_{\vec{\mu}_v}$ mes-

sage to all other verification replicas. Similarly, when a verification replica $v$ receives $n_V - u$ matching PREPARE messages, it marks this sequence number as *prepared* and sends a $\langle \text{COMMIT}, view, n, T, v \rangle_{\vec{\mu}_v}$ to all other verification replicas. Finally, when a verification replica $v$ receives $n_V - u$ matching COMMIT messages, it marks this sequence number as *committed* and sends a $\langle \text{VERIFY-RESPONSE}, view, n, T, v \rangle_{\vec{\mu}_v}$ message to all execution replicas. Note that the view number *view* is the same as that of the VERIFY message; this indicates that agreement was reached and no view change was necessary.

**Case 2: No agreement reached—view change** In order to guarantee liveness, replicas monitor the rate of committed requests and initiate a view change if it is not high enough [16]. This can happen a) because the primary execution replica misbehaved and sent different batches of requests to different replicas; or b) as a result of replica divergence, which stems from the mixer allowing two conflicting requests to execute in parallel; or c) because of asynchrony or network loss.

If a verifier replica $v$ receives $n_E - max(u, r) + 1$ mutually non-matching tokens—and is therefore certain that no matching quorum is possible—or the commit rate is not high enough, it proposes a view change by sending a $\langle \text{VIEW-CHANGE}, view+1, \mathcal{P}, \mathcal{PP}[\,], v \rangle_{\vec{\mu}_v}$ to all verifier replicas, where *view + 1* is the next view number, $\mathcal{P}$ is the maximum prepared batch sequence number, along with the corresponding VERIFY messages, and $\mathcal{PP}[\,]$ is the sequence numbers it has preprepared that are greater than $\mathcal{P}$, along with the corresponding VERIFY messages. A correct replica receiving a valid VIEW-CHANGE message will send a $\langle \text{VIEW-CHANGE-ACK}, view+1, s, d, v \rangle_{\vec{\mu}_v}$ to the primary verifier replica for view *view*, where *view + 1* is the view number contained in the received VIEW-CHANGE message, $s$ is the sender of the message and $d$ is a digest of the message.

When the primary verifier replica for that view receives $n_V - u$ valid VIEW-CHANGE messages, each supported by $n_V - u$ valid VIEW-CHANGE-ACK messages, it tries to identify a sequence number that should be committed.

First it identifies the maximum prepared sequence number $\mathcal{P}$ among the VIEW-CHANGE messages it received. It further tries to identify $r + 1$ replicas that have preprepared the same token for a sequence number greater than $\mathcal{P}$. If such a sequence number can be found, the primary will select it as the sequence number it will commit; otherwise $\mathcal{P}$ is selected. Next, the primary will send the message $\langle \text{NEW-VIEW}, view + 1, n, T, \mathcal{VC}, v \rangle_{\vec{\mu}_v}$ to all verifier replicas, where *view + 1* is the number of the new view, $n$ is the selected sequence number, $T$ is the corresponding token and $\mathcal{VC}$ is the quorum of VIEW-CHANGE messages that prove that the primary's choice was valid.

A correct replica that receives a valid NEW-VIEW message will change its current view number to *view* (assuming it is greater than its current view number) and send a $\langle \text{VERIFY-RESPONSE}, view + 1, n, T, v, f \rangle_{\vec{\mu}_v}$ message to all execution replicas, where $f$ is a flag that indicates that the next batch should be executed sequentially to ensure progress. Note that in this case the view number has increased; this indicates that agreement was not reached and a rollback to sequence number $n$ is required.

**Commit, State transfer and Rollback** Upon receipt of $r + 1$ matching VERIFY-RESPONSE messages, an execution replica $e$ distinguishes three cases:

**Commit** If the view number has not increased and the agreed-upon token matches the one $e$ previously sent, then $e$ marks that sequence number as stable, garbage-collects any portions of the state that have now become obsolete, and releases the responses computed from the requests in this batch to the corresponding clients.

**State transfer** If the view number has not increased, but the token does not match the one $e$ previously sent, it means that this replica has diverged from the agreed-upon state. To repair this divergence, $e$ issues a state transfer request to other replicas. This transfer is incremental: rather than transferring the entire state, Eve transfers only the part that has changed since the last stable sequence number. Incremental transfer,

which uses the Merkle tree to identify what state needs to be transferred, allows Eve to rapidly bring slow and diverging replicas up-to-date.

When performing a state transfer, the requesting replica sends its latest committed sequence number and the responding replica only sends the state difference from that sequence number to its own latest committed sequence number. Before applying the state difference, the requesting replica rolls back its own state to its latest committed sequence number, undoing any changes that might contribute to the divergence. Finally, to tolerate commission failures, the requesting replica checks that the state sent by the responding replica matches the token $T$ included in the VERIFY-RESPONSE message. This check ensures that the replica only accepts the state that the verification replicas agreed upon as the committed state for the corresponding sequence number.

**Rollback** If the view number has increased, this means that agreement could not be reached. Replica $e$ discards any unexecuted requests and rolls back its state to the sequence number indicated by the token $T$, while verifying that its new state matches the token (else it initiates a state transfer). The increased view number also implicitly rotates the execution primary. The replicas start receiving batches from the new primary and, since the flag $f$ was set, execute the first batch sequentially to ensure progress. In general, the flag does not need to trigger sequential execution immediately; for example, a different implementation would be to only employ sequential execution after $k$ consecutive attempts at parallel execution have led to divergence, for some—configurable—value of $k$.

### Read-only requests

Previous BFT protocols like PBFT and Zyzzyva [14, 43] handled read-only requests differently: requests that do not modify the state are executed at only $2f + 1$—out of $3f + 1$ total replicas—without going through the ordering

phase, where $f$ is the number of failures that the system can tolerate.

Perhaps surprisingly, Eve does not employ this so-called "read-only optimization". There are three reasons why this optimization is not as compelling in Eve.

First, Eve already executes requests in a reduced number of replicas: Eve uses $u + max(u, r) + 1$ execution replicas, which corresponds to $2f + 1$ replicas in the terminology of PBFT and Zyzzyva. Second, since Eve does not perform agreement on the ordering of requests, but rather on the state and outputs of the replicas, skipping the verification for read-only requests would only provide minimal performance gains.

Finally, ensuring liveness in an asynchronous system with $2f+1$ replicas, requires waiting for no more than $f + 1$ responses. Since a normal request can be committed with only $max(u, r) + 1$—i.e., $f + 1$—replicas, the quorum of responses to a read-only request overlaps with the quorum of normal requests in only one replica. Were that replica Byzantine, a read-only request could return a stale result, violating the abstraction of a single correct server.

### 3.4.2 Synchronous primary-backup

A system configured for synchronous primary-backup has only two replicas that are responsible for both execution and verification. While traditional primary-backup employs passive replication, where the backup simply absorbs state updates from the primary, Eve's primary-backup configuration adopts the active approach, where both replicas execute client requests independently.

The primary receives client requests and groups them into batches. When a batch $\mathcal{B}$ is formed, it sends a $\langle \text{EXECUTE-BATCH}, n, \mathcal{B}, ND \rangle$ message to the backup, where $n$ is the batch sequence number and $ND$ is the data needed for consistent execution of nondeterministic calls such as `random()` and `gettimeofday()`. Both replicas apply the mixer to the batch, execute the resulting parallelBatches, and compute the state token, as described in Section 3.2. The backup sends its token to the primary, which compares it to its own

token. If the tokens match, the primary marks this sequence number as stable and releases the responses to the clients. If the tokens differ, the primary rolls back its state to the latest stable sequence number and notifies the backup to do the same. To ensure progress, they execute the next batch sequentially.

If the primary crashes, the backup is eventually notified and assumes the role of the primary. As long as the old primary is unavailable, the new primary will keep executing requests on its own. After a period of unavailability, a replica uses incremental state transfer to bring its state up-to-date before processing any new requests.

### 3.4.3 Tolerating concurrency bugs

A happy consequence of the execute-verify architecture is that even when configured with the minimum number of replicas required to tolerate $u$ omission faults, Eve provides some protection against concurrency bugs.

Concurrency bugs can lead to both omission faults (e.g., a replica could get stuck) and commission faults (e.g., a replica could produce an output or transition to a state that is not part of its intended behavior). However, faults due to concurrency bugs have two important properties that in general cannot be assumed for Byzantine faults. First, since parallel execution is only employed among execution threads, such faults only affect the application state and outputs, and cannot corrupt the state of the replication protocol. Second, they are easy to repair. If Eve detects a concurrency fault, it can repair the fault via rollback and sequential re-execution.

It is important to remember that Eve does not need to know what the intended behavior of the application is; it simply detects that some replicas diverge from other replicas in their outputs and state transitions. This method is particularly effective against concurrency bugs, because such bugs only occur in a—typically small—subset of interleavings and are thus unlikely to affect a large number of replicas simultaneously.

**Asynchronous case** When configured with $r = 0$, Eve provides the following guarantee:

**Theorem 1** *When configured with $n_E = 2u + 1$ and $r = 0$, asynchronous Eve is safe, live, and correct despite up to $u$ concurrency or omission faults.*

Note that safety and liveness refer to the requirements of state machine replication—that the committed state and outputs at correct replicas match and that requests eventually commit. Correctness refers to the state machine itself; a committed state is correct if it is a state that can be reached by the state machine in a fault-free run.

**Proof sketch:** The system is always safe and correct because the verifier requires $u+1$ matching execution tokens to commit a batch. If there are at most $u$ concurrency faults and no other commission faults, then every committed batch has at least one execution token produced by a correct replica.

The system is live because if a batch fails to gather $u + 1$ matching tokens, the verifier forces the execution replicas to roll back and sequentially re-execute. During sequential execution deterministic correct replicas do not diverge; so, re-execution suffers at most $u$ omission faults and produces at least $u + 1$ matching execution tokens, allowing the batch to commit. ∎

When more than $u$ correlated concurrency faults produce exactly the same state and output, Eve still provides the safety and liveness properties of state machine replication, but can no longer guarantee correctness of the state machine itself.

**Synchronous case** When configured with just $u+1$ execution replicas, Eve can continue to operate with one replica if $u$ replicas fail by omission. In such configurations, Eve does not have spare redundancy and can not mask concurrency faults at the one remaining replica.

**Extra protection during good intervals** During *good intervals* when there are no replica faults or timeouts other than those caused by concurrency

bugs, Eve uses spare redundancy to boost its best-effort protection against concurrency bugs to $n_E - 1$ execution replicas in both the synchronous and asynchronous cases.

For example, in the synchronous primary-backup case, when both execution replicas are alive, the primary receives both execution responses, and if they do not match, it orders a rollback and sequential re-execution. Thus, during a good interval this configuration masks one-replica concurrency failures. We expect this to be the common case.

In both the synchronous and asynchronous case Eve, when configured for $r = 0$, enters *extra protection mode* (EPM) after $k$ consecutive batches for which all $n_E$ execution replicas provided matching, timely responses. While Eve is in EPM, after the verifiers receive the minimum number of execution responses necessary for progress, they continue to wait for up to a short timeout to receive all $n_E$ responses. If the verifiers receive all $n_E$ matching responses, they commit the response. Otherwise, they order a rollback and sequential re-execution. Then, if they receive $n_E$ matching responses within a short timeout, they commit the response and remain in EPM. Conversely, if sequential re-execution does not produce $n_E$ matching and timely responses, they suspect a non-concurrency failure and exit EPM to ensure liveness by allowing the system to make progress with fewer matching responses.

## 3.5 Evaluation

Our evaluation aims to answer the following questions:

- What is the throughput gain that Eve provides compared to a traditional sequential execution approach?
- How does Eve's performance compare to an unreplicated multithreaded execution and alternative replication approaches?
- How is Eve's performance affected by the mixer and by other workload characteristics?

- How efficient is Eve at masking concurrency bugs?

We address these questions by using a key-value store application and the H2 Database Engine. We implemented a simple key-value store application to perform microbenchmark measurements of Eve's sensitivity to various parameters. Specifically, we vary the amount of execution time required per request, the size of the application objects and the accuracy of our mixer, in terms of both false positives and false negatives. For the H2 Database Engine we use an open-source implementation of the TPC-W benchmark [69, 70]. We present the results of the browsing workload, which has more opportunities for concurrency.

Our current prototype omits some of the features described above. Specifically, although we implement the extra protection mode optimization from Section 3.4.3 for synchronous primary-backup replication, we do not implement it for our asynchronous configurations. Also, our current implementation does not handle applications that include objects for which Java's *finalize* method modifies state that needs to be consistent across replicas. Finally, our current prototype only supports in-memory application state.

We run our microbenchmarks on an Emulab testbed with 14x 4-core Intel Xeon @2.4 GHz, 4x 8-core Intel Xeon @2.66 GHz, and 2x 8-core hyper-threaded Intel Xeon @1.6 GHz, connected with a 1 Gb Ethernet. We were able to get limited access to 3x 16-core AMD Opteron @3.0 GHz and 2x 8-core Intel Xeon L5420 @2.5 Ghz. We use the AMD machines as execution replicas to run the TPC-W benchmark on the H2 Database Engine for both the synchronous primary-backup and the asynchronous BFT configuration (Figure 3.3). For the asynchronous BFT configuration we use 3 execution and 4 verifier nodes, which are sufficient to tolerate 1 Byzantine fault ($u = 1$, $r = 1$). The L5420 machines are running Xen and we use them to perform our comparison with Remus (Figure 3.10 and Figure 3.11).

### 3.5.1 H2 Database with TPC-W

Figures 3.3 demonstrates the performance of Eve for the H2 Database Engine [33] with the TPC-W browsing workload [69, 70]. We report the throughput of Eve using an asynchronous BFT configuration (*Eve-BFT*) and a synchronous active primary-backup configuration (*Eve-PrimaryBackup*). We compare against the throughput achieved by an unreplicated server that uses sequential execution regardless of the number of available hardware threads (*sequential*). Note that this represents an upper bound of the performance achievable by previous replication systems that use sequential execution [14, 18, 46, 52]. We also compare against the performance of an unreplicated server that uses parallel execution.

