Techniques for Integrating Erasure Codes and Model Checkers with Distributed Systems

Ellis Michael

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2023

Reading Committee:

Dan R. K. Ports, Chair

Thomas Anderson

Arvind Krishnamurthy

Program Authorized to Offer Degree:

Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Techniques for Integrating Erasure Codes and Model Checkers with Distributed Systems

Ellis Michael

Chair of the Supervisory Committee:
Dan R. K. Ports

Paul G. Allen School of Computer Science & Engineering

Distributed systems are vital components in our modern computing infrastructure. They must be correct, guaranteeing that important safety properties hold not just in the common case but in all possible cases. They must also be performant and efficient, providing low latency and high throughput to their users while consuming as few resources as possible. This thesis addresses these broad goals of distributed systems in two ways.

First, this thesis introduces the Amalgam protocol which utilizes erasure codes to reduce the storage impact of fault-tolerant distributed systems. Traditionally, state machine replication protocols have provided strong consistency guarantees along with the ability to execute arbitrary operations on shared state in a single, atomic transaction. However, replication comes at the cost of storage overhead. The Amalgam protocol obviates this trade-off by handling transactions on erasure-coded data with similar performance characteristics to a replicated baseline. Amalgam guarantees linearizability while tolerating the complete failure of a configurable number of machines. The protocol also supports the replacement of failed machines and utilizes a snapshot protocol to ensure that replacement servers correctly rebuild their state and restore the overall health of the system.

This thesis also presents the DSLabs framework. Distributed systems are notoriously difficult to implement, and because they are inherently non-deterministic, distributed systems are also notoriously difficult to test. Adverse network conditions can manifest bugs which might otherwise lie dormant. Explicit-state model checking allows for the

systematic exploration of all possible executions of a distributed system. However, many distributed systems have infinitely large state spaces which grow in size very quickly as the execution depth increases. DSLabs is a framework for designing and model checking distributed systems. DSLabs enables the creation of guided searches of the state space of a distributed system, allowing a system designer to circumvent the state explosion problem by focusing on areas of the state space which are likely to be problematic. The DSLabs programming framework is designed to be used by novice programmers — in particular, university students — and provides powerful tools for model checking and visualizing distributed systems, letting students focus on the already difficult task of implementation.

ACKNOWLEDGEMENTS

First and foremost, I would like to acknowledge and thank my advisors Dan and Tom. I've learned so much over the years from Dan and would not have become the researcher I am without his mentorship. Tom was an early champion of the DSLabs work, and his dedication to undergraduate education is admirable.

Next, I would like to thank all of my collaborators and coauthors: Jialin Li, Naveen Kr. Sharma, Adriana Szekeres, Doug Woos, Michael Earnst, Zachary Tatlock, Hang Zhu, Zhihao Bai, Ion Stoica, Xin Jin, Jacob Nelson, Inho Choi, and Yunfan Li. I would also like to thank and recognize all of the members of the UW CSE Systems Lab during my tenure: Lequn Chen, Raymond Cheng, Tapan Chugh, Tianyi Cui, Pedro Fonseca, Helga Gudmundsdottir, Seungyeop Han, Yuchen Jin, Antoine Kaufmann, Arvind Krishnamurthy, Niel Lebeck, Hank Levy, Jialin Li, Katie Lim, Vincent Liu, Ratul Mahajan, Ashlie Martinez, Samantha Miller, Luke Nelson, Simon Peter, Henry Schuh, Will Scott, Naveen Kr. Sharma, Haichen Shen, Helgi Sigurbjarnarson, Anna Kornfeld Simpson, Adriana Szekeres, Xi Wang, Irene Zhang, Kaiyuan Zhang, Qiao Zhang, Chenxingyu Zhao, Kevin Zhao, and Danyang Zhuo. One of the greatest joys and privileges of graduate school was spending my time surrounded by intelligent and driven people. I learned so much from each of you.

Lastly, I would like to thank all of my climbing, skiing, hiking, and running partners, especially the members of the Race Condition Running group: Chandrakana Nandi, Zachary

Tatlock, Max Willsey, William Agnew, Lauren Bricker, Maureen Daum, Sami Davies, Daniel Epstein, Martin Kellogg, Aishwarya Mandyam, Yuxuan Mei, Edward Misback, Amanda Robles, Max Ruttenberg, Brett Saiki, Gus Smith, Stephen Spencer, Ewin Tang, Trang Tran, Nick Walker, Remy Wang, Ethan Weinberger, James Wilcox, Eric Zeng, and many others. Compared to the marathon of graduate school, training for and running actual marathons with all of you was still hard. But at least we had perspective.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed systems are ubiquitous and vitally important. They undergird all modern cloud computing infrastructure and are the unseen backdrop of modern life. On the scale of data centers, hardware failures are commonplace, but distributed protocols can mask these failures. Without the strong consistency guarantees that protocols such as Paxos provide, users and application programmers could be exposed to anomalies such as data loss, operation reordering, and interleaved transactions. However, strongly consistent coordination protocols ensure that the systems behave as individual, correct processes. These protocols, however, come with various costs and are notoriously difficult to implement correctly.

One cost that is seemingly inherent to the idea of fault tolerance is storage overhead. In order to tolerate the complete failure of a machine or storage device, data is often *replicated*, and that replication is managed by state machine replications protocols such as Paxos or Viewstamped Replication. While this does indeed allow a system to continue on after the failure of one or more replicas, it means that the overall storage footprint of such a system is multiple times higher than that of an unreplicated system. The traditional way to reduce storage overhead while maintaining fault tolerance is *erasure coding*. However, existing distributed systems utilizing erasure codes only support limited read/write

interfaces and are not compatible with a rich state machine replication interface. This thesis introduces Amalgam, a sharded storage system which utilizes linear erasure codes to provide low-latency, read-modify-write, erasure-coded storage.

Distributed protocols are also difficult to devise and correctly implement. To that end, this thesis also presents the DSLabs model checker. Model checking is the systematic exploration of all possible executions of a distributed system and differs from traditional testing methodologies in that it can reliably find implementation bugs which only occur under rare or very specific conditions. Originally designed for teaching undergraduates, the DSLabs framework is designed for building runnable and checkable distributed systems. It is optimized for ease-of-use and contains a number of features for checking and debugging distributed systems, including a visual debugger for exploring problematic traces.

Previously Published Work

The DSLabs work was previously published in the proceedings of the 14th European Conference on Computer Systems [52]. Since publication, Oddity [71], the visual debugger which was included in DSLabs, has been replaced with a new debugger. Section 4.5 has been updated accordingly.

CHAPTER 2

BACKGROUND

This chapter introduces important background information on distributed systems, consensus, state machine replication, erasure coding, and explicit-state model checking. Section 2.1 discusses consensus and state machine replication, as well as some of the problems with previous approaches to state machine replication which Amalgam will address. Section 2.2 details one aspect of certain state machine replication protocols which reduces tail-latency and is incoporated in the Amalgam protocol. Section 2.3 describes linear block codes, which are used in Amalgam, and describes how linearity is used to efficiently update parity blocks. Section 2.4 lists some well-known problems with erasure codes which make them challenging to use in distributed systems. Finally, Section 2.5 gives background information on model checking which will be useful in understanding DSLabs's particular use of model checking.

2.1 Consensus and State Machine Replication

Consensus protocols such as Paxos [35, 36] have long been used to build distributed systems which tolerate server failures by replicating data across several replica machines. These protocols allow for the crash-failure of less than half of their servers, while provid-

ing strong consistency. In Paxos, proposers propose values to acceptors using proposal numbers. However, before a proposer is allowed to use a proposal number, it must acquire a promise from the acceptors that they will no longer accept smaller proposal numbers. In this first phase, proposers also learn of any proposals previously accepted by the acceptors; proposers adopt the value from *highest-numbered* proposal as their own value to propose, if one exists. In both phases, proposers require responses from a *quorum* of the acceptors (usually a simple majority). Any two quorums intersect, by definition; thus, if a value is accepted by a majority in the second phase of the protocol, any proposer attempting to use a higher-numbered proposal number will necessarily learn the chosen value during its own first phase. At most one value will be chosen, and all nodes in the system that learn of a chosen value will agree on the value chosen. Furthermore, that chosen value must have been proposed by some proposer.

Consensus protocols, which allow nodes in a distributed system to agree on a single chosen value, can be extended in a natural way to state machine replication protocols. In state machine replication, clients send *commands*, these commands are processed according to the rules of a pre-determined state machine, and the clients receive corresponding *results*. One way build a replicated state machine which provides strong consistency guarantees is through a shared log, where, for each entry in the log, server nodes running the replicated state machine use a consensus protocol to agree on a command. Because the state machine is deterministic, each server can independently execute these commands in order. From the clients' perspective, the results of this execution appear as if they came from a single, correct server node, even though some of the servers can fail.

Numerous variations on the Paxos protocol have been discovered over the years, including protocols which reduce the number of message delays required to commit operations [37, 41] and protocols which reduce contention between multiple proposers [48, 53]. However, most of these approaches share two common flaws: (1) their maximum throughput is limited to that of a single machine, and (2) replicating state machine data results in storage overhead.

Addressing the first issue, while scalability of fault-tolerant distributed systems is not a solved problem — indeed, many papers have been and will continue to be published on this topic — there is a canonical approach to achieving scalability — namely, *sharding*. The state is divided into multiple shards, each held independently by a subsystem. Operations on separate shards can be processed in parallel, allowing for horizontal scalability as more of these subsystems are added. In order to handle operations which read and write data on multiple shards, an atomic commitment protocol such as two-phase commit [5] is used by the subsystems to coordinate between themselves. A common approach is to use multiple, independent replica groups running Paxos to handle individual shards. Such a system will process transactions on data held at a single Paxos group with the same latency and throughput as a standard Paxos cluster, while still allowing atomic transactions across shards. We will refer to this system as ShardedPaxos.

The second issue, however, is fundamental to the idea of replication. In order to tolerate the failure of f machines, a replicated system must store at least $f+1$ copies of the data. The *storage overhead* of replication — the number of additional copies of the data that are stored — is f.

2.2 Replication Witnesses

Paxos state machine replication is typically deployed on groups of $2f+1$ servers, for some value of f, and when space is not a concern, state is replicated uniformly across all servers. Upon receiving some operation to commit to the Paxos log, the Paxos leader sends it to the other replicas and waits for at least f other replicas (i.e., a simple majority, including the leader) to accept the operation. Once the operation has been committed to the log, as long as all previous operations have been committed, the operation can be executed.

However, the initial explanations of the Paxos protocol [35, 36] and a subsequent examination focused on understandability [67] describe Paxos as a *heterogeneous* system

composed of actors with different roles. In particular, acceptors are the processes in the system responsible for accepting values and helping elect leaders. As explained in Paxos Made Moderately Complex (PMMC), once the replicas — which hold the permanent application state — have learned that state machine operations have been committed, and once they have applied them to their copies of the state machine, these operations can be garbage collected at the acceptors. At least $2f + 1$ acceptors are necessary for a deployment of PMMC to continue to function when f acceptors can fail, but only $f + 1$ replicas (with full copies of the state) are needed. Thus, when Paxos is deployed heterogeneously, its storage overhead is f – the theoretical minimum for a replicated system.

In Amalgam, we will refer to servers playing the same role as PMMC acceptors as *replication witnesses*. These processes help drive the system towards consensus but do not store permanent data. Other servers will play the roles of acceptors as well as learners (though, on erasure-coded data). A leader must ensure that operations are durable at a quorum of non-witness replicas before garbage collecting log entries. Once that happens, though, replication witnesses can discard the log entries entirely (rather than execute them and apply them to the current state).

Replication witnesses are distinct from the witnesses used in Harp [43] and Paxos Made Practical [49]. Those were election witnesses whose sole purpose was to elect members of a designated replication quorum in the case of a failure. Replication witnesses have the advantage of allowing the system to handle the temporary unavailability of servers without appreciably increasing tail latency.

Furthermore, while at least $2f + 1$ servers total are necessary to implement state machine replication, and at least $f + 1$ copies of the data must be kept for fault tolerance, the leader replica need not initially send operations to all replication witnesses. If desired, it can save network bandwidth by choosing to send operations to a smaller number of witnesses initially and only using the remaining witnesses as alternatives. Sending operations to just one or two replication witnesses (as well as the non-witness servers) can still dramatically reduce tail latency while also reducing network bandwidth.

2.3 Erasure Coding

Before any further discussion of distributed systems, we must establish some common knowledge of and terminology for erasure codes. Readers familiar with systematic, linear block codes (e.g., Reed–Solomon codes) can skip this section.

Reducing the storage overhead of fault-tolerant distributed systems requires the use of coding, rather than simple data replication. *Maximum-distance separable* (*MDS*) *linear block codes*, such as Reed–Solomon codes, are ideal for this application. The interface they provide takes k blocks of data as input and produces n blocks of output data such that any k of these n blocks can be used to reconstruct the original input data. Furthermore, any linear block code can be transformed into a *systematic* code — one in which the first k blocks of the n output blocks are identical the input blocks. We will refer to a systematic MDS linear block code with parameters n and k as a (n,k) code. The $n-k$ output blocks which do not correspond to input blocks are *parity blocks*.[1] The *storage overhead* of storing all output blocks exactly once is $(n-k)/k$.

These systematic, linear block codes have some nice properties. First, the encoding procedure of an (n,k) code can be executed as a matrix multiplication using the code's *generator matrix*, $G \in \mathbb{F}^{k\times n}$. Here, $\mathbb{F}$ is the finite field the blocks of data come from, typically consisting of 2^m elements. Because the code is systematic, we can write G as $G = [I_k \mid P]$, where I_k is the identity matrix. So, to encode a vector of input blocks $\boldsymbol{d} \in \mathbb{F}^k$, we simply multiply

$$\boldsymbol{d} \cdot P = \boldsymbol{p}$$

to get $\boldsymbol{p} \in \mathbb{F}^{n-k}$ the corresponding parity blocks.

Linear codes also generate *updates* to parity blocks efficiently. Suppose we have some $\boldsymbol{d}, \boldsymbol{p}$ pair generated as above, and we update the ith element of $\boldsymbol{d}$ to get $\boldsymbol{d}'$. We wish to

[1] The parity blocks are not actually bitwise parity for most erasure codes. However, this terminology is common in the literature. Also, the use of "block" to refer to a single element of the coding alphabet, is more common in the storage literature, while the "block" in "block code" typically refers to the entire input string in the coding literature.

generate $\boldsymbol{p}' = \boldsymbol{d}' \cdot P$.

$$\begin{aligned}\boldsymbol{p}' &= \boldsymbol{d}' \cdot P \\ &= (\boldsymbol{d}' - \boldsymbol{d}) \cdot P + \boldsymbol{d} \cdot P \\ &= (\boldsymbol{d}' - \boldsymbol{d}) \cdot P + \boldsymbol{p} \\ &= (0, \ldots, 0, d_i' - d_i, 0, \ldots, 0) \cdot P + \boldsymbol{p}\end{aligned}$$

When modifying a single data block, we can generate the necessary updates to the parity blocks without knowing the values of the other data blocks. We can also update multiple data blocks independently and apply the updates to each of the elements in our parity block vector in any order because addition is commutative. Moreover, because the vector $(0, \ldots, 0, d_i' - d_i, 0, \ldots, 0)$ consists of just a single non-zero element, we can compute $(0, \ldots, 0, d_i' - d_i, 0, \ldots, 0) \cdot P$ efficiently.

2.4 Erasure Coding and Distributed Storage

While adapting existing distributed storage protocols to use erasure codes may seem straightforward, the use of erasure codes introduces significant challenges. Taking Paxos as a running example, one might think a Paxos cluster could use a $(2f+1, f+1)$ code — a leader could encode data and send each node a single block, waiting for a majority to acknowledge the write. However, if the leader were to fail and another leader were to execute the first phase of Paxos, there would not be enough information in *every quorum* to successfully reconstruct the written data [55]. If a commit succeeded, the new leader would be guaranteed to see *some block*, but with a $(2f+1, f+1)$ code, $f+1$ blocks are necessary to reconstruct the data. In mitigating this, we have a few options. The first is to increase the cluster size and quorum size to $3f+1$ and $2f+1$, respectively — and to use a code with parameters $(3f+1, f+1)$. Expanding the number of nodes in this

way preserves the property that the intersection of any two quorums contains a *full copy of the data*, which is the property that Paxos requires.

Another possible strategy to integrate erasure codes with Paxos is adding another round of communication. The leader would propagate the (non-coded) data normally. Once the operation is acknowledged by a majority, it would be committed, and the leader could reply to the client. Once *all servers* acknowledge the operation, the leader would instruct each of them to encode the data, save their respective block, and delete the original data. Stated another way, commands to the state machine could be replicated to a shared log in the normal fashion, and only during the execution phase would each of the replicas actually encode data. A variation on this approach is that the leader could send the coded pieces to all nodes *first* and wait for *all nodes* to acknowledge, only using normal, non-coded Paxos as a backup.

The second problem that erasure coding presents in the context distributed storage protocols is that executing read-modify-write (RMW) operations can require first gathering and decoding data before executing the operation and then re-encoding it. Indeed, some existing protocols which integrate erasure codes with Paxos, such as RSPaxos, suffer from this problem [55]. If a single data item is split into separate blocks, parity blocks are generated, and the blocks are distributed across a Paxos cluster, then an RMW operation would entail at least two extra message delays and additional network bandwidth in order to first gather and reconstruct the data item before executing and committing the operation.

It is for this reason that existing distributed systems for storing erasure-coded data do not support arbitrary read-modify-write operations [12, 13, 55]. They either only allow key-value operations such as `PUT` and `GET`, or they are designed for long-term storage and only allow the addition and (eventual) deletion of data.

2.5 Exploratory Model Checking

In DSLabs, we check implementations of distributed systems by exploring their *state graphs*. These state graphs are a distributed version of Kripke structures [30], where state labels are given by state predicates that are evaluated on each discovered state. We will discuss the details of the DSLabs programming model and API more thoroughly in Section 4.1, but we describe the basics here briefly.

A distributed system in DSLabs consists of a set of *nodes*. Each node defines message and timer handlers. In its message and timer handlers, a node can update its internal state, send messages, and set timers. Each of these handler functions is a single-threaded, atomic operation that should run to completion without blocking or waiting for I/O. By allowing the test harness to interpose on all concurrency in the system and make scheduling decisions — ordering the events delivered at each node — we give it the flexibility to explore many possible schedules, even ones unlikely to occur under normal conditions.

Now, we can define the state graph of a distributed system. Each vertex in this graph is a particular state of the entire system — consisting of the internal states of all nodes, the state of the network (which messages can be delivered), and the state of the nodes' timer queues (which timers are waiting to be delivered). There is a directed edge from s to s' if s' is the resulting state after delivering a single event (a message or timer) to a single node in state s. The initial state of the system is the state of all nodes after the initialization event (but before they have handled any other events). The state graph of the system, then, is the graph of all states reachable from this initial state. Figure 2.1 shows a portion of an example state graph.