With 16 execution threads, Eve achieves a speedup of 6.5x compared to sequential execution. That approaches the 7.5x speedup achieved by an unreplicated H2 Database server using 16 threads.

In both configurations and across all runs and for all data points, Eve never needs to roll back. This suggests that our simple mixer never parallelized requests it should have serialized. At the same time, the good speedup indicates that it was adequately aggressive in identifying opportunities for parallelism.

### 3.5.2 Microbenchmarks

In this section, we use a simple key-value store application to measure how various parameters affect Eve's performance. We only show the graphs for the primary-backup configuration; the results for asynchronous replication are very similar, because the verification stage has minimal impact on performance, as Figure 3.3 shows. Except when noted, the default workload consumes 1 ms of execution time per request, each request updates one application object, and the application object size is 1 KB.

Figure 3.4 shows the impact of varying the CPU demand of each request. We observe that heavier workloads (10 ms of execution time per request) scale

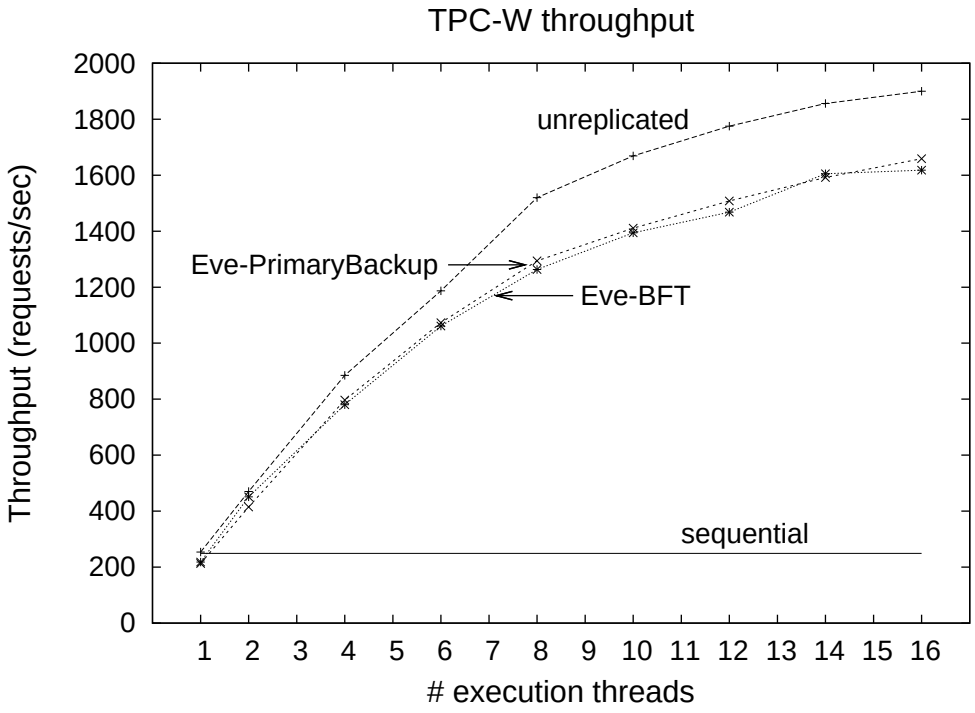


Figure 3.3: The throughput of Eve running the TPC-W browsing workload on the H2 Database Engine.

well, up to 12.5x on 16 threads compared to sequential execution. As the workload gets lighter, the overhead of Eve becomes more pronounced. Speedups fall to 10x for 1 ms/request and to 3.3x for 0.1 ms/request. The 3.3x scaling is partially an artifact of our inability to fully load the server with lightweight requests. In our workload generator, clients have one outstanding request at a time, thus requiring a high number of clients to saturate the servers; this causes our servers to run out of sockets before they are fully loaded. We measure our server CPU utilization during this experiment to be about 30%.

In Figure 3.4 we plot throughput speedup, so that trends are apparent. For reference, the absolute peak throughputs in requests per second are 25.2K,

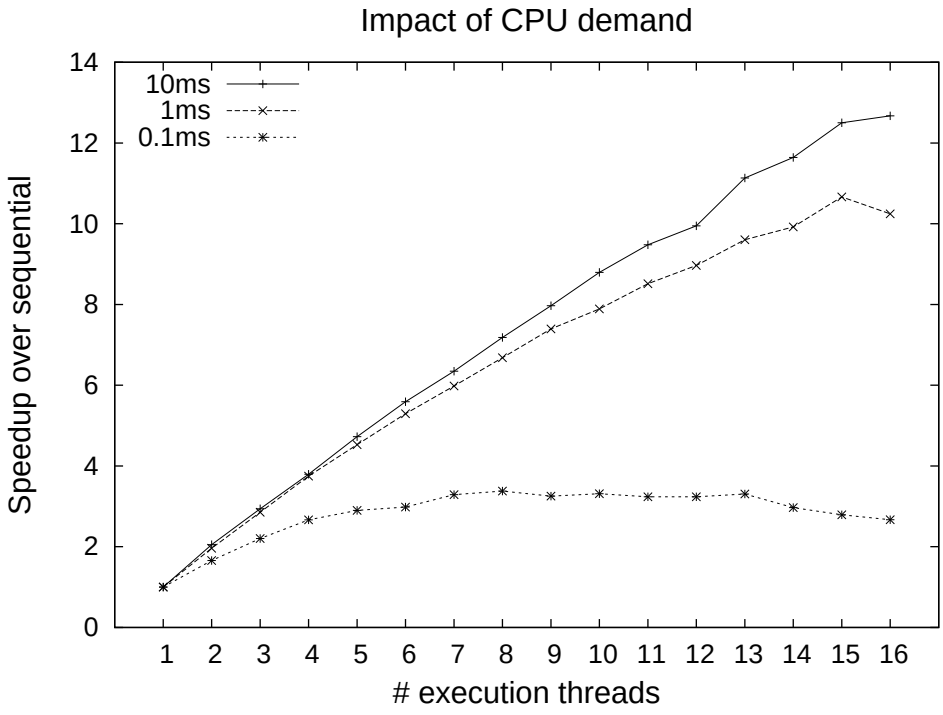


Figure 3.4: The impact of CPU demand per request on Eve's throughput speedup.

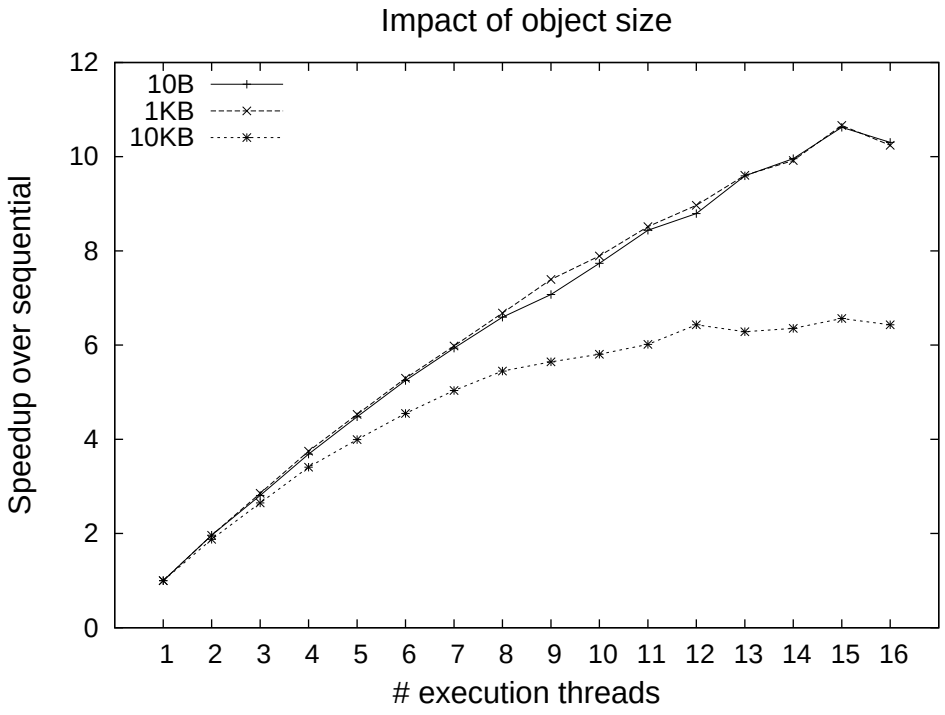


Figure 3.5: The impact of application object size on Eve's throughput speedup.

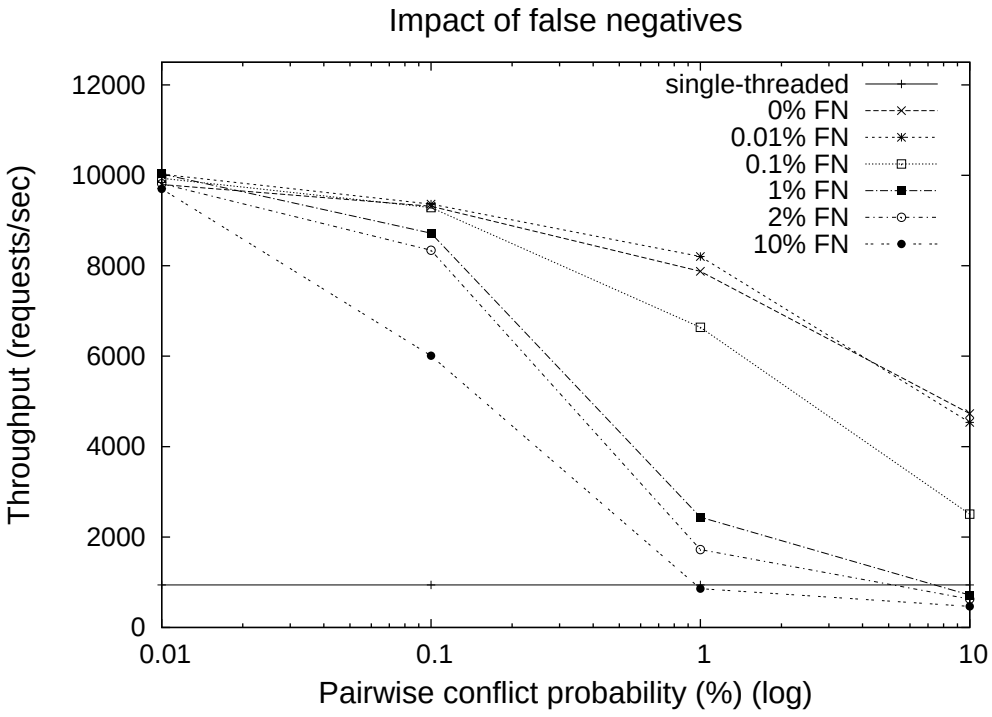


Figure 3.6: The impact of conflict probability and false negative rate on Eve's throughput.

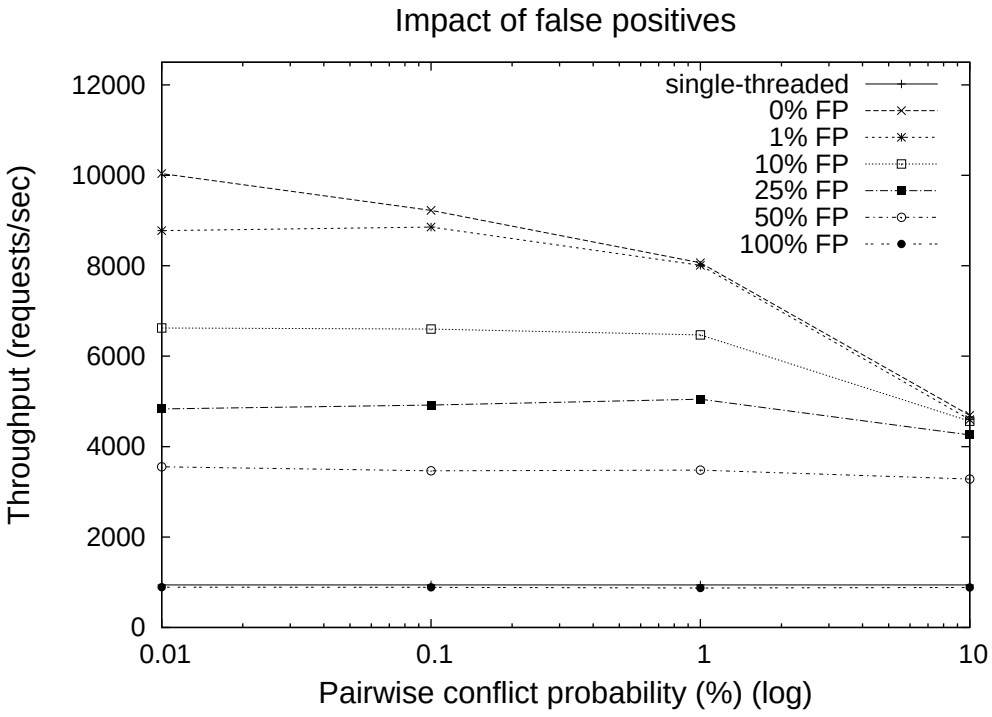


Figure 3.7: The impact of conflict probability and false positive rate on Eve's throughput.

10.0K, 1242 for the 0.1 ms, 1 ms, 10 ms lines, respectively.

The next experiment explores the impact of the application object size on the system throughput. We run the experiment using object sizes of 10 B, 1 KB, and 10 KB. Figure 3.5 shows the results. While the achieved throughput scales well for object sizes of 10 B and 1 KB, its scalability decreases for larger objects (10 KB). This is an artifact of the hashing library we use, as it first copies the object before computing its hash: for large objects, this memory copy limits the achievable throughput. Note that in this figure we plot throughput speedup rather than absolute throughput to better indicate the trends across workloads. For reference, the absolute peak throughput values in requests per second are 10.0K, 10.0K, 5.6K for the 10 B, 1 KB, 10 KB lines, respectively.