Note that this graph is merely conceptual; it is not explicitly constructed during the execution of a distributed system. Also note that vertices in the state graph are defined by the state of the system, not by the order of events which produced them. Some events, in fact, necessarily commute because they are *concurrent* — i.e., they are not ordered by the happens-before relation [34]. Finally, note that this graph is distinct from the execution lattice associated with a particular distributed execution [3]. The state graph captures *all*

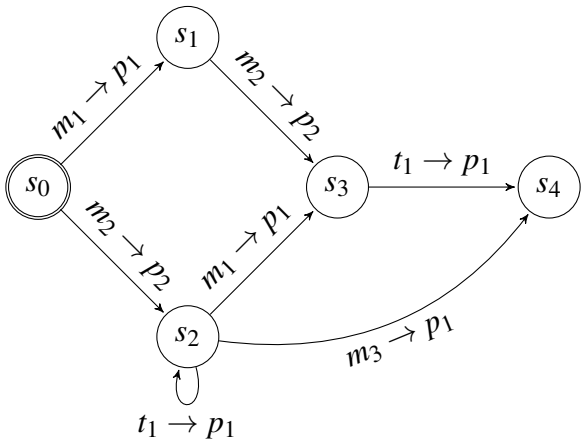

Figure 2.1: *A portion of a state graph. The initial state is s_0. In this example, edges are labeled by their corresponding events (i.e., $e \to p$ denotes event e being delivered to p). Some executions necessarily lead to the same state based on the commutativity inherent to the distributed programming model (e.g., delivering messages m_1, m_2 in either order starting from s_0). Others lead to the same state by accident of the specific implementation (e.g., delivering either m_3 or m_1 followed by timer t_1 starting from s_2).*

possible executions of a distributed system; an execution corresponds to a particular path in this graph, starting from the initial state.

Now, we can define *explicit-state model checking*, which is the systematic exploration of the state graph, checking that each state satisfies certain properties set forth by the specification.

Generally speaking, an implementation of a distributed system is correct if it meets the *safety* and *liveness* criteria of the specification [2]. Safety properties describe the "bad things" that must not happen during any execution. Liveness properties, on the other hand, describe the "good things" that must eventually happen in all executions. Whereas running a distributed system is equivalent to taking a single path through the state graph, model checking is the systematic exploration of this graph, checking that it meets the given criteria. In DSLabs, we restrict our model checking efforts to checking safety properties expressed through state invariants — predicates which must be true of all reachable states.

The simplest form of model checking is breadth-first search of the state graph. Using breadth-first search guarantees that when the model checker finds an invariant-violating state, it can produce a trace (a path through the graph) which demonstrates the error, and that trace will be of *minimal length*. For example, in Figure 2.1 if s_4 violated an invariant, the model checker would return the trace $[m_2; m_3]$ rather than $[m_2; m_1; t_1]$. Moreover, unlike tests which rely on running the system to find invariant violations, systematic exploration of the state graph will reliably find bugs which rely on precise (and unlikely) orderings of events.

The DSLabs model checker is stateful — it maintains the set of discovered states and a queue of states to explore. It explores the successors of a state by taking each pending event, cloning the state, and then delivering that event to the appropriate node in the clone. Importantly, the model checker tests whether the new state is equivalent to any previously-discovered state (as discussed in Section 4.3.1.1), avoiding wasting work exploring duplicate states.

CHAPTER 3

AMALGAM: LOW-LATENCY, TRANSACTIONAL, ERASURE-CODED BLOCK STORAGE

Replication is a basic building block of fault-tolerant distributed systems. State machine replication protocols such as Paxos allow a system to mask server failures while maintaining strong consistency properties. Sharding of data allows for the deployment of multiple such subsystems, each handling a portion of the total state, providing scalability. When layered with an atomic commitment protocol such as two-phase commit, these subsystems can coordinate to handle cross-shard transactions. Traditionally, however, these protocols come with performance and resource costs.

Many distributed systems have used erasure codes to help ameliorate the storage overhead of replication. Ordinarily, tolerating f server failures requires keeping at least $f+1$ full copies of the system's state (with many systems using $2f+1$ copies). Erasure codes such as Reed–Solomon codes and Reed–Muller codes allow distributed systems to use far less storage while still providing the same level of fault tolerance.

However, storage systems utilizing erasure codes often sacrifice performance, functionality, or both. State machine replication protocols such as Paxos can handle arbitrary

read-modify-write operations with low latency.[1] That is, in four one-way message delays, a client can apply an arbitrary command to the replicated state machine and receive the results of the computation. In contrast, existing distributed systems utilizing erasure codes only support a limited key/value interface (e.g., only allowing `PUT` and `GET`) or even disallow the overwriting of data entirely, supporting only write-once/delete-once storage. These limitations are acceptable in many contexts such as cold storage, where data is written and read infrequently. However, these existing systems do not provide the same *transactional* interface that replications protocols such as Paxos support.

The goal of this chapter is to devise a distributed coordination protocol which:

- is **strongly consistent**, guaranteeing linearizability of client operations;
- is **transactional**, handling arbitrary **read-modify-write** operations;
- is **scalable** and **efficient**, making good use of the resources available and growing in serving capacity as resources are added;
- provides **low-latency** access for clients, committing operations in four message delays in the normal case;
- does not use excessive **network bandwidth**;
- is **fault-tolerant**, allowing for the failure and recovery of multiple constituent storage nodes without data loss;
- and finally, has a low and configurable **storage overhead**, utilizing erasure codes to reduce storage usage lower than can be achieved via replication.

These goals impose certain constraints on our design. The goal of handling read-modify-write operations with low latency and without excess network traffic requires

[1] In this thesis, we use read-modify-write (RMW) to refer to the entire class of functions which can be executed against the shared state. "Read-modify-write" does not imply that there is only a single read phase nor does it imply that the *read set* of operations is known *a priori*.

co-locating system data with the server processes responsible for driving the operation commit procedure to completion. As we will see in Section 3.2, the goal of transactionality also imposes constraints on the way in which our system can utilize erasure codes. Finally, the goal of scalability precludes solutions in which all operations are serialized to a single shared log, for example.

In this thesis, we present Amalgam, a new approach to using erasure codes in the context of sharded distributed storage. Amalgam provides the abstraction of sharded, scalable, fault-tolerant, strongly consistent, transactional block storage. Amalgam matches the performance of a similarly scalable system of multiple, independent Paxos clusters, while maintaining a storage overhead close to zero. Importantly, Amalgam retains the ability to handle arbitrary read-modify-write operations with the same performance characteristics as Paxos. And like our reference ShardedPaxos system of multiple, independent Paxos clusters, Amalgam can handle cross-shard transactions when an atomic commitment protocol such as two-phase commit is layered on top of it.

Amalgam is able to achieve this using linear erasure codes to update parity blocks. Rather than splitting individual data items into pieces to be coded and distributed to storage servers, *separate data items* in Amalgam are organized into stripes. In a stripe of data, each data block is maintained by a separate data server, while each parity block is held at a separate parity server. Data servers accept operations from clients, execute transactions across their entire shard of data blocks, and send state updates to parity servers to be committed and eventually applied to the parity blocks.

Amalgam is a coordination protocol, akin to Paxos or Viewstamped Replication, although with two key differences. First, although Amalgam can process read-modify-write operations, it operates on fixed-size values. Second, whereas state machine replication protocols operate on a single shard of data and scale horizontally, Amalgam is inherently sharded and must be configured to operate on multiple, independent shards of data in order to achieve storage savings compared to replicated protocols. Amalgam is not a

database nor an object storage system. However, it can form the basis for such systems provided that they can operate on top of Amalgam's data model.

Traditionally, erasure codes have been used in the context of disk-based storage. However, Amalgam can store its data on disk or in memory, and it is for in-memory systems that Amalgam provides the best value proposition. While erasure coding for long-term on-disk storage is important, the cost of DRAM is orders of magnitude higher than spinning disks and much higher than SSDs. Moreover, even enterprise hardware is limited in the amount of RAM it can support on a single machine. Furthermore, in-memory systems are the ones most likely to require low-latency transactions.

It is important to note, however, that the choice of storage medium is entirely orthogonal to the design of the Amalgam protocol. If it is desirable to tolerate the temporary failure of *all* machines in a deployment of Amalgam (e.g., due to a power outage), then Amalgam nodes can persist their data to non-volatile storage (although we do not discuss in this thesis which pieces of state can remain in volatile storage).

This chapter is organized as follows: Section 3.1 lays out our system model — our assumptions about the environment and the guarantees Amalgam provides. Section 3.3 describes the Amalgam protocol, including the reconfiguration and data recovery protocols. Section 3.4 gives an argument for the safety and liveness of the Amalgam protocol. Section 3.5 analyzes the fault tolerance of Amalgam compared to a replicated system. And finally, Section 3.7 summarizes the work most closely related to Amalgam.

3.1 System Model

We assume a completely asynchronous system. The network can arbitrarily drop, reorder, and duplicate messages. When we write that servers communicate over FIFO channels, these channels are built on top of an underlying asynchronous network. We do not assume any bound on clock skew.

Amalgam is designed to tolerate crash (i.e., non-Byzantine) failures. When servers fail (or are suspected of having failed), they are replaced. Server replacement and reconfiguration is handled by a fault-tolerant reconfiguration service; this service is itself implemented by a state machine replication protocol. System availability is dependent on sufficient network synchrony [14] and the absence of more than f simultaneous failures. Specifically, no more than f physical machines can simultaneously crash or have their data in a *degraded* state (Section 3.3.2).

Amalgam provides *linearizability* of client operations. From the clients' perspective, the entire system appears as a single, correct process executing operations serially.