Next, we evaluate Eve's sensitivity to inaccurate mixers. Specifically, we explore the limits of tolerance to false negatives (misclassifying conflicting requests as non-conflicting) and false positives (misclassifying non-conflicting requests as conflicting). The effect of these parameters is measured as a function of the pairwise conflict probability: the probability that two requests have a conflict. In practice, we achieve this by having each request modify one object and then varying the number of application objects. For example, to produce a 1% conflict chance, we create 100 objects. Similarly, a 1% false negative rate means that each pair of conflicting requests has a 1% chance of being classified as non-conflicting.

Figure 3.6 shows the effect of false negatives on throughput. First notice that, even with no false negatives, the throughput drops as the pairwise conflict chance increases because of the decrease of available parallelism. For example, if a batch has 100 requests and each request has a 10% chance of conflicting with each other request, then a perfect mixer is likely to divide the batch into about 10 parallelBatches, each with about 10 requests.

When we add false negatives, we add rollbacks, and the number of rollbacks increases with both the underlying conflict rate and the false negative rate. Notice that the impact builds more quickly than one might expect be-

cause there is essentially a birthday "paradox"—if we have a 1% conflict rate and a 1% false negative rate, then the probability that any pair of conflicting requests be misclassified is 1 in 10000. But in a batch of 100 requests, each of these requests has about a 1% chance of being party to a conflict, which means there is about a 39% chance that a batch of 100 requests contain an undetected conflict. Furthermore, with a 1% conflict rate, the batch will be divided into only a few parallelBatches, so there is a good chance that conflicting requests will land in the same parallelBatch. In fact, in this case we measure 1 rollback per 7 parallelBatches executed. Despite this high conflict rate and this high number of rollbacks, Eve achieves a speedup of 2.6x compared to sequential execution.

Figure 3.7 shows the effect of false positives on throughput. As expected, increased false positive ratios can lead to lower throughput, but the effect is not as significant as for false negatives. The reason is simple: false positives reduce the opportunities for parallel execution, but they don't incur any additional overhead.

From these experiments, we conclude that Eve does require a good mixer to achieve good performance. This requirement does not particularly worry us. We found it easy to build a mixer that (to the best of our knowledge) detects all conflicts and still allows for a good amount of parallelism. Others have had similar experience [44]. Although creating perfect mixers may be difficult in some cases, we speculate that it will often be feasible to construct mixers with the low false negative rates and modest false positive rates needed by Eve.

### 3.5.3 Failure and recovery

In Figure 3.8, we demonstrate Eve's ability to mask and recover from failures. In the primary-backup configuration we run an experiment where we kill the primary node $n_1$ at $t = 30$ seconds and recover it at $t = 60$ seconds (by which time the secondary $n_2$ has become the new primary). We then kill

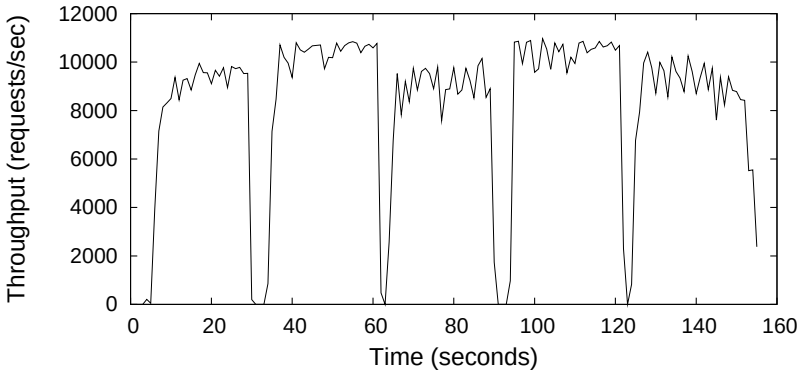


Figure 3.8: Throughput during node crash and recovery for an Eve primary-backup configuration.

the new secondary ($n_1$) at $t = 90$ seconds and recover it at $t = 120$ seconds. We observe that after the first failure the throughput drops to zero until the backup realizes that the primary is dead after a timeout of four seconds.[3] The backup then assumes the role of the primary and starts processing requests. The throughput during this period is higher because the new primary knows that the other node is crashed and does not send any messages to it. At $t = 60$, the first node recovers, and the throughput drops to zero for about one second while the newly recovered node catches up. Then the throughput returns to its original value. The process repeats when $n_1$ crashes again at $t = 90$ seconds and recovers at $t = 120$ seconds.

### 3.5.4 Concurrency faults

To evaluate Eve's ability to mask concurrency faults, we use a primary-backup configuration with 16 execution threads and run the TPC-W browsing workload on the H2 Database Engine with various mixers. H2 has a previously undiagnosed concurrency bug in which a row counter is not incremented properly when multiple requests access the same table in *read_uncommitted* mode. Our standard mixer completely masks this bug because it does not let re-

[3]One could use a fast failure detector [50] to achieve sub-second detection.

| | Group all | 1% FN | 0.5% FN | 0.1% FN | Original Mixer |
|---|---|---|---|---|---|
| Times bug manifested | 73 | 51 | 29 | 4 | 0 |
| Fixed with rollback | 60 | 38 | 18 | 3 | 0 |
| All identical (not masked) | 13 | 13 | 11 | 1 | 0 |
| Throughput | 1104 | 1233 | 1240 | 1299 | 1322 |

Figure 3.9: Effectiveness of Eve in masking concurrency bugs when various mixers are used.

quests that modify the same table execute in parallel. By introducing less accurate mixers we explore how well Eve's second line of defense—the verification stage—works in masking this bug.

Figure 3.9 shows the number of times that the bug manifested in one or both replicas. When the bug manifests only in one replica, Eve detects that the replicas have diverged and repairs the damage by rolling back and reexecuting sequentially. If the bug happens to manifest in both replicas in the same way, Eve will not detect it.

The first column shows the results when there is a trivial aggressive mixer that places all requests of batch $i$ in the same parallelBatch. In this case, all requests that arrive together in a batch are allowed to execute in parallel. Naturally, this case has the highest number of bug manifestations. We observe that even when the mixer does no filtering at all, Eve masks 82% of the instances where the bug manifests. In the remaining 18% of the cases, the bug manifested in the same way in both replicas and was not corrected by Eve. In columns 2 through 4, we introduce mixers with high rates of false negatives. This results in fewer manifestations of the bug, with Eve still masking the majority of those manifestations. In the fifth column, we show results for our original mixer, which (to the best of our knowledge) does not introduce false negatives. In this case, the bug does not manifest at all.

Although we do not claim that these results are general, we find them promising.

### 3.5.5 Remus

Remus [23] is a primary-backup system that uses Virtual Machines (VMs) to send modified state from the primary to the backup. An advantage of this approach is that it is simple and requires no modifications to the application. A drawback of this approach is that it aggressively utilizes network resources to keep the backup consistent with the primary. The issue is aggravated by two properties of Remus. First, Remus does not make fine-grain distinctions between state that is required for the state machine and temporary state. Second, Remus operates on the VM level, which forces it to send entire pages, rather than just the modified objects. Also, because Remus is using passive replication, it tolerates a narrower range of faults than Eve. Our experiments show that, despite Eve's stronger guarantees, it outperforms Remus by a factor of 4.7x, while using two orders of magnitude less network bandwidth.

Figure 3.10 shows the throughput achieved by Remus and Eve on the browsing workload of the TPC-W benchmark. We also show the latency and throughput of the unreplicated system for the same workload. Both systems run the H2 Database Engine on Xen and using a 2-node (primary-backup) configuration. Remus achieves a maximum throughput of 235 requests per second, while Eve peaks at 1225 requests per second. Remus crashes for loads higher than 235 requests per second, as its bandwidth requirements approach the capacity of the network, as Figure 3.11 shows. In contrast with Remus, Eve executes requests independently at each replica and does not need to propagate state modifications over the network. The practical consequence is that Eve uses significantly less bandwidth, achieves higher throughput, and provides stronger guarantees compared to a passive replication approach like Remus. Of course, the increased performance and stronger guarantees do not come for free. To achieve them, Eve pays the price of having to manually identify the relevant parts of the application state and abstract away the irrelevant ones; a programming effort that is not required for Remus.

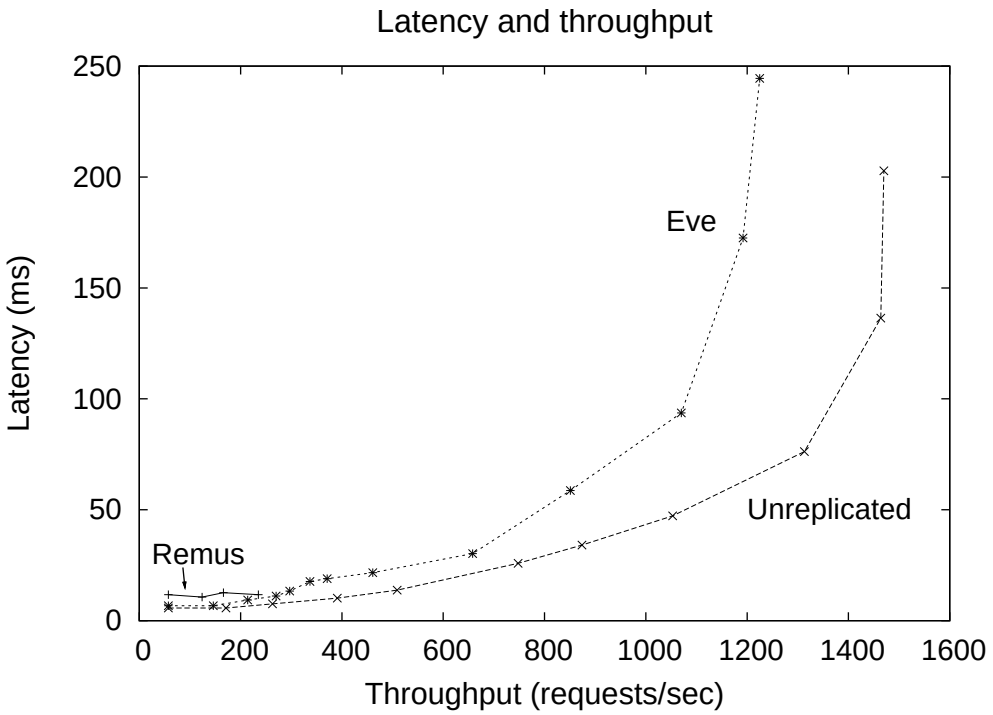


Figure 3.10: The latency and throughput of Remus and Eve running the H2 Database Engine on Xen. Both systems use a 2-node configuration. The workload is the browsing workload of the TPC-W benchmark.

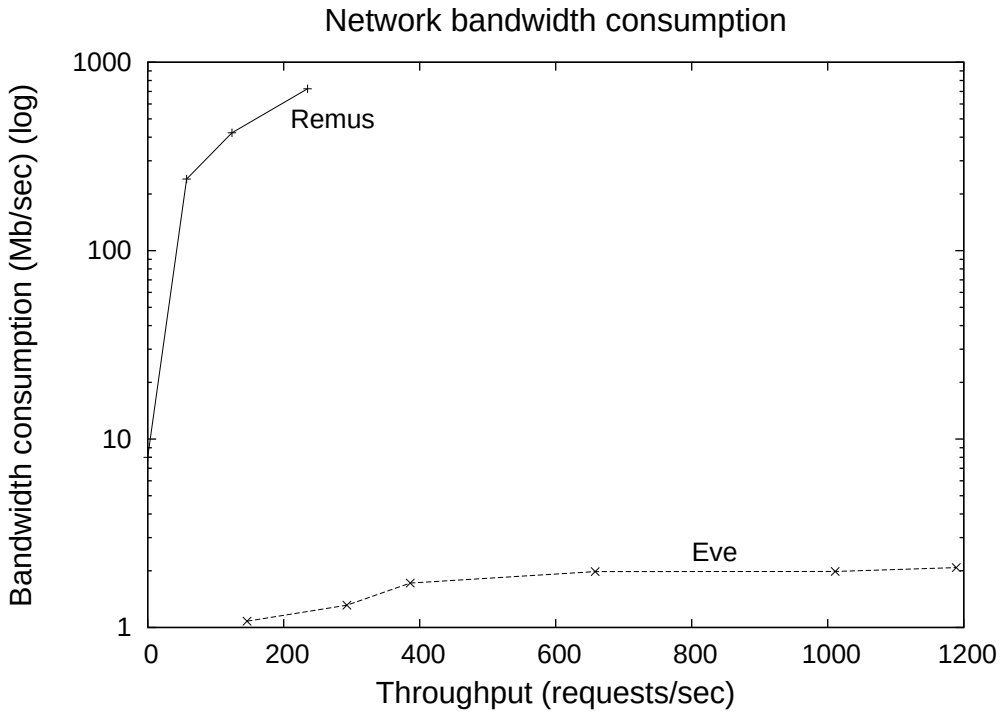


Figure 3.11: The bandwidth consumption of Remus and Eve for the experiment shown in Figure 3.10.

### 3.5.6 Latency and batching

Figure 3.10 provides some insight in Eve's tradeoff between latency and throughput. When Eve is not saturated, its latency is only marginally higher than that of an unreplicated server. As the load increases, Eve's latency increases somewhat, until it finally spikes up at the saturation point, at a throughput of 1225 requests per second; the unreplicated server's latency spikes up at around 1470 requests per second. To keep its latency low while maintaining a high peak throughput, Eve uses a dynamic batching scheme: the batch size decreases when the demand is low (providing good latency), and increases when the system starts becoming saturated, in order to leverage as much parallelism as possible.

## 3.6 Conclusion

Eve is a new execute-verify architecture that allows state machine replication to scale to multi-core servers. By revisiting the role of determinism in replica coordination, Eve enables new SMR protocols that for the first time allow replicas to interleave requests nondeterministically and execute independently. This unprecedented combination is critical to both Eve's scalability and to its generality, as Eve can be configured to tolerate both omission and commission failures in both synchronous and asynchronous settings. As an added bonus, Eve's unconventional architecture can be easily tuned to provide low-cost, best-effort protection against concurrency bugs.

# Chapter 4

# Adam: Interacting Replicated State Machines

## 4.1 Introduction

Interaction between services is an integral part of today's computing. Services, large and small, are typically used in conjunction with other services to build large systems, like large-scale key-value stores [15, 25, 35], online shopping centers [4], data processing systems [24, 34], etc. In such environments, services frequently need to issue requests to other services. Such interactions take place both among services belonging to different domains, as well as among components of a single infrastructure service. For example, when an online store processes a client checkout, it issues a *nested* request—i.e., a request that is issued while the original checkout request is being processed—to the client's credit card service, asking for a certain amount of money to be blocked. Similarly, a web server processing client HTTP requests frequently needs to issue nested requests to a back-end database.

Infrastructure services follow a similar pattern; such services typically consist of multiple components that work together to provide a high-level service. For example, Figure 4.1 shows the components of Google's Photon

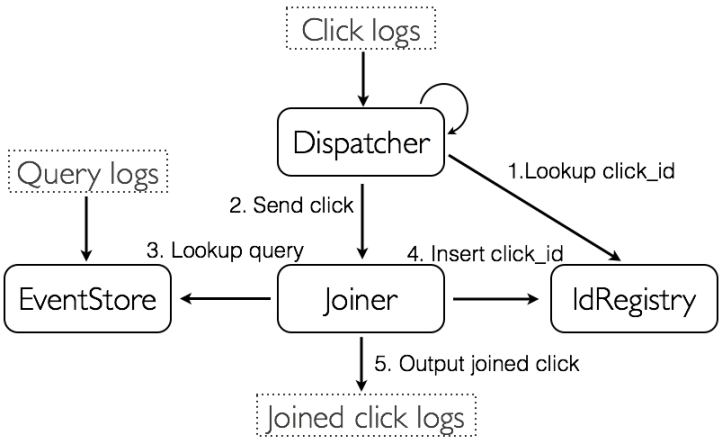


Figure 4.1: Components of Google's Photon system within a single datacenter.

system [5] for processing ad clicks—their main source of revenue. Photon consists of four main components, which regularly interact with each other to provide the required functionality. Similarly, the HBase key-value store consists of multiple components, some of which need to interact with as many as three other components in order to process client requests.

**The challenge** To make such services highly available, one might hope to use the tried abstraction of a Replicate State Machine. After all, if each of these services can be replicated so that it provides the abstraction of a single correct server, it should be easy to get these correct servers to interact with each other. In principle there is indeed nothing that prevents the RSM abstraction from being applicable in this setting. In practice, however, 40 years of applying RSM to the client-server model have led to lack of attention toward the challenges involved in allowing RSMs to communicate with one another efficiently. In particular, RSM implementations that use the client-server model assume that the replicated service can process requests *independently* and is therefore allowed to choose any strategy to process requests that does not violate the single-correct-server abstraction.

Fundamentally, maintaining this abstraction in an interactive setting

is harder than in the traditional client-server model, because the abstraction must be maintained towards *other services* as well; services that can be exposed—through the means of nested requests—to intermediate states of the replicated service.

Hence, execution of requests is no longer independent in this setting: the way in which requests are processed directly affects the states that are exposed to other services. Below, we describe the consequences of two popular execution modes that are used by most replication protocols to achieve replica convergence.

**Sequential execution** Most replication protocols require replicas to execute requests sequentially to ensure that correct replicas make the same state transitions and produce the same output. When services need to make nested requests to other services, however, the requirement of sequential execution prevents replicas from executing any other request while a nested request is in flight, forcing them to remain idle for long periods of time, causing a significant throughput reduction.

**Speculative execution** Some replication protocols employ speculation to achieve high throughput [40, 43]. For example, Eve speculatively executes requests in parallel, and then verifies whether replicas indeed arrived at the same state. When the speculation fails, the service must roll back its state to a consistent checkpoint. If, however, a service ($A$) had already sent some nested requests to another service $B$, the state of $B$ is now inconsistent, since it has executed requests that may never be issued when $A$ re-executes the requests that caused the misspeculation.

In this chapter we present Adam, a replication library that allows replicated services to interact with other services via nested requests. Adam's design focuses on addressing the performance repercussions of sequential execution—a requirement that lies at the heart of all active replication protocols[4]—

[4]Even in the execute-verify architecture, sequential execution is required to guarantee replica convergence, when speculation fails (see Section 3.4).

as well as the correctness consequences of speculative execution, which is employed by replication protocols such as Eve and Zyzzyva to achieve high performance [40, 43].

## 4.2 System model

Adam assumes a system model similar to Eve. Adam applies to both synchronous and asynchronous systems and can be configured to tolerate failures of any severity, from crashes to Byzantine faults.

Similar to Eve, in Adam we primarily target asynchronous environments where the network can arbitrarily delay, reorder, or lose messages without imperiling safety. For liveness, we require the existence of synchronous intervals during which the network is well-behaved and messages sent between two correct nodes are received and processed with bounded delay.

Each of the services participating in the Adam protocol can be configured separately, depending on their fault tolerance requirements; i.e., using their own $u$ and $r$ parameters. Each service is configured to be *live*, i.e., provide a response to client requests, despite a total of up to $u$ failures, whether of omission or commission, and to ensure that all responses accepted by correct clients are *correct* despite up to $r$ commission failures and any number of omission failures [18]. As in Eve, we assume that failures do not break cryptographic primitives; i.e., a faulty node can never produce a correct node's MAC.

## 4.3 The Adam protocol

The Adam replication protocol stands on the shoulders of many previous protocols [14, 18, 40, 46, 78]. At a high level, the basic operation of Adam is not different from these protocols: clients send requests to a designated *leader* replica, which forwards the requests to all other replicas. These replicas process the requests and send their replies back to the client. If there are

$max(u, r)+1$ matching replies, the client accepts the reply. One could choose to implement this high-level description with either of the two architectures we have described so far: the traditional agree-execute architecture or Eve's new execute-verify architecture. We choose to focus on the execute-verify architecture, since it encompasses both speculative and sequential execution—the latter to guarantee replica convergence, when speculation fails. Additionally, the execute-verify architecture can be used to emulate the behavior of the agree-execute architecture by employing sequential execution at all times; the verification phase effectively becomes an agreement on the order of requests. Section 4.3.1 discusses how Adam addresses the performance inefficiencies associated with sequential execution when services interact, and Section 4.3.2 discusses how Adam addresses the inconsistencies that arise from using speculative execution in this setting.

### 4.3.1 Deterministic pipelining

Adam employs deterministic pipelining as an alternative to sequential execution: its goal is to guarantee replica convergence without forcing replicas to remain idle while nested requests are in flight. The only assumption we make is that the leader replica organizes client requests in batches before forwarding them to other replicas of the same group, where each batch is an ordered sequence of requests. This assumption is trivial to satisfy in leader-based replication systems. While deterministic pipelining has the same goal as sequential execution—replica convergence—it does *not* provide the same end-to-end guarantees as sequential execution.

Previous systems which employed sequential execution [1, 14, 16, 21, 43, 46] typically provide *linearizability* of requests [36]. At first glance, linearizability may seem to be an integral part of the single-correct-node abstraction; while that may have been true in a world where execution was only sequential, it is no longer true when a single node can execute requests in parallel. Even in a world of multithreaded execution, however, linearizability is still desir-

able in some settings, because of the strong and intuitively simple semantics it provides. These strong semantics, however, come at a certain performance cost, which is often unnecessary. Enforcing linearizability at the level of the replication library appears to be yet another instance where the intricacies of implementation have seeped into the specification.

This seems a situation ripe for applying the end-to-end argument [64]: applications that require linearizability can enforce it end-to-end, but the underlying abstraction should not. Our deterministic pipelining mode of execution is a first attempt at freeing the abstraction from the obligation to provide overly restrictive properties, yielding a significant performance increase. Deterministic pipelining is based on a simple insight: *sequential execution, while sufficient to provide replica convergence, is not necessary*—even when the traditional agree-execute architecture is used. Sequential execution is merely an instance of a deterministic schedule; in fact, *any* deterministic schedule is sufficient to provide replica convergence.

Deterministic pipelining works as follows. Each replica is configured to have the same number of execution threads, $N$. When a replica receives a batch of requests, it starts executing the first request on the first thread. The execution of that thread will only be stopped in two cases: a) when it finishes executing a request and no other request is available in this batch for this thread; or b) after it sends a nested request to another service. When either of these *halting events* occur, the corresponding thread will be paused, the second thread will become active and will start executing the second request. Similarly, every thread will yield execution to the next thread when a halting event occurs. This process is repeated until the $N^{th}$ thread yields, at which point the first thread becomes active again, in a round-robin manner. Despite pipelining, the active thread may still remain idle for some time, while the response to its nested request is in flight. This idling can be reduced by increasing the depth of the pipeline. Note, finally, that when a thread finishes executing a request, it does not yield if another request is available; it starts executing that request.

Any thread that resumes execution is in one of two states, depending on the kind of halting event that caused it to yield. If the thread has reached the end of the batch, the thread is considered *stopped* and does not participate in the execution of this batch anymore—i.e. it yields immediately and is not considered for reactivation until the next batch starts. If the halting event was the sending of a nested request, then the thread waits until the response to that nested request arrives. If the response arrives before the thread becomes active, it is cached so that the thread can retrieve it immediately upon activation. Note that the order in which requests are processed does not depend on the timing of the responses, making the schedule deterministic across replicas.

### Avoiding deadlocks

An undesirable side-effect of enforcing this round-robin schedule among half-finished requests is that it raises the possibility of introducing deadlocks if some suspended threads hold exclusive locks when they yield execution. In Figure 4.2, the first request acquires a lock on $x$ and then yields to the second request, while still holding that lock. If the second request tries to acquire the same lock, a deadlock occurs: the first request will keep waiting for the second to yield, while the second request will keep waiting for the first to release the lock.

To address this side-effect, we impose an intermediate layer of control over the lock acquisitions. Specifically, we replace the `lock(x)` and `unlock(x)` calls with `trylock(x)` and `release(x)`, respectively. The pseudocode for `trylock` and `release` is shown in Figure 4.3. The use of `trylock` prevents deadlocks from being introduced by ensuring that, if a lock is already acquired, no other request will be blocked waiting for that lock to be released. Instead, that request will immediately yield control of the execution to the next request.

An immediate consequence of the above discussion is that failing to acquire a lock can cause a thread to yield. An unsuccessful `trylock` is then a halting event, just like sending a nested request and reaching the end of a request. Table 4.1 summarizes all halting events that will cause a thread to

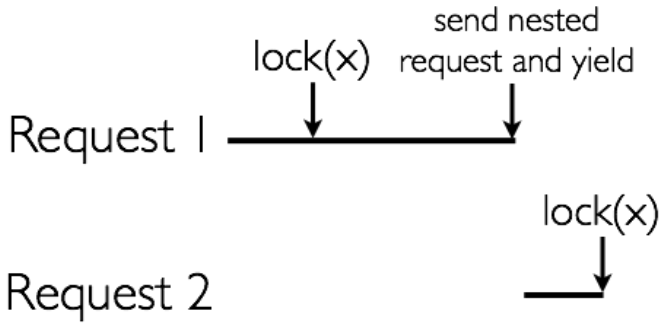


Figure 4.2: An example of how enforcing the deterministic pipeline schedule among half-finished requests can introduce deadlocks.

```
1   function trylock(x)
2           while acquireLock(x)=false
3                   yield()
4           return

6   synchronized function acquireLock(x)
7           if x.isAvailable
8                   lock(x)
9                   return true
10          else
11                  return false

13  function release(x)
14          unlock(x)
15          return
```

Figure 4.3: Pseudocode for the `trylock` and `release` function calls.

yield.

### 4.3.2 Taming speculation

As we discussed earlier, employing speculative execution in an environment where a service $A$ can send nested requests to another service $B$ can cause correctness violations. Consider two services $A$ and $B$, that use the Eve replication library. When speculation fails at $A$—because of non-matching tokens at the end of a batch—the state of $A$ must be rolled back; unfortunately, any nested requests that were sent as part of executing the batch have already

| Event name | Summary |
|---|---|
| end_batch | The request has executed fully and no more requests are available in this batch |
| send_nested_request | A nested request is sent to another service |
| fail_lock | The request tried to acquire a lock that is already acquired by another request |

Table 4.1: Events that cause a request to yield control of the execution to the next request.

been executed at $B$. To make matters worse, the effects of that execution may have been observed by other clients of $B$. As a result, even rolling back the state of $B$ may not be sufficient to repair this inconsistency.

The reason for such correctness violations is that nested requests are sent half-way through the execution of a request or, more specifically, at a time when the speculation is still unresolved: the requests are therefore contingent on the speculation succeeding.

We propose two approaches to address this problem: the first requires no modification to the backend service, while the second does require small modifications to the backend service in exchange for lower request latency.

**A1: resolve speculation** The first approach treats the sending of a nested request as an *output commit*. To prevent inconsistencies due to misspeculation, we resolve the speculation *before* the nested request is sent out. Resolving the speculation in the middle of executing a batch is, of course, not straightforward.

**A2: make speculation explicit** The second approach does not try to resolve the speculation before sending nested requests: instead, it simply ensures that the nested requests carry explicit information about the speculative state they depend on. This information can be used by service $B$ to determine whether enough replicas of $A$ agree on their speculative state. In this approach, service $A$ uses service $B$ as its verification stage.