3.2 Amalgam Overview

As previously mentioned, a system consisting of multiple, independent Paxos groups (ShardedPaxos) provides scalability, fault tolerance, and strong consistency — but comes at the cost of a storage overhead of f to tolerate f machine failures. In order to further reduce storage overhead and still provide fault tolerance, erasure codes must be used. However, systems utilizing erasure codes come with a variety of drawbacks compared to ShardedPaxos. Storage systems such as RAID [62] use multiple disks and erasure coding to reduce storage overhead; however, the storage controller itself is a single point of failure and a throughput bottleneck. Most distributed systems utilizing erasure codes are designed for *cold storage*. These systems can recover from faults and scale to meet high demand, but modifications to existing data are either not allowed or are very expensive.

There have been attempts to integrate the Paxos replication protocol with erasure codes [12, 55]. However, these protocols do not allow RMW operations like ShardedPaxos. One issue many previous erasure-coded distributed systems have is their *data model*. Systems such as RSPaxos [55] take individual individual values from key-value pairs, split them into blocks, use erasure codes to generate parity blocks, and distribute the blocks among the servers. Because the value has been split into pieces, in order to process a mod-

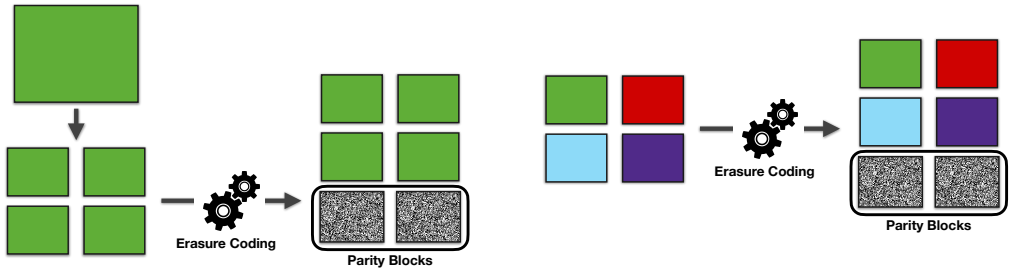

Figure 3.1: *The two basic ways to use linear erasure block codes. On the left, a single data item is divided into pieces, and parity blocks are generated with respect to those pieces. On the right, separate data items are grouped together, and parity blocks are generated with respect to the group. Both schemes allow for the erasure of any two blocks. However, in order to read and modify the green image in the left example, the pieces must be gathered together from wherever they are stored.*

ification to the value, those pieces must first be gathered and reassembled — resulting in additional latency and network utilization.

Our system, Amalgam, takes a different approach. Instead of taking objects and splitting them apart, in Amalgam, completely different objects are organized into logical groups, and parity blocks are generated with respect to those groups. Each object is itself an input block to the encoding function of our erasure code. Figure 3.1 illustrates the difference between the two coding models.

Amalgam provides the same level of scalability and functionality as ShardedPaxos, while having storage overhead close to zero. Amalgam achieves this by using erasure codes to generate parity blocks for data held at separate servers and by using the linearity of erasure codes to efficiently update those parity blocks when data is modified. Instead of taking values and splitting them up into blocks, Amalgam takes matching size values, one from each shard, and organizes them into *stripes*.

Amalgam comprises three kinds of processes: data servers, parity servers, and witnesses. Each data server stores a single shard of data blocks, while parity servers store parity blocks. Figure 3.2 depicts the data layout for a simplified version of Amalgam, which nevertheless demonstrates the usefulness of linear erasure codes. In this example,

Stripes

$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$p_{1,1}$	$p_{1,2}$
$d_{2,1}$	$d_{2,2}$	$d_{2,3}$	$p_{2,1}$	$p_{2,2}$
$d_{3,1}$	$d_{3,2}$	$d_{3,3}$	$p_{3,1}$	$p_{3,2}$
$d_{4,1}$	$d_{4,2}$	$d_{4,3}$	$p_{4,1}$	$p_{4,2}$

Figure 3.2: *Data layout of a system with a single group of parity servers for every stripe. Each column represents the data stored by a single server. Data blocks are shown in gray; parity blocks are shown in red. Blocks in the same row belong to the same stripe.*

each physical machine is either a data or parity server. Data servers commit operations by first executing them locally and calculating the *state update* — i.e., the `XOR` of the old data and new data. They then send these updates to the parity servers, and once the updates are committed, the parity servers can use these state updates to update their parity blocks. The storage overhead of this system is only dependent on the parameters of the erasure code used. At least f parity servers are needed to tolerate f server failures, but these f parity servers can support as many data servers as desired.

Stripes

$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$p_{1,1}$	$p_{1,2}$
$d_{2,2}$	$d_{2,3}$	$p_{2,1}$	$p_{2,2}$	$d_{2,1}$
$d_{3,3}$	$p_{3,1}$	$p_{3,2}$	$d_{3,1}$	$d_{3,2}$
$p_{4,1}$	$p_{4,2}$	$d_{4,1}$	$d_{4,2}$	$d_{4,3}$

Figure 3.3: *Physical data layout of a distributed parity system. Columns represent the data held by a physical machine; however, in Amalgam, data blocks and parity blocks are handled by separate processes co-located on physical machines.*

The simplified system described above has an obvious flaw, however. Whereas the maximum throughput of ShardedPaxos increases with the number of Paxos groups added, the throughput of our simplified version of Amalgam is limited to that of a single server. As long as the load is distributed evenly, data servers may receive a small fraction of the incoming operations, but each of our parity servers must process parity updates for all

of the system's operations. Therefore, in Amalgam we provision a single parity server per machine, but each parity server only maintains parity blocks for a subset of the stripes. Importantly, a single physical machine never plays more than one role for a single stripe. Figure 3.3 depicts this type of data layout. Data servers in Amalgam still commit operations to parity servers using state updates; however, which parity servers they use depends on which stripe the operation modifies.

Stripes

$d_{1,1}$	$d_{1,2}$	$d_{1,3}$	$p_{1,1}$	$p_{1,2}$	w	w
$d_{2,2}$	$d_{2,3}$	$p_{2,1}$	$p_{2,2}$	w	w	$d_{2,1}$
$d_{3,3}$	$p_{3,1}$	$p_{3,2}$	w	w	$d_{3,1}$	$d_{3,2}$
$p_{4,1}$	$p_{4,2}$	w	w	$d_{4,1}$	$d_{4,2}$	$d_{4,3}$
$p_{5,2}$	w	w	$d_{5,1}$	$d_{5,2}$	$d_{5,3}$	$p_{5,1}$
w	w	$d_{6,1}$	$d_{6,2}$	$d_{6,3}$	$p_{6,1}$	$p_{6,2}$

Figure 3.4: *Physical data layout of a distributed parity system with witnesses. Blocks shown in blue indicate that there is no data nor parity block for that stripe on the machine; instead, the server functions as a witness to replication for that stripe. Replication witnessing is handled by a separate process.*

The last type of process run on Amalgam machines is the replication witness which, as described in Section 2.2, allows Amalgam to reduce tail latency. Replication witnesses help commit operations just like parity servers; however, they store no permanent state aside from a small amount of protocol data. As soon as an operation is safely committed at all parity servers, witnesses can discard it. Figure 3.4 depicts an Amalgam deployment with replication witnesses. Again, data, parity, and witness servers are co-located on physical machines for load balancing, but they are entirely separate processes, and no physical machine can play more than one role in a given stripe.

Figure 3.5 depicts a logical view of Amalgam's architecture and process by which a single operation is committed. A client wishing to modify the data on a single data server sends the operation to the data server. The data server executes the operation on its local state and computes the state update caused by the operation. The data server sends the

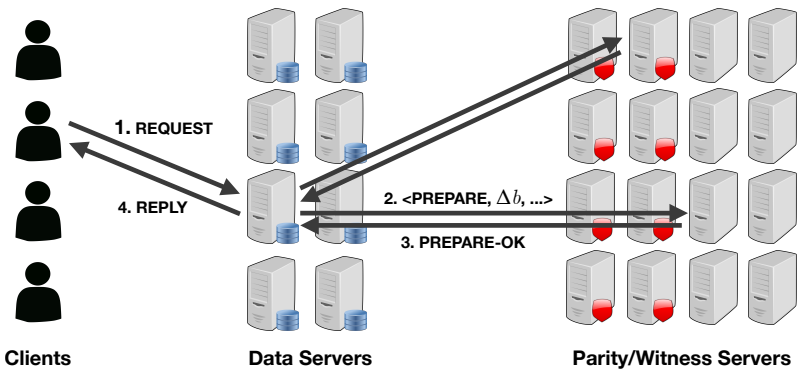

Figure 3.5: *Architecture overview of Amalgam and the flow of messages to commit a single request which updates block b.*

state update to the designated parity servers and witnesses for the block's (or blocks') stripe, and waits for a reply from a quorum. Once the data server has received acknowledgements from a quorum, it can reply to the client with the results of the operation.

In order to handle server failures, Amalgam makes use of a separate, fault-tolerant *reconfiguration service*. This service is responsible for coordinating the replacement of data servers, parity servers, and witnesses (Section 3.3.2). The reconfiguration service itself can be replicated using standard means (e.g. Paxos). When servers fail and are replaced in Amalgam, they are initialized in a *degraded* state; they contain no data. Data servers and parity servers must use Amalgam's data recovery protocol (Section 3.3.2) to rebuild their state.

Because Amalgam provides the abstraction of multiple, independent, fault-tolerant storage nodes, atomic commitment protocols such as two-phase commit can be used to coordinate atomic transactions across data servers. Implementing and evaluating these protocols is beyond the scope of this thesis, but in principle their performance with Amalgam should be similar to ShardedPaxos and similar designs. We briefly discuss integration of atomic commitment protocols with Amalgam in Section 3.3.5.

The one limitation of Amalgam, as compared to ShardedPaxos, is that it provides a block storage abstraction, or rather, a fixed-size storage abstraction. While data in Paxos and ShardedPaxos can have arbitrary size and shape, data items in Amalgam must be

organized into stripes across data servers. Different stripes can have different sizes, but in Amalgam, all data items in the same stripe must fit into a single block size. Amalgam is indifferent to the strategy used to allocate stripes. This could be done statically — e.g. by having a uniform block size and letting each data server allocate blocks as needed from the beginning — or dynamically by a separate allocation service. For this thesis, we focus on a deployment of Amalgam with a single uniform block size.

3.3 The Amalgam Protocol for Erasure Coded Storage

In order to mitigate the high storage overhead of sharded storage while still providing fault-tolerant, strongly consistent storage, we introduce Amalgam. Amalgam provides the abstraction of sharded block storage. Each shard consists of a number of data blocks. The Amalgam protocol consists of three sub-protocols: the update protocol used to commit arbitrary RMW operations to a shard's blocks (Section 3.3.1), the server replacement protocol used to replace failed servers (Section 3.3.2), and the block recovery protocol used to recover individual data/parity blocks after server replacement (Section 3.3.2.3). We present the protocols for the data servers and parity servers first and explain how to integrate replication witnesses in Section 3.3.3.

The g data servers each maintain a single shard of data blocks, while the $p \geq f$ parity servers maintain the common set of parity blocks. The blocks within a shard need not be the same size. However, the data blocks and parity blocks are organized into *stripes* of size $g+f$, each consisting of identically sized data blocks and parity blocks from different servers. Each stripe contains exactly one data block from each data server and exactly one parity block from f of the p parity servers. The parity blocks are generated using a $(g+f, g)$ systematic linear block code — using the Amalgam block recovery protocol, any g of the $g+f$ blocks in a stripe is sufficient to reconstruct the stripe's g data blocks. Therefore, the long-term storage overhead of Amalgam is $\frac{f}{g}$.

Data server:

- `primary_num` (int) — the data server's ID, set by the configuration service
- `op_num` (`int`) — number for next operation
- `pending_ops` (`set[operation]`) — operations which have not been acknowledged by all parity servers
- `blocks` — local data, map from block IDs to data blocks
- `config` — set of parity and witness servers for the current configuration

Parity server:

- `prepares` (`set[message]`) — set of accepted PREPARE messages
- `op_nums` (`array[int]`) — most recently executed `op_nums` from each shard
- `blocks` — local data, map from block IDs to parity blocks
- `primary_nums` (`array[int]`) — vector of `primary_nums`, the data server IDs for each shard

Witness server:

- `prepares` (`set[message]`) — set of accepted PREPARES
- `primary_nums` (`array[int]`) — vector of `primary_nums`, the data server IDs for each shard

Figure 3.6: *Local state of Amalgam servers. Data servers and parity servers hold data/parity blocks, while witness servers only hold a subset of the state held at parity servers and cache updates currently in-flight.*

In a standard deployment of Amalgam, each physical machine hosts a data server and a parity server process. However, for each stripe, the data server and parity server co-located on the same machine cannot both have constituent blocks. Therefore, if n is the total number of physical machines, $n \geq g + f$. In order to achieve a storage overhead that is less than what can be achieved by replication and in order to allow Amalgam to recover from failures, we only consider Amalgam deployments where $g > f$.

Data servers communicate with parity servers over a FIFO, lossless channel (i.e., over TCP or another protocol which handles re-transmission and message reordering in the case of dropped or out-of-order messages).[2] This ensures that state updates are received and applied in the same order, and is important to Amalgam's data recovery protocol.

[2]It is not strictly necessary that all messages be sent over this channel. New state updates and data recovery messages between the data servers and parity servers are the only messages that must be sent in this way.

The protocol as described in this section ensures that operations are committed and that committed updates are linearizable. If at-most-once execution of client operations is desired, it can be enforced through the standard means of a client table. Clients would number their operations sequentially, and data servers would keep track of clients' pending operations and their returned results. Client table information would be replicated and recovered as a part of the replication and server replacement protocols.

The state held at each server is summarized in Figure 3.6 and is referred to in the protocol descriptions below.

3.3.1 Normal Operations

For simplicity, we will describe the update protocol for a single data block. However, multi-block updates to blocks held at the same shard can be handled in almost exactly the same manner; we discuss multi-block updates in more detail in Section 3.3.4.

In order to commit an operation, the following protocol is run:

1. Client c sends its operation modifying data block b to the data server on which the block resides.

2. The data server computes the new value of the block, b'. It updates its local state, `blocks`, with the value b' and then calculates[3] the state *update* for the block $\Delta b = b' - b$. The data server then assigns a new operation number, $\texttt{op\_num}_{\Delta b}$, and sends $\langle \textsc{prepare}, \Delta b, \texttt{op\_num}_{\Delta b}, \texttt{primary\_num} \rangle$ to the parity servers for the block's stripe.

   The data server caches the operation, its `op_num`, and any output from the operation's execution in `pending_ops`.

[3] This update is calculated *in the arithmetic of the erasure code's finite field.* For the representation used by erasure coding libraries, addition and subtraction are both equivalent to bitwise `XOR`.

3. The parity servers, upon receiving a valid PREPARE message (i.e. one sent by the current data server — see Section 3.3.2), add the message to their `prepares` and respond with a PREPARE-OK message.

4. The data server waits for f PREPARE-OKs. Once these are received and all previous operations in `pending_ops` (ordered by `op_num`) are committed, the data server considers Δb committed. When an operation is committed, the data server replies to client c with the cached results of the operation.

   The data server also sends $\langle \text{COMMIT}, \texttt{op\_num}_{\Delta b} \rangle$ to the *parity servers*. Once a parity server receives a COMMIT, it updates its local `blocks` with Δb (as discussed in Section 2.3) and updates the corresponding entry in `op_nums`.

5. Once a PREPARE-OK has been acknowledged by *all parity servers* for the stripe, the data server additionally sends a PREPARE-DROP to the parity servers. The parity servers can then discard the PREPARE (once they have locally committed the operation). The data server also discards the operation from its `pending_ops`.

This message flow closely resembles other state machine replication protocols (e.g., Viewstamped Replication); however, unlike the backups in state machine replication protocols, the state stored by Amalgam parity servers is not identical. Therefore, pending operations cannot be discarded — even by the parity servers — until they are durable at all parity servers. Handling this case by transferring the state from one parity server to another is not possible, unlike in replication protocols where state updates can be applied to a local state machine and discarded as soon as the operation is durable because replicas can always transfer their entire state to one another in the case that updates are lost entirely during a leader change.

3.3.2 Server Replacement and Data Recovery

Amalgam can tolerate the failure of up to f blocks *per stripe*, and since each physical machine hosts at most one data/parity block for each stripe, Amalgam can tolerate the

failure of *at least f physical machines*; the contents of any data block can be read as long as $n - f$ machines remain available. We will now discuss how to recover from data server and parity server failures separately; however, replacing a physical machine hosting both processes will require replacing first the data server and then the parity server hosted at that machine. Moreover, the recovery protocol in Amalgam always replaces a server process with a new one; parity servers (and witness servers) are never "promoted" to data servers, as might happen in other consensus-based systems. This recovery protocol is more complex than in replicated systems because a recovering server must reconstruct its previous state. It is required that the data and parity blocks used in this reconstruction are *consistent* with each other and that the recovering server recovers metadata (e.g., `op_nums`) which is consistent with the recovered block.

When new data servers and parity servers are initialized, they are initialized into a *degraded* state. In order to fully recover into a *normal* state, they must rebuild all of their blocks using the data recovery protocol described below. While in a degraded state, they can participate in the sever replacement protocol, but they cannot participate in the data recovery protocol for blocks they have not yet recovered themselves.

For simplicity, server replacement in Amalgam is coordinated through a separate, fault-tolerant reconfiguration service (e.g., running on a Paxos cluster). This ensures that no two conflicting server replacement operations are executed concurrently. This is not strictly necessary — server replacement could be integrated into the core Amalgam protocol. However, reconfiguration is a notoriously tricky problem and would be all the more complicated in a heterogeneous system such as Amalgam. Moreover, reconfiguration is infrequent, and the data rebuilding process will be much slower than reconfiguration itself.

The reconfiguration service processes a single reconfiguration at a time; however, the replacement of a data server can preempt the replacement of a parity or witness server, as we will see. Importantly, the reconfiguration service ensures that reconfigurations happen in order. That is, even though the replacement of a data server can preempt the

replacement of a parity or witness server, upon the completion of a reconfiguration operation, the new server is initialized with configuration information reflecting all previously completed reconfigurations. In order to complete a reconfiguration operation, the reconfiguration service will ensure that it gets the necessary promises from servers *in the most recent configuration*, and the reconfiguration service ensures that servers process successive completed reconfigurations in order (e.g., by numbering them).

3.3.2.1 Replacing Data Servers

In order to replace a data server, the reconfiguration service must decide which of the old data server's operations were committed and which should be permanently dropped. It runs the following protocol:

1. The reconfiguration service first requests that the parity servers no longer accept messages from the old data server and send the reconfiguration service their local set of `prepares` for the data server's shard.

2. As soon as the reconfiguration service receives $p - f + 1$ acknowledgements[4], it constructs the set of committed operations with which to start the new configuration. First, it calculates the largest `op_num` executed by any parity server, τ. Then, it calculates the union of the `prepares` sets it was sent, $\mathscr{P}$. It then forms the largest contiguous chain of operations in $\mathscr{P}$, starting with τ, and removes from $\mathscr{P}$ any operations not in that chain. For example, if $\tau = 5$ and $\mathscr{P}$ contains operations with `op_nums` $\{1, 3, 6, 7, 9, 10\}$, the reconfiguration service would remove operations 9 and 10.