Each approach has its own merits. A1 is transparent, since it does not require any changes to service $B$. As such, it is preferred when service $A$ interacts with several services, especially if those services are not replicated to begin with. A2 requires some modification at service $B$, but, as we describe below, it is more efficient; service $A$ does not need to perform an extra verification phase before sending nested requests to $B$.

**A1: resolve speculation**

In Eve, speculation can only be resolved at the end of the batch, which serves as a deterministically identifiable point at which a token of the state and responses of each replica can be computed and verified. This mechanism is obviously no longer sufficient for our case, as it resolves speculation after it has been exposed to other services. Note, however, that verification of convergence does not necessarily have to occur at the end of the batch: all that is required is that the verification point be deterministically identifiable across all correct replicas.

Fortunately, the end of the batch is not the only deterministically identifiable point in the execution. The sending of a nested request can serve the same purpose: nested requests are sent at the same point in the execution, across all replicas. Our goal is then to resolve speculation by leveraging the existence of those deterministically identifiable points.

The execution stage proceeds in the same way as in Eve with respect to batching and mixing. The primary execution replica collects client requests and forms batches of requests, which it forwards to all other execution replicas. Each replica then applies a deterministic and application-specific mixer, which partitions each batch into the same sequence of parallelBatches—groups of requests that the mixer believes can be executed in parallel with little likelihood that different interleavings will produce diverging results at distinct replicas. This is where things start diverging from the way Eve executes requests.

To aid in our description, we will again use the notion of halting events, similar to our deterministic pipelining technique. In this case, the halting

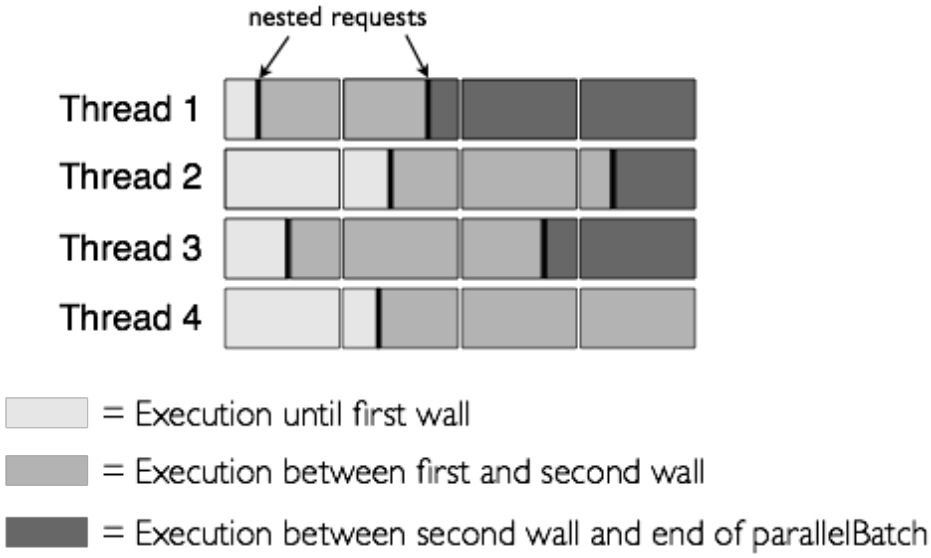


Figure 4.4: An example of four threads processing a parallelBatch. Thick vertical lines represent the sending of nested requests. The execution of the parallelBatch is divided by the two walls into three parts, each colored with an increasingly dark shade of gray. Note that the fourth thread does not execute any requests during the third part, since it has already reached the end of the parallelBatch.

events are two: the sending of a nested request and reaching the end of a parallelBatch.

Consider the execution of a parallelBatch of requests using $N$ execution threads. Initially, up to $N$ requests start being executed, one by each execution thread. When every one of these requests reaches its first halting event—either because it needs to send a nested request or because it just finished executing a request and there are no more requests in the current parallelBatch—we say that the execution has *hit a wall*. Note that, depending on how often requests trigger nested requests, the execution can hit a sequence of walls while processing a single parallelBatch. For example, in Figure 4.4 the execution hits two walls before reaching the end of the parallelBatch. When a thread hits the wall, its execution is paused until all threads hit the wall. When the halting

event is a send, execution is paused immediately before the nested request is sent. When all its $N$ threads hit the current wall, a replica calculates a token that represents its state and any responses made by requests in this parallel-Batch, using a mechanism similar to Eve, but with two notable differences. First, the token must also represent any nested requests that are about to be sent out, as these are also affected by the speculation in the execution. Second, the application state must also include the entire application stack—function calls and local variables—which is necessary to perform rollback and state transfer, as we discuss below.

When the token is calculated, the execution replicas, as in Eve, send it for verification: each replica sends a $\langle \text{VERIFY}, view, n, T, e \rangle_{\vec{\mu}_e}$ message to all verifier replicas, where $n$ is a sequence number that uniquely identifies the current wall. The verification replicas respond with the last agreed-upon token, along with the current view number; an increased view number indicates that no agreement was reached and a rollback is required. Upon receipt of $r + 1$ matching VERIFY-RESPONSE messages, an execution replica $e$ distinguishes three cases:

**Commit** If the view number has not increased and the agreed-upon token matches the one $e$ previously sent, then $e$ considers the execution up until $n$ to be committed. It can now send any nested requests that were about to be sent before execution was paused. After those nested requests have been sent, the execution replica resumes all threads. Any thread that was paused at a send halting event will resume execution right after that send. Note that threads do not need to receive the responses to nested requests synchronously; a thread can continue executing until it needs to access that response, at which point it will make a `readResponse()` call, which will block until the corresponding response becomes available.

Correct replicas will send their nested requests to the primary execution replica of $B$, the receiving service. That primary replica treats those requests as a single client request—i.e., it does not process that request

more than once—with the difference that it waits for a quorum of size $\max(u_A,r_A)+1$ of matching requests before it starts processing the request, where $u_A$ and $r_A$ are the configuration parameters of the sending service, $A$. Similarly, after $B$ has finished processing the request, correct execution replicas of $B$ will send the response to all execution replicas of $A$, which will in turn wait for a quorum of $\max(u_B,r_B)+1$ matching responses, where $u_B$ and $r_B$ are the configuration parameters of service $B$.

**Rollback** If the view number has increased, then agreement could not be reached. Replica $e$ discards any unexecuted requests and rolls back its state to the sequence number indicated by the token $T$, while verifying that its new state matches the token (else it initiates a state transfer). The main difference from Eve is that the replica does not necessarily roll back to the beginning of the parallelBatch, but rather to the last committed wall, which could correspond to some requests being partially executed. Rolling back to the middle of a request requires more than restoring the corresponding application state: one must restore the application stack that was active when the wall was reached. To that end, the replicas record the application stack as part of the application state.

As in Eve, the increased view number also implicitly rotates the execution primary. In Adam, however, if a batch has been partially executed the new primary does not propose new contents for that batch; replicas execute the remaining requests in that batch sequentially and then start receiving new batches from the primary. To ensure that a malicious primary cannot prevent the system from making progress indefinitely by sending a different batch of requests to different replicas, the batch contents are included in the state that is verified. That way, if a wall that corresponds to a partially executed batch is committed, replicas are guaranteed to have implicitly reached agreement on the contents of the partially executed batch; they can therefore always make progress by

executing requests within that batch sequentially.

**State transfer** If the view number has not increased, but the token does not match the one $e$ previously sent, then this replica has diverged from the agreed-upon state. To repair this divergence, the replica issues an incremental state transfer request to other replicas, as in Eve. The difference is that the transferred state contains also a) the application stack, since it is possible that the transferred state corresponds to a wall that includes partially executed requests; and b) the contents of the partially executed batch. The replica then restores not only the application state, but also the application stack and the requests that are yet to be executed as well. The details of how the application stack is recorded, rolled back and restored in our prototype are described in Section 4.4.

**Avoiding deadlocks** Similarly to deterministic pipelining, the above protocol has the potential of introducing deadlocks. Figure 4.5 demonstrates an example of such a deadlock. Requests 1 and 2 belong to the same parallelBatch and both try to acquire the same lock. If the first request acquires the lock, the second one will be blocked until the lock is released. The lock will never be released, however, since the first request must wait until the second request reaches the wall before proceeding to execute its second part, which would release the lock.

The reason why this type of deadlock is introduced lies in the fact that requests can send out nested requests while holding exclusive locks, thereby creating a two-way dependency with another request of the same parallelBatch that waits to acquire the same lock. We detect such deadlocks by using a timeout. When such a timeout occurs at replica $e$, we perform a rollback at $e$ and start re-executing requests in that parallelBatch sequentially. Note that other replicas may not necessarily roll back, too, as these deadlocks are subject to races and are not deterministically triggered. This could lead to a situation where the remaining replicas successfully reach the wall and produce enough matching tokens to commit that wall. In that case, $e$ will eventually

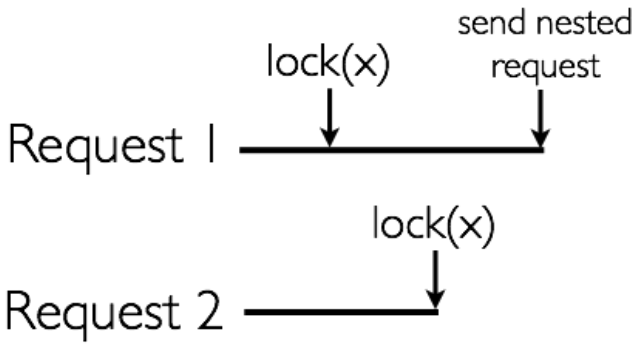


Figure 4.5: An example of how a deadlock can arise when two requests that belong to the same parallelBatch attempt to acquire the same lock.

realize—upon reception of the appropriate VERIFY-RESPONSE messages—that it has diverged from the other replicas and will request a state transfer from one of these replicas.

Fortunately, the root cause of these deadlocks—sending out nested requests while holding exclusive locks—is not very likely to occur in practice. Programmers typically avoid holding exclusive locks while performing time-consuming tasks, such as calls to remote services. Additionally, even when such exclusive locks are held for some requests, because of necessity or neglect, the mixer can be designed to avoid placing requests that might acquire the same locks in the same parallelBatch. We therefore believe that even a conservative timeout should be enough to deal with the rather uncommon circumstance where a request acquires a lock while making a nested request to another service, while another request in the same parallelBatch tries to acquire that same lock.

### A2: make speculation explicit

This optimization is based on the insight that sending a speculative nested request to another service is acceptable if that service can resolve the speculation on its own. Instead of resolving speculation before making an output

commit to another service, we allow such output commits to be speculative—but only if they carry an explicit description of the speculative condition on which they depend. This optimization can have a significant benefit with respect to request latency, since the middle service does not need to perform an extra round of verification before sending the nested requests; the backend service can perform that verification on behalf of the middle service.

The protocol for this approach proceeds just like A1 with respect to batching and mixing. We again use the notion of the wall, executing each request until it reaches a halting event, and calculating the application state token when all threads reach the wall. Instead of sending that token for verification, however, we simply include it in any nested requests that are sent to the backend service, $B$, along with the corresponding token for the previous wall. A nested request has the form $\langle \text{REQ}, \text{op}, t, c, e, T \rangle_{\vec{\mu}_{c,r}}$, where op is the operation that should be performed, $t$ is a timestamp that can be used instead of calls to `gettimeofday` to ensure convergence among replicas, $c$ is the client id, $e$ is an identifier for this replica, and $T$ is a token that contains a) the wall sequence number $n$; b) a checksum $s$ of the current state and generated responses; and c) a checksum $h$ of the state and responses generated for wall $n$-1.

The primary execution replica of $B$, just like in the first approach, gathers a quorum of $max(u_A, r_A)+1$ matching requests. If such a quorum is found, the execution replicas of $B$ process the requests taking the—by now familiar—steps of batching, mixing, execution, and verification. Recall, however, that the token sent for verification includes a checksum for both the current and the previous sequence number, to prevent inconsistent histories from being committed. In this approach, when the execution replicas of $B$ send a VERIFY message, they need to include the token they received from service $A$. The VERIFY message has therefore the following form: $\langle \text{VERIFY}, view, n, T_A, T_B, e \rangle_{\vec{\mu}_e}$.

The verification of service $B$, in addition to verifying that the $h$ checksum of $T_B$ matches the last committed token for $B$, must also perform the same check for the $h$ checksum of $T_A$. To do that, the verification stage of $B$

must always be aware of the last token committed by service $A$. To prevent a complicated protocol where the verification stages of $A$ and $B$ exchange information about which tokens are committed, we simply have service $A$ send all VERIFY messages to the verification stage of $B$, which must now store the last committed token and view number for both services. Hence, in this approach $A$ does not need to have its own verification replicas.

If the verification at $B$ is successful, the execution replicas of $B$ send the responses to the nested request to all execution replicas of $A$, including the token $T_A$ that was accepted and the corresponding view number. An execution replica of $A$ that receives a quorum of $max(u_B, r_B)+1$ such messages implicitly knows that $T_A$ is the verified token, and takes the appropriate action—commit or state transfer—depending on whether its own token matches $T_A$. If the verification at $B$ is unsuccessful, the verification replicas of $B$ send a VERIFY-RESPONSE message to all execution replicas of $A$ and $B$, causing both services to roll back to their last committed wall, respectively.

If the primary execution replica of $B$ receives $n_{E_A} - max(u_A, r_A)+1$ mutually non-matching requests, where $n_{E_A}$ is the number of execution replicas of $A$, the primary replica of $B$ sends a special VERIFY message to the verification stage, notifying it of the divergence at $A$. The verification stage then sends a VERIFY-RESPONSE message to all execution replicas of $A$, causing them to roll back to the last committed wall.