[4] $p - f + 1$ acknowledgements are not strictly necessary. Rather, the reconfiguration service needs acknowledgements from a quorum of the parity servers which *share stripes with the data server*, where a quorum is of size $p' - f + 1$ with p' being the number of parity servers sharing stripes with the data server to be replaced. This ensures that the replaced data server will no longer be able to commit operations to any stripe and that all operations committed by the old data server will be discovered during the replacement process. $p - f + 1$ acknowledgements from the full set of parity servers necessarily contains responses from a quorum from our restricted set of parity servers and thus is sufficient; we use this criterion in the protocol for simplicity.

The reconfiguration service sends the parity servers the new data server's ID as well as the set of PREPARES $\mathscr{P}$; the service saves this information until all parity servers have successfully processed the reconfiguration.

The reconfiguration service then initializes the new data server.

3. Parity servers processing a data server replacement first execute all operations up to the largest in $\mathscr{P}$ which apply to their parity blocks, using either their local `prepares` or the PREPARES sent by the reconfiguration service, and reset their entry in `op_nums` for that shard to 0. They also update their local data server ID for that shard and clear their `prepares` of any PREPARE messages for the replaced data server's shard. Once a parity server has processed the reconfiguration, it is ready to accept operations from the newly initialized data server.

   If a parity server is in a degraded state (see below), it simply discards PREPARE messages for blocks it has not yet recovered. These operations will be incorporated into the block the parity server recovers once it runs the data recovery protocol.

At this point, the new data server has been initialized and can participate in subsequent reconfigurations. However, the newly initialized data server contains no data — it is initialized in a *degraded* state. It must use the data recovery protocol we describe in Section 3.3.2.3 to rebuild its local state. In order to process operations, the data server must first recover the blocks the operation reads and writes; performance for this shard will be limited while this happens. It can, however, process operations for whichever blocks it has already rebuilt. The new data server can also recover additional blocks in the background. As soon as the data server has recovered all of its state, it is fully recovered and can once again contribute to the fault tolerance of the system.

This protocol is similar to the view change protocol in Viewstamped Replication [42, 58]. The calculation by the reconfiguration service in step 2 ensures that the new configuration is initialized with all operations which *could have been committed* by the previous data server (recall that data servers only consider an operation committed and send replies

to clients once all previous operations have also been committed). However, discarding higher-numbered operations when intermediate operations cannot be recovered is necessary because operations are propagated as state deltas rather than stored procedures. These state deltas may reflect the results of lost in-flight operations, and committing a later operation while discarding an earlier one could lead to linearizability violations.

One might think that it would not be possible to see missing intermediate operations in step 2 of this protocol because data servers communicate state updates to parity servers over FIFO channels. However, it is important to remember that depending on which stripe(s) an operation affects, a data server will use different parity servers to commit an operation. If operations to different stripes are in flight when a data server failure and replacement occurs, the reconfiguration service could indeed see gaps in the `op_nums` of the operations it is able to recover.

3.3.2.2 Replacing Parity Servers

The replacement of a parity server requires the approval of *all data servers*; if a data server is unavailable, it must first be replaced. Once all data servers agree to ignore future and locally cached PREPARE-OKs from the old parity server, a replacement parity server is initialized, once again in a degraded state. Once a new parity server has been initialized, the data servers are then updated with its identity.

If a parity server replacement is preempted because a data server is not available, the reconfiguration service can still use the old parity server as part of the quorum to replace a data server.

Parity servers will recover their data using the same protocol as data servers. However, they do not need to recover parity blocks before participating in the replication protocol; they can instead store committed operations until they have recovered the relevant blocks and can execute and apply the changes locally. They do not contribute to the fault tolerance of the system until they have fully recovered their blocks, though.

Data servers must first send new parity servers PREPARE messages for all operations in their `pending_ops` in `op_num` order before sending any other messages.

3.3.2.3 Data Recovery

Newly instantiated data and parity servers need to recover their data and parity blocks and will use the same protocol, described below, to do so. The data recovery protocol we describe can be used to recover a single stripe of data or multiple stripes simultaneously. The main purpose of this protocol is to ensure that the inputs to the decoding function of the erasure code are *consistent*; if each parity block does not have the same set of updates applied to it as all of the data blocks and other parity blocks, the decoding procedure could fail. This protocol can be run by any server, which we will refer to as the *recovering server*. The Amalgam data recovery protocol is similar in operation to the Chandy-Lamport consistent snapshot algorithm [10].

The protocol works as follows:

1. The recovering server first generates a unique nonce, ν, and sends $\langle$STRIPE-SNAPSHOT, `stripe-id`, $\nu\rangle$ to each of the data servers.

2. A data server, upon receiving the STRIPE-SNAPSHOT message, sends its data block for the stripe in question to the recovering server, along with its current `op_num`. Degraded data servers which have not yet rebuilt the block in question do not send a block to the recovering server but do send their current `op_num`.

3. The first time a parity server receives a STRIPE-SNAPSHOT message with nonce ν, it makes a local copy of its parity block for the stripe. It applies any pending PREPARE messages to this copy. The parity server waits until it has received a matching STRIPE-SNAPSHOT message from all data servers, updating the copy of the parity block with state updates from PREPARE messages it receives from data servers *from which the parity server has not yet received a matching STRIPE-SNAPSHOT message*.

Once the parity server has received these matching STRIPE-SNAPSHOT messages, it sends this copy of the parity block to the recovering server.

A degraded parity server that has not yet rebuilt the stripe waits for STRIPE-SNAPSHOT messages from all data servers and then sends a null response to the recovering server.

4. Once the recovering server has received responses from all servers (including itself) and at least g non-null data/parity blocks, it can use these blocks to decode the original stripe.

When a reconfiguration happens, servers discard any ongoing data recovery snapshots. Any data recoveries not completed must be retried by the recovering server in the new configuration. Moreover, parity servers do not accept snapshot requests from data servers not in their current configuration.

It should be noted that parity servers can use a copy-on-write approach for maintaining snapshot copies of parity blocks. Also, as an optimization, a recovering server can specify which (if any) parity servers to use if the recovering server knows which other data and parity servers are healthy and which are degraded. Unused parity servers can simply send null responses to the recovering server.

Because recovering servers wait for response messages from all data servers, we know that any data/parity blocks the recovering server receives must have been sent from servers *in the same configuration as the recovering server*. If two parity servers considered different data servers active for a given shard, at most one of those data servers would have received the initial STRIPE-SNAPSHOT token, and therefore at most one parity server would successfully send its parity block to the recovering server. Similarly, if two data servers from different shards considered different parity servers active (for the same parity blocks), at most one of those parity servers would receive STRIPE-SNAPSHOT messages from all data servers. Moreover, because the data servers communicate with parity servers over FIFO, lossless channels, the snapshot protocol guarantees that the blocks the

data servers and parity servers return have the same set of state updates applied to them. Each of these state updates is or will be committed, since they have been received by all parity servers.

The data recovery protocol is not guaranteed to complete successfully. If more than f servers are degraded and have not yet recovered a block, that data is permanently lost. Moreover, if a data server is failed, it must be replaced before data recovery can continue. However, when the protocol does succeed, we know that the blocks it returns are in a consistent state.

Server replacement in Amalgam is a potentially dangerous operation — if a server is only temporarily unavailable and is incorrectly replaced, it could lead to unnecessary data loss (e.g., if another failure occurs that puts Amalgam over its fault tolerance threshold while an unnecessary replacement is happening). An open direction for future work on the Amalgam protocol is to remove this limitation. Incorrectly replaced servers should be able to be swapped back in with their state intact.

Because a recovering data server must reconstruct a data block before reading or writing to it (and because the replacement protocol ensures that the previous data server can no longer commit operations and that all operations from previous data servers have either been permanently committed or discarded), it knows that the data block it recovers for a stripe has exactly those operations committed by previous data servers for its shard. However, the parity block a parity server recovers can potentially be written to by all data servers concurrent with the data recovery. Therefore, some mechanism is needed to infer which operations were and were not applied to the block the parity server actually recovers. Recovering parity servers retain the `op_nums` from the STRIPE-SNAPSHOT messages until their local `op_nums` are at least as large; parity servers do not execute operations for this stripe with `op_nums` smaller than the ones received during data recovery. Instead, they simply drop them. These recovered `op_nums` serve as temporary version numbers for a recovered parity block.

3.3.3 Integrating Replication Witnesses

As described in Section 2.2, consensus-based protocols can utilize replication witnesses to reduce the tail-latency of operations. And as previously mentioned, Amalgam also utilizes witnesses in the form of witness servers. These witnesses play a role similar to the acceptor nodes in Paxos Made Moderately Complex [67] and are unlike Harp's witnesses which only play a role during view changes [43]. In a standard deployment, each physical machine will also host a witness server process in addition to a data server and parity server. However, the same requirement still applies — at most one of the data/parity/witness servers can play a role in a given stripe.

The state held at witness servers is summarized in Figure 3.6. Their function is similar to parity servers; however, they hold no long-term state. They accept PREPARE messages from data servers. The data server only waits for acknowledgements from all parity servers (and not all parity and witness servers) before issuing a PREPARE-DROP, which allows both parity and witness servers to drop the PREPARE message.

They also participate in the data server replacement protocol; however, witness servers have no `op_nums`. If w is the total number of witness servers, the reconfiguration service waits for $w + p - f + 1$ total acknowledgements before completing a data server replacement. Witnesses cannot, however, participate in the data recovery protocol, as they do not store any long-term data. Witnesses servers are replaced in the exact same manner as parity servers — with the unanimous consent of the data servers.

3.3.4 Multi-block Operations

A data server can execute an operation which modifies multiple keys it hosts just as easily as it executes single-key operations. In order to do so, it sends the state updates for each of the keys to the parity servers for those stripes, and those parity servers may be different for different stripes. The operation is given a single `op_num`, but each parity server only receives the state updates for its own parity blocks. The data server only considers the

entire operation committed once *all of the state updates* are committed, i.e., acknowledged by at least f parity (or witness) servers.

Moreover, PREPAREs for multi-key operations are marked with metadata showing which blocks are modified. When the reconfiguration service replaces a data server and builds the set of committed PREPAREs from the previous configuration, it only considers a multi-key operation for inclusion if it can recover all of the constituent state updates. Otherwise, the operation was not committed and can be safely discarded.

3.3.5 Cross-server Transactions

Amalgam does not explicitly handle transactions across separate shards. However, the protocol presented above provides the abstraction of independent, fault-tolerant storage groups. Any protocol designed to coordinate and atomically commit transactions across fault-tolerant nodes can, in principle, be run on an Amalgam deployment, and there exist numerous such protocols [5].

Of course, atomic commitment protocols themselves rely on storage nodes to retain protocol state (i.e., COMMIT and ABORT decisions). For performance reasons, this protocol state should be replicated separately from the erasure coded state and will need to be handled by Amalgam reconfiguration and recovery protocols. Each parity server will maintain a separate, non-coded region of protocol state for each data server. Data servers can commit operations to this state normally, and during recovery, parity servers will forward this state to the reconfiguration service. The reconfiguration service will then initialize the new data server with the most up-to-date copy of its non-coded state.

3.4 Correctness

Now that the Amalgam protocol has been described in detail, we will argue that it is correct. Namely, we will argue that it fulfills the safety and liveness properties of state machine replication. As previously stated, Amalgam provides linearizable consistency for

the operations it receives and the results it returns. Amalgam will also continue to make progress and process new client operations, as long as the pattern of server failures and the network are sufficiently well behaved. We do not seek to give a full formal proof of correctness here. While the previous section described the important details of the Amalgam protocol in their entirety, the description is not complete enough to be amenable to formal proof. In this section, we will present arguments such that someone with sufficient familiarity with distributed systems would agree that the protocol as presented is indeed compatible with our safety and liveness properties.

3.4.1 Safety

We wish to show that Amalgam preserves *linearizability*, which is the strongest consistency requirement possible. The responses to clients' operations should appear as if a single, correct machine executes them in a sequential order that respects the real-time orderings of operations. In order to prove linearizability, we want to show two facts. First, we want to show that within a given configuration, Amalgam preserves linearizability. Then, we want to show that linearizability is preserved across configuration boundaries by the reconfiguration and data recovery protocols. In particular, we want to prove that the following invariant is maintained by Amalgam.

> **Invariant:** If a state update, Δb, is *committed*, then there exists a set of at least $f + 1$ servers where each server either has a pending PREPARE with Δb, has a data block or parity block to which Δb has been applied, or has a data block or parity block for Δb's stripe that is degraded. Also, if a state update Δb is committed, then any earlier state update executed on the same shard of data is also committed.

Because data servers always commit state updates *in the order they were executed*, if this invariant is maintained by Amalgam, then the system preserves linearizability. Note that this invariant does not consider cross-server transactions. As discussed in Section 3.3.5, Amalgam is compatible with existing atomic commitment protocols, and if

Amalgam preserves linearizability of single-shard operations, then cross-shard transactions will be linearizable when Amalgam is integrated with a linearizable transaction layer.

First, we show that in a given configuration, the invariant is maintained. The invariant is trivially maintained by the normal operation commit protocol. Data servers wait for at least f responses before considering an operation committed and replying to the client. This ensures that $f+1$ total servers (including the data server itself) at least have a copy of the state update, Δb. Moreover, the PREPARE message is only discarded once *all parity servers* (of which there are at least f) have acknowledged receipt of Δb, and when parity servers discard a cached PREPARE message, they apply the state update to the corresponding parity block.

Next, we need to show that the invariant is maintained across configurations. As described in Section 3.3.2, the reconfiguration service ensures that the systems goes through a sequence of configurations $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots$ where C_0 is the initial configuration of the system and the only difference between C_i and C_{i+1} is the replacement of a single server. When the reconfiguration service replaces a parity server, it first ensures that *all* data servers (which are the same servers in the old and new configurations) no longer consider PREPARE-OK messages from the old parity server as valid. When the new parity server is initialized, all of its parity blocks are initialized in a degraded state; therefore the invariant is preserved. Moreover, because messages from the old parity server are ignored by the new data servers, they will not commit operations without a quorum in the new configuration, even after the new parity server recovers its parity blocks.

When the reconfiguration service replaces a data server, it requires acknowledgements from $p-f+1$ parity servers (or $p+w-f+1$ parity/witness servers, where w is the number of witness servers). Consider a set of parity servers C used to commit a state update Δb to a shard and a set of parity servers R used by the reconfiguration service to replace the shard's data server. $|C| \geq f$ and $|R| \geq p-f+1$; therefore $|C|+|R| \geq f+p-f+1 \geq p+1 > p$. Because p is the total number of parity servers,

these sets must intersect in at least one server. Therefore, during the reconfiguration, the reconfiguration service will discover Δb or Δb will have been applied to all parity blocks belonging to servers in R. Because data servers commit operations in order, the same is true of all state updates that preceded Δb as they must have been committed as well. The reconfiguration service might discover other operations which had not yet been committed by the old data server, but upon entering the new configuration, all parity servers will apply these non-committed updates and they will be permanently committed to the shard's state. Importantly, the reconfiguration service ignores state updates with `op_nums` not contiguous with previously committed updates and updates discovered during reconfiguration. These updates could not have been committed (since there are previous non-committed updates implied by the holes in the sequence of `op_nums`), and committing them would violate our invariant. The reconfiguration does preserve our invariant, however.

Finally, we need to show that our invariant is preserved by the data recovery protocol. First, the data recovery protocol must ensure that the blocks retrieved are *consistent*. As explained in Section 3.3.2.3, the snapshot protocol, when it completes successfully, guarantees that the participating servers (all of the data servers and some subset of the parity servers) were in the same configuration. Both state updates and STRIPE-SNAPSHOT messages are sent over FIFO, lossless channels. If a data block retrieved by the snapshot protocol has a state update applied to it, then the corresponding PREPARE message to each of the parity servers will precede the STRIPE-SNAPSHOT message on their FIFO links from that data server. Therefore, any parity block retrieved must also have that state update applied to it. Conversely, if a parity block has a state update applied to it, that state update was either sent by the data server from the current configuration or was committed in a previous configuration. If the state update was sent by the current data server, then it must have preceded the STRIPE-SNAPSHOT message from the data server to the parity server, and the data server will have applied it to its own data block (which means that all parity blocks retrieved will reflect this update, as previously shown). If the state update

was applied during a previous configuration, then the reconfiguration servers must have ensured that all parity servers were sent and applied the corresponding PREPARE messages before starting the current configuration.

Finally, all that remains to be shown is that any block recovered by the data recovery protocol reflects all previously committed state updates. This follows by induction. Any blocks retrieved by the snapshot protocol necessarily were sent by servers whose blocks for that stripe were not degraded. If, before the data recovery procedure began, the state update in question was already committed, then the blocks sent to the recovering server reflected that state update by induction, and so will the recovered block. If the committing of a state update is concurrent with data recovery, then the recovering server must be a parity server, as data servers do not commit operations until they have recovered the relevant data blocks. For the state update to not be reflected in the block recovered by the parity server, it must be applied to the data server's data block *after it sends the* STRIPE-SNAPSHOT *message to the recovering parity server*. In fact, it must send the PREPARE for the update in question after the STRIPE-SNAPSHOT to all parity servers, implying that the operation commit and the data recovery are not actually concurrent. The only way the data server can commit the state update is by receiving at least f PREPARE-OKS (possibly including the recovering server), the parity server's data recovery will not violate our invariant.

3.4.2 Liveness

The objective for any state machine replication protocol is that it continues to process clients' operations. The criteria under which Amalgam achieves this goal will be different from protocols built for the crash-stop or crash-fail failure models. Those protocols can require that no more than f servers fail, for their own values of f. Once f total failures have occurred, these protocols will become permanently unavailable. Amalgam allows failed server to be replaced, and whether Amalgam experiences permanent failure and data loss or whether it can continue to process new client operations can depend not only

on the number and pattern of failures but on how quickly failed servers can be replaced and their data recovered. In particular, we will show that Amalgam can continue to make progress if the following condition is met.

> **Server Liveness Criteria (SLC):** For all times t, the number of failed and non-replaced data servers and the number of failed and non-replaced parity servers is less than or equal to f. Also, for all stripes s, the number of servers at time t with either a parity block or data block in stripe s which have failed and not yet been replaced or whose block in stripe s is in a degraded state is less than or equal to f.

Showing that Amalgam *can* make progress under this condition is straightforward; all we want to show is that as long as SLC holds, Amalgam never reaches a state from which progress is impossible. We only need to construct a path from any reachable state to a progress-making state, which can happen as follows. First, the reconfiguration service replaces any failed data servers, which it can do since no more than f parity servers have failed and not yet been replaced by our assumption that SLC holds. Then, the reconfiguration service replaces any failed parity servers. If the data server for our new client operation has any degraded blocks for the relevant stripes, it can recover those, and the data recovery protocol will succeed by assumption. Finally, the normal operation commit protocol can succeed.