To avoid ignoring requests indefinitely because of a malicious primary replica, we take measures similar to Eve's. The verification replicas of $B$ propose a view change if the commit rate is not high enough. Additionally, the execution replicas of $A$, just like any client in Eve, resend nested requests to all execution replicas of $B$ if they receive no response within a timeout. Those replicas then forward the requests to the primary and expect it to include those requests in an upcoming batch within a given timeout; otherwise, they stop participating, eventually causing the verification replicas to perform a view change.

The approach of making speculation explicit can reduce request latency,

as it saves one verification phase and the corresponding messages between execution and verification. This reduced latency, however, comes at the cost of complexity, since the two services are no longer separate and independent, but must be designed and maintained as a whole. Hence, this approach is primarily targeting environments where both services are part of the same administrative domain.

## 4.4 Implementation

We have implemented a prototype of Adam in Java. The codebase of the Adam prototype descends from the Eve codebase, with modifications made in the parts where the design of Eve and Adam diverge. Most notably, these changes affect the following parts of the code.

**Replicated client** Since the middle service of Adam functions as a replicated client to the backend service, our prototype includes the necessary code to support quorum gathering of nested requests at the backend service, as well as quorum gathering of responses at the middle service.

**Nested request sequence numbers** A subtle implementation problem is that of assigning sequence numbers to nested requests. The middle service in Adam functions like a client for the backend service. As such, its requests must have a sequence number that increases by one with every new request. However, when multiple threads are executing in parallel, some of which need to issue nested requests and some do not, it becomes very hard, if not impossible, to assign those request sequence numbers in a deterministic way across all replicas. We therefore create a separate client for each of our $N$ execution threads. To ensure that each thread is assigned the same requests across all replicas, we no longer assign requests to threads in a first-come-first-serve fashion, but rather preassign requests to each thread before the parallelBatch is executed, using a round-robin algorithm. Using this approach, each thread

can independently generate request sequence numbers that increase by one with each new request and are consistent across all replicas.

**The wall, rollback, and state transfer** The hardest part of the Adam prototype is the implementation of the wall, the ability to roll back to a previous wall, and to implement state transfer that can bring another replica to the correct wall. To implement rollback in Eve, it is sufficient to undo the uncommitted changes to the application state, since the execution will restart at the beginning of the batch. Instead, in Adam the execution will have to restart at the last committed wall, which usually involves resuming the execution of some half-finished requests. It is therefore not sufficient to roll back the application state; local variables as well as the entire stack must be reset appropriately.

To implement such functionality, we use the notion of a *continuation*: a data structure that represents the computational process at a given point in the process's execution. Unfortunately, Java does not have native support for continuations. Instead, we use the open source Javaflow library, which provides the basic continuation functionality. Specifically, the Javaflow API includes two calls, Suspend() and ContinueWith(). When a thread calls Suspend(), Javaflow starts iterating through all stack frames and saving them into a continuation object, stopping only when it reaches the stack frame where ContinueWith() is called; and finally returning the continuation object. ContinueWith() takes a continuation object as an argument and causes the thread to resume execution from that continuation, by restoring all stack frames in the continuation and resuming execution from the last instruction after the Suspend() was called.

Adam uses this continuation functionality in the following way: when a new parallelBatch is started, each thread calls ContinueWith(), to mark the beginning of a "clean" execution. Whenever a thread hits the wall, it calls Suspend() which returns to where the initial ContinueWith is called and returns the appropriate continuation object. At this point, the main *coordinator*

thread waits for all execution threads to call Suspend (i.e., hit the wall), and then starts calculating the token that will be send for verification. Depending on the verification results, the coordinator thread will ask each execution thread to call ContinueWith using the appropriate continuation object as its argument: the one returned by the last Suspend, if verification succeeds; or the one returned by the previous Suspend()—or null if no such Suspend exists—if the verification failed.

While Javaflow provides us with a basic continuation functionality, it is not sufficient for implementing state rollback to a given wall. In particular, adapting Javaflow to fit our needs raised two problems. First, the Javaflow continuations only store shallow copies—i.e., references—to application objects. Hence, if an object is modified between walls $n-1$ and $n$, then simply rolling back to the previous $(n-1)$ version of the reference will not actually roll back the modifications to the object. Instead, in Adam we combine the continuation functionality with the Merkle tree implementation, keeping a deep copy of the application objects that are referenced by a continuation. Keeping deep copies that are versioned through the Merkle tree module has the additional benefit that it simplifies the implementation of state transfer. When a state transfer is required, the replica sends a serialized version of the appropriate continuation, which is sufficient—since it contains deep copies of the required objects—to bootstrap the requesting replica. The second problem we met is that the stack keeps references to *all* objects used in the execution. Remember that in Eve we discussed the need to abstract away parts of the state that might not be identical byte-wise, even though their respective replicas have not diverged. Such state includes the state used by the replication library itself—e.g., the replica id, which is by definition different across all replicas—local state like IP address, or timestamps that may be used for logging. Adam takes an approach similar to Eve: we manually annotate such objects to prevent them from being included in the hash calculation of the state.

Our current prototype omits some of the features described above. Specifically, it does not implement approach A2 of making speculation explicit;

our experiments in Section 4.5 are performed using approach A1. Additionally, our prototype does not implement the deadlock recovery mechanisms; we have not yet found need for this mechanism, as the applications we have experimented with do not acquire locks while nested requests are in flight.

## 4.5 Evaluation

Our evaluation of Adam aims to answer the following questions:

- What is the throughput gain of using deterministic pipelining compared to a traditional sequential execution?
- How does Adam's performance when using speculative execution compare to sequential execution?

We address these questions by using the key-value store application we presented in the evaluation of Eve (Section 3.5). In our experiments clients send requests to a key-value store $A$. Each request requires some amount of execution time, modifies one key-value pair, and makes a nested request to a back-end key-value store, $B$. The purpose of these experiments is to understand the potential benefit of using deterministic pipelining and speculation instead of sequential execution. Hence, our focus is mostly on the amount of computation required by each requests, rather than the nature of computation—which could be anything from performing queries in a database to accessing files on a file server.

We run our experiments on a testbed with 12x 4-core Intel Xeon @2.4 GHz and 3x 16-core AMD Opteron @3.0 GHz, connected with a 1 Gb Ethernet. We use the AMD machines as the execution nodes of the middle service and use the Intel machines as clients, execution replicas for the back-end service, and verification replicas for the middle and back-end service. Both services are configured with $u = r = 1$, and have therefore three execution and four verification replicas each. Due to the limited size of our testbed, we colocate

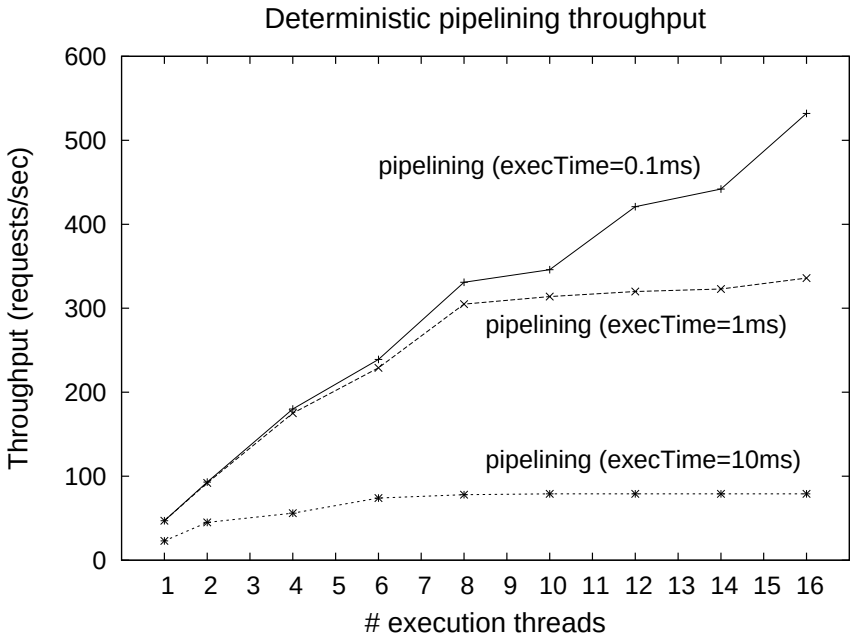


Figure 4.6: The throughput of Adam using deterministic pipelining.

one verification replica of each service on the same machine. This does not affect our results, however, since verification is very lightweight compared to execution and does not introduce a performance bottleneck.

### 4.5.1 Deterministic pipelining

Despite its simplicity, deterministic pipelining can be significantly faster than sequential execution. Figure 4.6 shows the throughput of Adam using deterministic pipelining and for requests that require execution times of 0.1 ms, 1 ms, and 10 ms. The performance of sequential execution is equal to the performance of deterministic pipelining when only one thread is used. As expected, in all three cases Adam's throughput increases as more execution threads become available: a deeper pipeline means that the middle service

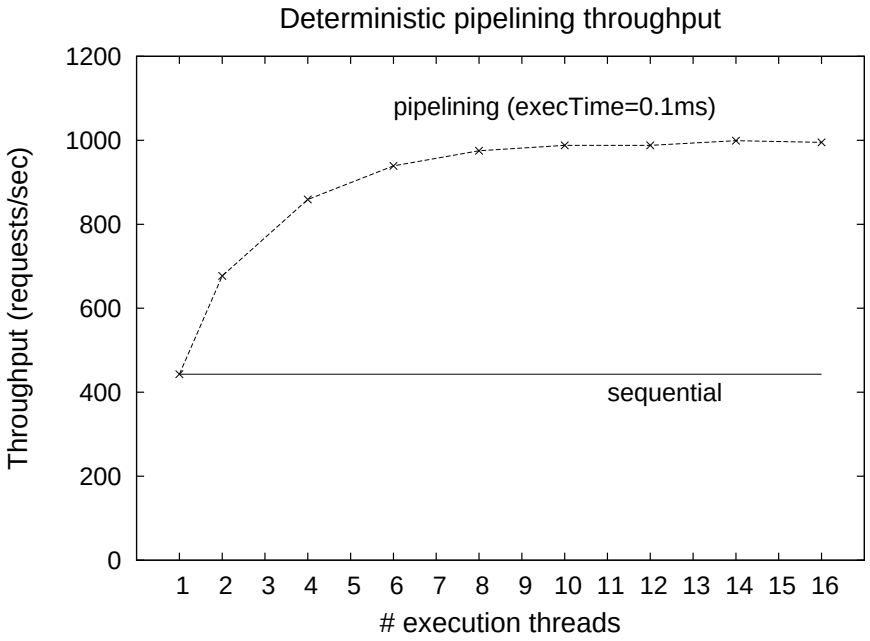


Figure 4.7: The throughput of Adam using deterministic pipelining when the back-end service is optimized to use a small time interval for batching. The execution time of each request is 0.1ms.

remains idle for smaller intervals, masking the delay of previously sent nested requests. When requests are lightweight (e.g., 0.1 ms of execution time), the importance of deterministic pipelining becomes more pronounced: a system bound by sequential execution ends up spending most of its time—about 95%—idling, waiting for responses to its nested requests.

To understand why the middle service may have to wait that long, we must take a closer look at the way requests are processed by the back-end service. To mitigate the costs of performing agreement, replicated services perform *batching* of requests: the primary waits until it has received a certain amount of client requests or a predefined time interval has elapsed, before sending those requests to other replicas for agreement or execution, depending

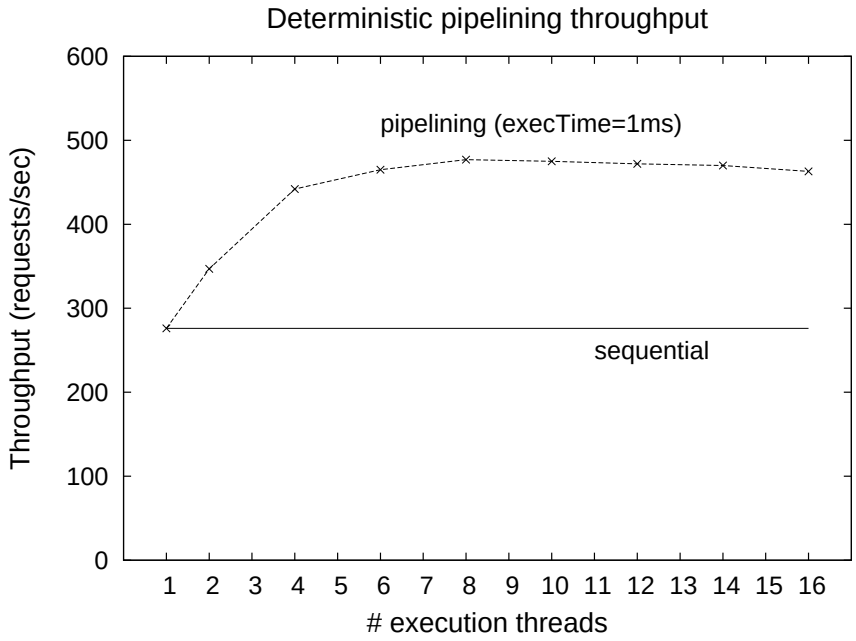


Figure 4.8: The throughput of Adam using deterministic pipelining when the back-end service is optimized to use a small time interval for batching. The execution time of each request is 1ms.

on the architecture. If the middle service is the only client to the back-end service, then the back-end service will end up waiting for that time interval to elapse, causing significant delays in the processing of nested requests. In the experiments of Figure 4.6 that interval is set to 20 ms, which is the default value used in UpRight [18] and Eve.

When the middle service is the only client to the back-end service, one can tune this time interval, to reduce idle time at the middle service. Figures 4.7, 4.8, and 4.9 show the throughput of deterministic pipelining when that time interval is reduced to 1 ms, for execution times of 0.1 ms, 1 ms, and 10 ms, accordingly. As expected, the speedup of deterministic pipelining is significantly reduced in all these cases—about twice the throughput of sequential

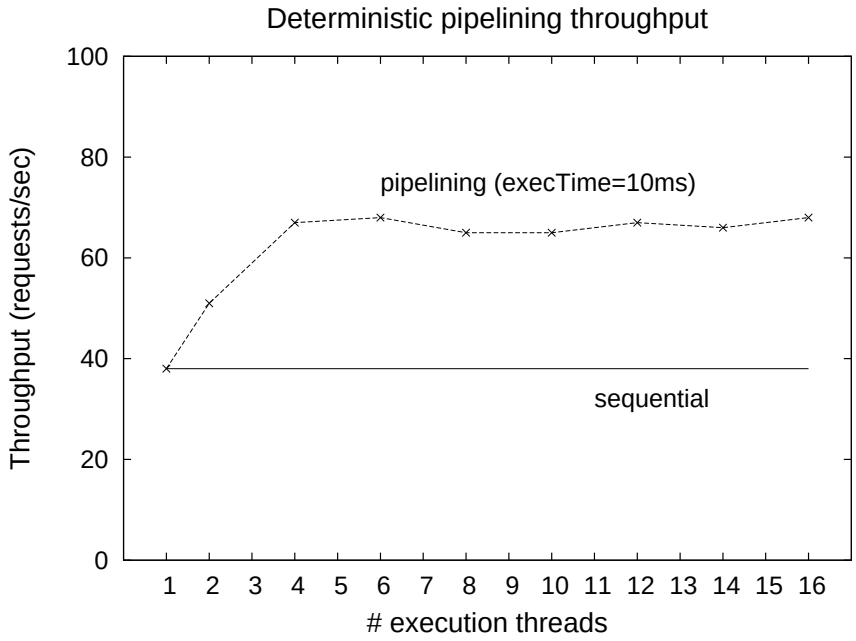


Figure 4.9: The throughput of Adam using deterministic pipelining when the back-end service is optimized to use a small time interval for batching. The execution time of each request is 10ms.

execution—since, even when using sequential execution, the middle service does not have to remain idle for quite as long. Once again, the benefit of pipelining is more pronounced for lightweight requests, ranging from a 1.8x speedup over sequential execution when the request execution time is 10 ms, to a 2.2x speedup when the request execution time is 0.1 ms.

Note, however, that there are many cases where tuning the parameters of the back-end service is impossible or undesirable. For example, the back-end service might belong to a different administrative domain, or it could be that our middle service is not its only client. Especially when the back-end service is operating close to saturation, reducing the amount of batching performed in that service could be detrimental to its throughput. In fact, such a reduction

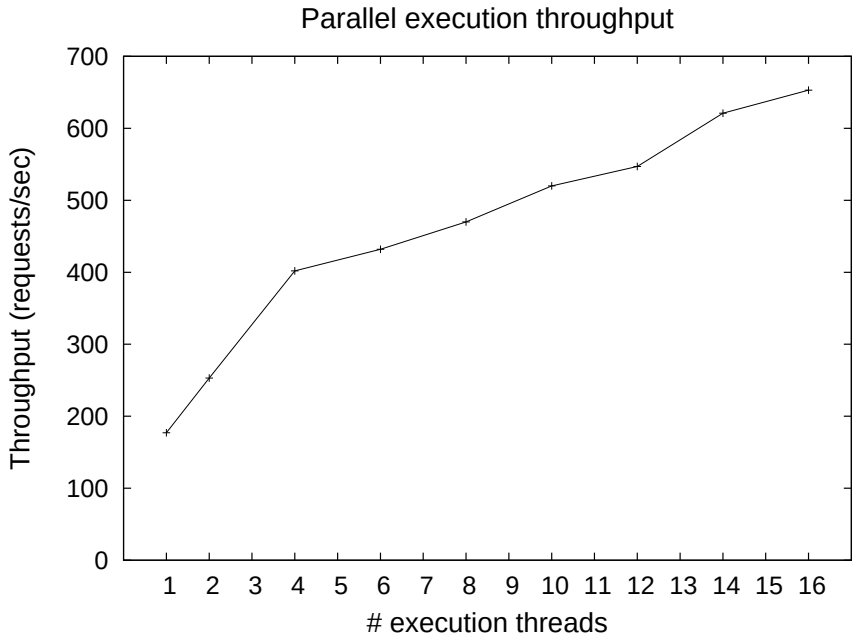


Figure 4.10: The throughput of Adam using parallel execution. The execution time of each request is 1ms.

would drive that service past its saturation point, greatly increasing the latency of requests and in turn hurting the throughput of our middle service as well.

### 4.5.2 Speculative execution

Our final experiment demonstrates the throughput of Adam when employing parallel execution. In this setting Adam uses approach A1, where speculation is resolved—by calculating the application token and performing verification—before sending a nested request. Figure 4.10 shows the results of this experiment. Notice that as more threads become available, the performance of Adam increases, since more requests can be executed in parallel. The performance increase compared to single-threaded execution is about 3.7x. We believe that there are some steps one can take to reach even higher performance. For ex-

ample, while parallel execution is important to increase system throughput, pipelining of requests is still crucial, even though sequential execution is not used. When the execution threads hit the wall, they currently remain idle while waiting for the response to their nested requests. We believe that a combination of the two techniques we propose in this chapter—deterministic pipelining and the wall—should achieve even higher performance than any of those two techniques in isolation.

## 4.6 Conclusion

Adam is a new replication library that allows replicated services to be deployed in environments where they have to interact with other services. Previous replication protocols based on the Replicated State Machine abstraction assume a client-server model, which, as we showed, can have significant performance limitations and consistency problems in interactive settings. In Adam, we trace the cause of these issues to the use of sequential and speculative execution, and propose new techniques that address these issues, allowing fast and consistent replication of services in this setting.

# Chapter 5

# Related work

## 5.1 Replicating multithreaded services

The end goal of Eve is to allow a replicated service to maintain correctness while executing requests in parallel. While we believe that an execute-verify replication architecture is the most fitting way to achieve this goal, it is certainly not the only way. In this section we discuss other alternatives to achieving multithreaded replication and review other work that is related to Eve.

### 5.1.1 Deterministic Multithreading

Deterministic execution of multithreaded programs [6, 7, 9, 51] guarantees that, given the same input, all correct replicas of a multithreaded application will produce identical internal application states and outputs. Although at first glance this approach appears a perfect match for the challenge of multithreaded SMR on multi-core servers, there are two issues that lead us to look beyond it. The first issue [8] is straightforward: current techniques for deterministic multithreading either require hardware support [26, 27, 37] or are too slow (1.2x-10x overhead) [6, 7, 9] for production environments. The second issue originates from the semantic gap that exists between modern SMR protocols and the techniques used to achieve deterministic multithreading.

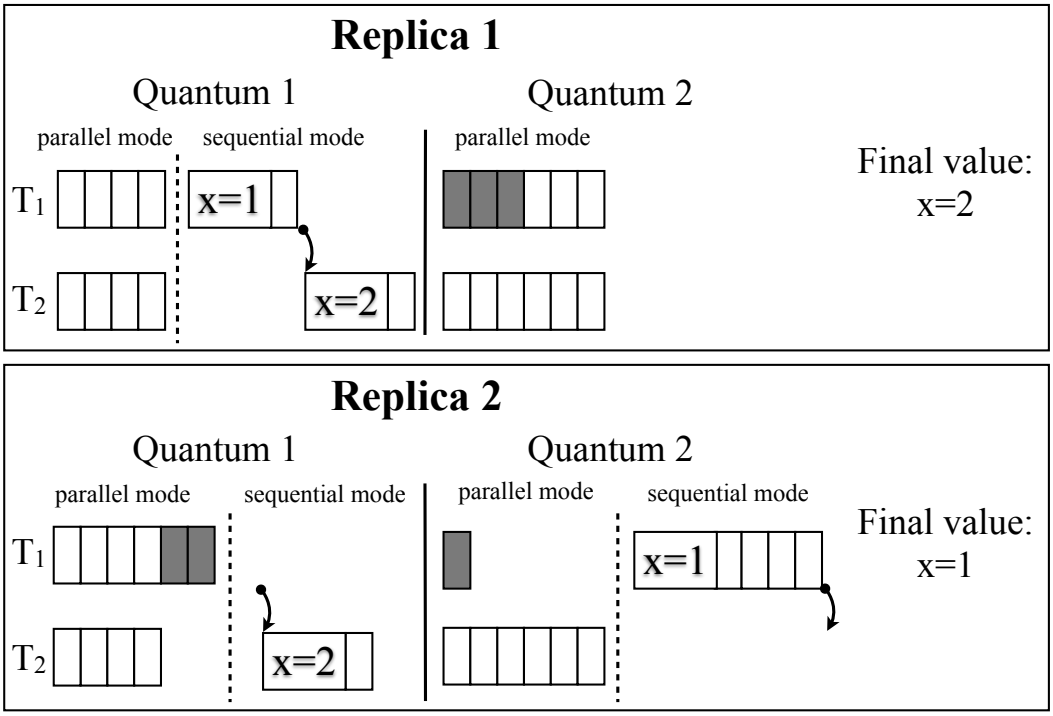


Figure 5.1: An example where ordering read-only requests (depicted as shaded rectangles) differently at different replicas can lead to state divergence. The replicas are using the DMP-O algorithm. The initial ownership status of variable $x$ is "shared" and the quantum size is 6.

Seeking opportunities for higher throughput, SMR protocols have in recent years looked for ways to exploit the semantics of the requests processed by the replicas to achieve replica coordination without forcing all replicas to process identical sequences of inputs. For example, many modern SMR systems no longer insist that read requests be performed in the same order at all replicas, since read requests do not modify the state of the replicated application. This *read-only optimization* [14, 18, 42] is often combined with a second optimization that allows read requests to be executed only at a *preferred quorum* of replicas, rather than at *all* replicas [38]. Several SMR systems [21, 74] use the preferred quorum optimization during failure-free executions also for requests that change the application's state, asking other replicas to execute these requests only if a preferred replica fails.

Unfortunately, deterministic multithreading techniques know nothing of the semantics of the operations they perform. Their ability to guarantee replica coordination of multithreaded servers is based purely on syntactic mechanisms that critically rely on the assumption that all replicas receive identical sequences of inputs: only then can deterministic multithreading ensure that the replicas' states and outputs will be the same. Read-only optimizations and preferred quorum operations violate that assumption, leading correct replicas to diverge. For instance, read-only requests advance a replica's instruction counter and may cause the replica to acquire additional read locks.

Figure 5.1 illustrates an example where executing a set of read-only instructions in a different order at different replicas can cause their state to diverge. In this example, replicas use the DMP-O deterministic multithreading algorithm [7], with a quantum size of 6. The shaded instructions represent the instructions of a read-only request. DMP-O uses instruction counting to guarantee that each thread eventually leaves the sequential mode, even if it does not need to synchronize with other threads. In this example, the read-only requests are executed *after* the request $[x = 1]$ at Replica 1, but *before* that request at Replica 2. The presence of the read-only requests before $[x = 1]$ at Replica 2 ends the quantum and "pushes" the request $[x = 1]$ to the next

quantum. This leads to different replicas serializing the $[x = 1]$ and $[x = 2]$ requests in different ways, which causes the final value of $x$ to diverge.

Paradoxically, the troubles of deterministic replication stem from sticking to the letter of the state machine approach [45, 66], at the same time that modern SMR protocols have relaxed its requirements while staying true to its spirit.

### 5.1.2 Transactional processing systems

Vandiver et al. [71] describe a Byzantine-tolerant semi-active replication scheme for transaction processing systems. Their system supports concurrent execution of queries but its scope is limited: it applies to the subset of transaction processing systems that use strict two-phase locking (2PL). A recent paper suggests that it may be viable to enforce deterministic concurrency control in transactional systems [68], but the general case remains hard. Kim et al. [41] recently proposed applying this idea to a transactional operating system. This approach assumes that all application state is manageable by the kernel and does not handle in-memory application state.

### 5.1.3 Semi-active replication and record-replay

One alternative is to use a replication technique other than state machine replication. *Semi-active replication* [60] weakens state machine replication with respect to both determinism and execution independence: *one* replica, the primary, executes nondeterministically and logs all the nondeterministic actions it performs. All other replicas then execute by deterministically reproducing the primary's choices. In this context, one may hope to be able to leverage the large body of work on deterministic multiprocessor replay [3, 22, 28, 49, 58, 59, 63, 72, 75, 76]. Unfortunately, relaxing the requirement of independent execution makes these systems vulnerable to commission failures. Also, similar to deterministic multithreaded execution approaches, record and replay approaches assume that the same input is given to all replicas. As discussed

in Section 5.1.1 this assumption is violated in modern replication systems.

Rex [32] recently proposed an alternative architecture for replicating multithreaded services, based on the notion of deterministic replay. Instead of the traditional agree-execute architecture, or Eve's execute-verify, Rex proposes a new execute-agree-follow architecture, which has a primary replica executing requests, recording the nondeterministic events during that execution—called a trace—including dependencies among synchronization events. After the execution has finished, replicas run a consensus protocol to agree on the trace that was executed by the primary. Finally, non-primary replicas execute those requests, reproducing the nondeterministic decisions of the primary, to guarantee convergence.

Rex represents a different point in the design space than Eve. Since replicas no longer execute independently, Rex cannot tolerate commission failures. Also, Rex makes different assumptions than Eve when it comes to guaranteeing correctness. In the execute-verify architecture, correctness depends on being able to identify the relevant application state. In the execute-agree-follow architecture, instead, correctness hinges on accurately capturing all sources of nondeterminism, like data races and synchronization events.

### 5.1.4 Passive primary-backup

Remus [23] is a typical example of a passive primary-backup system. Remus takes a different approach to replicating multithreaded services: the backup does not execute requests, but instead passively absorbs state updates from the primary: since execution occurs only at the primary, the costs and difficulty of coordinating parallel execution are sidestepped. These advantages however come at a significant price in terms of fault coverage: Remus can only tolerate omission failures—all commission failures, including common failures such as concurrency bugs, are beyond its reach. Like Remus, Eve neither tracks nor eliminates nondeterminism, but it manages to do so without forsaking fault coverage; further, despite its stronger guarantees, Eve outperforms Remus by

a factor of 4.7x and uses two orders of magnitude less network bandwidth (see Section 3.5.5) because it can ensure that the states of replicas converge without requiring the transfer of all modified state.

### 5.1.5 Speculative systems

One of the keys to Eve's ability to combine independent execution with nondeterministic interleaving of requests is the use of the mixer, which allows replicas to execute requests concurrently with low chance of interference. Kotla et al. [44] use a similar mechanism to improve the throughput of BFT replication systems. However, since they still assume a traditional agree-execute architecture, the safety of their system depends on the assumption that the criteria used by the mixer never mistakenly parallelize conflicting requests: a single unanticipated conflict can lead to a safety violation.

Both Eve and Zyzzyva [42] allow speculative execution that precedes completion of agreement, but the assumptions on which Eve and Zyzzyva rest are fundamentally different. Zyzzyva depends on correct nodes being deterministic, so that agreement on inputs is enough to guarantee agreement on outputs: hence, a replica need only send (a hash of) the sequence of requests it has executed to convey its state to a client. In contrast, in Eve there is no guarantee that correct replicas, even if they have executed the same batch of requests, will be in the same state, as the mixer may have incorrectly placed conflicting requests in the same parallelBatch.

We did contemplate an Eve implementation in which verification is not performed within the logical boundaries of the replicated service but, as in Zyzzyva, it is moved to the clients to reduce overhead. For example, a server's reply to a client's request could contain not just the response, but also the root of the Merkle tree that encodes the server's state. However, since agreement is not a bottleneck for the applications we consider, we ultimately chose to heed the lessons of Aardvark [16] and steer away from the corner cases that such an implementation would have introduced.

### 5.1.6 Finding concurrency bugs

Concurrency bugs are notoriously hard to find. Several tools exist that try to identify as many concurrency bugs as possible, without introducing a large number of false positives. Many of these tools focus on identifying concurrency bugs that cause some immediate or severe system failure (e.g. a crash) [13, 57, 79]. These tools, however, are rather limited in scope, as concurrency bugs frequently do not have any such immediate side-effects [31].

Pike [30] is a tool for finding such elusive concurrency bugs. Specifically, it tries to identify two classes of bugs: *semantic* bugs, which manifest as any violation of the application semantics, such as returning an incorrect result to the user; and *latent* bugs, which silently corrupt internal data structures and can potentially manifest much later, when they are triggered by a subsequent input. To identify these bugs, Pike tests whether the parallel execution of a set of requests matches—in terms of both outputs and application state—some sequential execution of those requests.

Pike resembles Eve in that they are both concerned with preventing silent corruption of the application state. Hence, both systems require some programming effort in identifying the relevant application state and abstracting away the irrelevant parts.

Other than that, though, Pike's goal is fundamentally different from Eve's: to identify the source of concurrency bugs, rather than to mask them, as Eve does. Additionally, Pike's approach is based on the assumption that even complex executions provide semantics that are reasonably close to linearizability [36], which introduces false positives. Eve, instead, does not make any such assumptions, since it does not need to detect the source of the bug—or even to reason about what the applications semantics are. Eve simply uses the redundancy that is inherent in replicated systems to mask concurrency bugs, as long as they do not manifest in the same way across all replicas.

### 5.1.7 Scaling State Machine Replication

There is a lot of work on improving various aspects of the performance of State Machine Replication. Eve focuses on achieving high performance by executing requests in parallel; as such, it focuses on environments where request execution is the performance bottleneck. Eyrie [10] has a similar goal: to achieve scalability of the execution of requests by partitioning the application state. Despite this partitioning, Eyrie guarantees linearizability, while allowing commands to access any combination of partitions.

Other works have tried to scale the performance of State Machine Replication when the bottleneck lies in the agreement (or ordering) phase [39, 53–56]. These works typically split the client requests into a number of substreams of request, which are then ordered separately and then joined as required, before being executed.

## 5.2 Interacting Replicating State Machines

Service interaction is an integral part of building large distributed systems. When faced with such interactions, previous works have proposed various solutions; some to particular instances, others more general. In this section we review previous work on service interaction and comment on the relation to Adam.

**Replicated Remote Procedure Call** A number of early works provide the functionality of a replicated—or fault tolerant—Remote Procedure Call (RPC) [19, 20, 77]. These approaches typically provide mechanisms to ensure that the receiving service will perform the requested procedure exactly once, and that each replica of the sending service will receive the results of the RPC, while providing transparency to the programmer, making Replicated RPCs look like normal procedure calls. These approaches, however, predate modern replication protocols and therefore do not consider the consequences of

those protocols' design choices. In particular, they do not address the performance limitations of sequential execution, and, since they predate speculative protocols, such as Zyzzyva and Eve, do not consider the possibility that the RPC of one service might be based on speculative state that can not be naively exposed to other services.

**Optimizing procedure calls among services** More recently, Song et al. proposed RPC Chains, a technique that allows a number of interacting services to optimize complex patterns of RPCs, by composing multiple such remote calls into a single path that weaves its way through all the required services. This technique aims to reduce latency by eliminating the requirement that an RPC always returns to the caller before the next RPC is called. Although operating in a similar setting, where multiple services interact to provide some high-level functionality, Adam and RPC Chains have very different goals. Adam's main goal is to allow such interactions for replicated services, while RPC Chains only targets singleton services.

### 5.2.1 Interaction among Replicated Services

Replication has been used in many previous works to enhance the dependability of a certain service. When that service needs to interact with other services or components, typically a custom protocol is used to regulate this interaction. For example, in Salus [73] a replicated region server needs to issue requests to a replicated storage layer. The resulting protocol is quite complicated, despite the fact that the region server does not employ speculative execution. Similarly in Farsite [2], groups of replicated nodes can issue requests to other such groups. To simplify the protocol for those interactions, Farsite groups communicate through message passing and avoid using nested calls altogether. As such, it does not address the complications most services would face in such settings. Adam, instead, tries to provide a general solution that transparently allows each layer to provide the abstraction of a single correct server, thereby facilitating the interaction between replicated services.

# Chapter 6

# Conclusion

This dissertation rethinks the design of replicated services in the era of multi-core computers and large cloud infrastructures. In particular, it presents the design and implementation of Eve, a replication system that is based on a new execute-verify architecture; and Adam, a replication library that allows replicated services to be fast and correct, even in environments where they must interact with other services. In this dissertation we make the following contributions.

In Chapter 3, we take a closer look at the Replicated State Machine abstraction and refine its specification, removing the unnecessary strengthening that had seeped into it: the assumption of sequential execution. We believe that this refined specification expresses the essence of State Machine Replication, free from implementation details.

We rethink the architecture of active replication protocols, taking into account the need for multithreaded execution. Our insight is that it does not make sense to agree on the order of requests before they are executed, since multithreaded execution will undo that order, anyway. We therefore submit that the traditional agree-execute architecture is not well-suited to supporting multithreaded execution of requests. Instead, we propose a new execute-verify architecture, which has replicas first executing requests in parallel, and then trying to reach agreement; not on the order of requests, but on the outcome

of the execution, namely the states and responses produced by the replicas.

We describe the design and implementation of Eve, a replication library designed for the execute-verify architecture. To allow this architecture to be efficient in practice, Eve takes the following steps. To make divergence uncommon, it introduces the mixer, an application-specific heuristic that tries to identify commutative requests. The mixer is not assumed to be perfect; it may occasionally allow non-commutative requests to execute in parallel, causing replicas to diverge. To check for such divergence efficiently, Eve organizes the application state in a Merkle tree, which allows for rapid and incremental computation of a hash of the entire state. Finally, to efficiently recover from divergence, Eve uses a versioned copy-on-write mechanism, allowing for rapid rollback and incremental state transfer.

In Chapter 4, we rethink our replication mechanisms in environments where services communicate with other services. We show that two popular execution modes, sequential and speculative execution, raise performance and consistency concerns, respectively, when we move away from the traditional client-server model.

To address the performance limitations of sequential execution in this setting, we apply the end-to-end argument to replication, no longer insisting that the replication library provide linearizability of requests. Instead, we propose deterministic pipelining, a simple alternative to sequential execution that guarantees replica convergence while not being subject to the performance limitations of sequential execution.

To allow the use of speculative execution without creating inconsistencies among services, we propose two techniques that allow speculation to be used within a service, albeit in a controlled manner. The first technique is transparent, resolving speculation before it is exposed to another service, while the second trades transparency for lower latency, ensuring that nested requests explicitly denote the speculation they depend on.

Through a combination of a novel architecture, protocols and mechanisms, Adam and Eve offer replicated services the necessary tools to meet

the increasing demands of today's computing world, in terms of both performance and complexity. Just like their biblical namesakes, Adam and Eve are expected to have descendants: there is still a long way to go and several problems to be addressed; chief among them is the challenge of providing efficient replication solutions that handle all types of failures, without requiring manual modification of the application code. This challenge is hard as it requires a combination of independent execution and the ability to compare application states in an automated fashion.

# Bibliography


[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, October 2005.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th OSDI*, pages 1–14, December 2002.

[3] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[4] Amazon. http://www.amazon.com.

[5] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, 2013.

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

[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, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails? In *2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.

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

[10] Eduardo Bezerra, Fernando Pedone, and Robbert van Renesse. Scalable state-machine replication. In *DSN*, Atlanta, GA, June 2014.

[11] N. Budhijara, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, 2nd edition, 1993.

[12] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In *CDCCA*, 1992.

[13] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGARCH Comput. Archit. News*, 38(1):167–178, March 2010.

[14] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[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*, pages 15–15, 2006.

[16] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[17] Allen Clement. *UpRight Fault Tolerance*. PhD thesis, The University of Texas at Austin, December 2010.

[18] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. UpRight cluster services. In *SOSP*, 2009.

[19] Eric C. Cooper. Replicated procedure call. In *PODC*, pages 220–232, 1984.

[20] Eric C. Cooper. Replicated distributed programs. In *SOSP*, pages 63–78, 1985.

[21] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

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

[23] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[26] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[27] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. In *ASPLOS*, 2011.

[28] George Dunlap, Dominic Lucchetti, Michael Fetterman, and Peter Chen. Execution replay for multiprocessor virtual machines. In *VEE*, 2008.

[29] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD, April 2006.

[30] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Eurosys*, 2011.

[31] P. Fonseca, Cheng Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *DSN*, 2010.

[32] Zhenyu Guo, Chuntao Hong, Mao Yang, Lidong Zhou, Li Zhuang, and Dong Zhou. Rex: Replication at the speed of multi-core. In *Eurosys*, 2014.

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

[34] Hadoop. http://hadoop.apache.org/core/.

[35] Hbase. http://hbase.apache.org/.

[36] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[37] D.R. Hower, P. Dudnik, M.D. Hill, and D.A. Wood. Calvin: Deterministic or not? Free will to choose. In *HPCA*, 2011.

[38] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX*, 2010.

[39] Manos Kapritsos and Flavio P. Junqueira. Scalable agreement: Toward ordering as a service. In *HotDep*, 2010.

[40] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *OSDI*, October 2012.

[41] Sangman Kim, Michael Z. Lee, Alan M. Dunn, Owen S. Hofmann, Xuan Wang, Emmett Witchel, and Donald E. Porter. Improving server applications with system transactions. In *EuroSys*, 2012.

[42] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[43] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin. Zyzzyva: speculative byzantine fault tolerance. *Communications of the ACM*, November 2008.

[44] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *DSN*, 2004.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[46] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

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

[48] Leslie Lamport and Mike Masa. Cheap paxos. In *Proc. DSN-2004*, pages 307–314, June 2004.

[49] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In *ASPLOS*, 2010.

[50] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. Detecting failures in distributed systems with the Falcon spy network. In *SOSP*, 2011.

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

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

[53] Parisa Jalili Marandi, Eduardo bezerra, and Fernando Pedone. Rethinking state-machine replication for parallelism. In *ICDCS*, 2014.

[54] Parisa Jalili Marandi and Fernando Pedone. Optimistic parallel state-machine replication. In *SRDS*, 2014.

[55] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *DSN*, 2012.

[56] Parisa Jalili Marandi, Marco Primi, Nicholas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *DSN*, 2010.

[57] M. Musuvathi, S. Qadeer, T. Bell, G. Basier, P. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. 2008.

[58] Josep Torrellas Pablo Montesinos, Luis Ceze. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA*, 2008.

[59] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessor. In *SOSP*, 2009.

[60] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4. *ACM OSR*, 1991.

[61] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.

[62] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: using abstraction to improve fault tolerance. In *SOSP*, 2001.

[63] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM TCS*, 1999.

[64] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[65] F. B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[66] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 1990.

[67] Sun Microsystems, Inc. Memory management in the Java HotSpot virtual machine, April 2006.

[68] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[69] TPC-W. Open-source TPC-W implementation. http://pharm.ece.wisc.edu/tpcw.shtml.

[70] Transaction Processing Performance Council. The TPC-W home page. http://www.tpc.org/tpcw.

[71] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, 2007.

[72] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *ASPLOS*, 2011.

[73] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus scalable block store. In *NSDI*, pages 357–370, Lombard, IL, 2013.

[74] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT. In *Eurosys*, 2011.

[75] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[76] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MOBS*, 2007.

[77] Kiam S. Yap, Pankaj Jalote, and Satish K. Tripathi. Fault tolerant remote procedure call. In *ICDCS*, pages 48–54, 1988.

[78] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, 2003.

[79] Wei Zhang, Chong Sun, and Shan Lu. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, New York, NY, USA, 2010. ACM.