This shows that Amalgam never gets permanently stuck, unless it suffers too many simultaneous failures. However, this argument relies on "luck" from both the network and the incidence of new failures. Proving that Amalgam will make progress under all conditions in an asynchronous system in which server failures can occur is not possible [14], even if we assume that SLC holds. However, we would like to show something slightly stronger about the liveness of Amalgam. In particular, we want to show that if the network is sufficiently well-behaved (and SLC holds), then Amalgam will make progress.

We first assume that our network is fair-lossy; that is, if a message m is sent infinitely many times, then it is delivered infinitely many times. Then, we assume that each of our servers is equipped with a $\diamond P$ failure detector [9]; every failed server is eventually permanently suspected to be failed by every non-failed server and eventually no non-failed server is suspected by any non-failed server.[5] We also assume that servers continually retry messages until they are acknowledged, either with the correct responses as dictated by the Amalgam protocol or with a message layer acknowledgement. Given our assumption of fair-lossy links, we can simplify and assume that protocol messages are eventually delivered. Given these assumptions (as well as the SLC assumption), the argument that Amalgam guarantees liveness is straightforward. The reconfiguration service replaces servers it suspects of having failed based on the $\diamond P$ failure detectors on the processes running the service. Eventually, these failure detectors will be completely accurate. After this point in time, the reconfiguration service will replace all failed parity and data servers. Furthermore, any degraded blocks can be recovered by the data recovery protocol, which will eventually succeed. Finally, the normal operation commit protocol will eventually succeed.

There is one hidden assumption in SLC. Namely, SLC assumes that for all times t, that no stripe is permanently lost because more than f of its blocks are degraded or were on servers which have failed and not yet been replaced. If exactly f blocks in a stripe are currently degraded and have not yet been recovered by the data and parity servers and the reconfiguration service incorrectly replaces another server holding a block for that stripe, the stripe would be permanently lost. As mentioned in Section 3.3.2.3, this means that reconfiguration is a potentially dangerous operation that could turn a perfectly healthy Amalgam execution into one in which progress is impossible. Whether one views

[5] The $\diamond P$ failure detector is stronger than the weakest failure detector that is able to solve consensus, $\diamond W$ [8]. However, Amalgam uses a multi-leader approach (where every data server acts as the leader for its shard of data), which suggests that merely guaranteeing that a single non-failed server is eventually never suspected is insufficient. Whether or not Amalgam, or a system very much like it, could guarantee liveness with only $\diamond W$ failure detectors is direction for future work.

this condition as an assumption that should be made of executions of Amalgam or as an obligation that the reconfiguration service should fulfill is a matter of perspective. Another way of characterizing this requirement is by stipulating that the failure detectors belonging to the reconfiguration service are even stronger (P failure detectors would avoid the problem of incorrectly suspecting non-failed servers entirely). It should be noted that this problem of reconfiguration being potentially dangerous is not unique to Amalgam and is present in many protocols which support reconfiguration.

Lastly, it should be noted that if SLC holds, then a protocol which never replaces servers could be equally as good as Amalgam at achieving liveness. If servers are never replaced, then SLC dictates that no more than f data servers and parity servers ever fail. This would be unsatisfactory, as one of the goals of Amalgam is to support long-term deployments which should be able to recover from the failure of individual components. However, as we have just seen, Amalgam can replace and does replace servers (given fairness assumptions and assumptions about the well-behavedness of failure detectors). Even still, whether or not SLC holds in a given execution with a given pattern of failures can depend on the speed at which server replacement and data recovery happens. This is unavoidable, but as we will see in the following section, we can quantify and analyze the effects of data recovery speed on the overall resilience of the system.

3.5 Data Durability

The benefits of using erasure coding in a distributed system might seem clear; coded data uses less total storage than replicated data (depending on the parameters for the erasure code). However, we must be careful to *also* compare the fault tolerance of Amalgam to designs like ShardedPaxos. After all, an easy way to reduce the storage overhead of ShardedPaxos is to reduce the replication factor. At the extreme, running a system without any backups at all yields the best possible storage overhead but turns any failure into a

catastrophic one. If we give up fault tolerance to lessen storage overhead, then we haven't gained anything.

Amalgam can only lose f servers in each stripe at a given time. If stripes are *full width* (i.e., each physical machine plays some role for each stripe), then Amalgam can only lose f servers total before suffering data loss. ShardedPaxos, on the other hand, can survive the failure of f replicas in each of its Paxos groups. Here we will not show that *all* configurations of Amalgam yield storage overhead benefits without sacrificing fault tolerance. Rather, our goal in this section is to point to a general strategy for assessing the fault tolerance of Amalgam and show that with reasonable parameters, a configuration of Amalgam can give excellent storage overhead and fault tolerance.

When Amalgam is deployed with g data servers and f parity servers (and no witnesses) per stripe, it can survive the failure of any f out of $g+f$ machines. Whereas, a ShardedPaxos deployment of g Paxos groups of $2f+1$ replicas each on a separate machine can survive the failure of any f out of $2f+1$ replicas in all of its replica groups. Clearly, for fixed values of f and g, ShardedPaxos is more resilient, but how much more resilient?[6] And by how much would you have to increase f to get an equally resilient deployment of Amalgam? To answer these questions, we invoke the standard metric of mean time to data loss (MTTDL) that dates back to the original RAID paper [62]. Burkhard and Menon gave a useful approximation for the MTTDL of a system of g separate groups of n identical components, each group tolerating f failures [6].

$$\text{MTTDL} \approx \frac{\text{MTTF}}{g(n-f)\binom{n}{n-f}} \left(\frac{\text{MTTF}}{\text{MTTR}}\right)^{f}$$

[6] A ShardedPaxos system with each replica on a separate machine is not completely analogous to Amalgam. Rather, a more comparable deployment of ShardedPaxos would have each machine host a single leader replica and $2f$ followers, each in a different Paxos group. This case is harder to analyze numerically, but it is less fault-tolerant than a single-process-per-machine ShardedPaxos and more fault-tolerant than Amalgam. In particular, if $g > 2f+1$, ShardedPaxos with co-located replicas can tolerate more than f failures, whereas Amalgam cannot. A completely disaggregated ShardedPaxos deployment does represent a best case for the fault tolerance of a replicated system and is therefore a worse case as far as comparison with Amalgam is concerned.

Here, MTTF is the mean time to failure of a single component, and MTTR is the mean time to replacement/recovery of a failed component. This formula is based on a Markov model of component failure and is a good approximation when MTTDL $\gg$ MTTR.

Using this MTTDL formula, we can compare the storage overhead for a particular configuration of Amalgam to a similar configuration of ShardedPaxos, safe in the knowledge that we didn't sacrifice data resilience. Before we start plugging in various points, however, we should note that MTTR is dependent on system design. Replacing a failed node in ShardedPaxos is a relatively simple affair; the data can be directly transferred from another replica. The amount of data needed to be transferred is approximately $\frac{1}{g}$ of the total stored data (not accounting for replication). However, if we use an erasure code like Reed–Solomon, Amalgam needs to read the equivalent of all of the data stored in the system to restore a single server. Naïvely reading this data serially would mean that the MTTR is a factor of g larger for Amalgam, which we can see will dramatically reduce the MTTDL. Even with this worst-case assumption, however, we see that erasure coding yields benefits; if MTTR is held constant between Amalgam and ShardedPaxos (i.e., if reading is done in parallel and other servers help parallelize the task of reconstruction) the benefits are even more dramatic.

For the sake of example, let's assume: that our servers have a MTTF of 1 year, that each server in our system holds 256 TB and is connected by a 40 Gbps network, and that a recovering node can write data locally at the full 40 Gbps rate (if recovering servers are bottlenecked by writing their data locally, this is only more favorable to Amalgam). This give an MTTR for ShardedPaxos of approximately 14 hours. An ShardedPaxos deployment of 10 groups of 5 servers each ($f = 2$) would therefore have an MTTDL of approximately 1,300 years and a storage overhead of 2. A similar (naïvely recovering) Amalgam deployment with 10 data servers and 2 parity servers per stripe, deployed on 12 machines would only have an MTTDL of 6 years. However, if we increase the number of parity blocks from 2 to 4 (and the number of machines from 12 to 14), we get an MTTDL of 1,400 years, with a storage overhead of just 0.4. If we assume optimal recovery for Amalgam, we

get a similar level of resilience with just 3 parity blocks per stripe and an overhead of 0.3. The particular values of MTTF and MTTR chosen here do not matter. As long as MTTDL $\gg$ MTTR, we see dramatic improvements in MTTDL with small increases to f, and Amalgam retains a large storage reduction when compared to a similarly resilient deployment of ShardedPaxos.

3.6 Evaluation

In evaluating Amalgam, we will answer the following questions: First, does Amalgam provide performance similar to a ShardedPaxos baseline on simple read/write workloads as well as read-modify-write workloads? And second, what are the performance characteristics of Amalgam while it is recovering from a failure — how quickly can an Amalgam server rebuild its state and what kind of service can a degraded server provide while it is rebuilding its state? We will analyze the storage overhead reduction provided by Amalgam in light of these results in order to quantify the trade-offs Amalgam provides when compared to a replicated system and an alternative erasure-coded system, RSPaxos. As mention in Section 3.1, RSPaxos is a system which does not utilize linear updates to parity blocks but rather breaks apart individual values sent by clients, encodes those pieces to produce parity information, and sends the data pieces and parity blocks to individual replicas.

3.6.1 Experimental Design

Amalgam was implemented in Rust, and ShardedPaxos and RSPaxos were implemented in the same framework for the sake of comparison. All experiments are run on 20 Microsoft Azure VMs equipped with 8 virtualized CPUs and 32 GB of memory. The VMs are connected with 12.5 Gbps links and are capable transferring this full bandwidth between any pair. These experiments are run on in-memory data which is not persisted to stable storage.

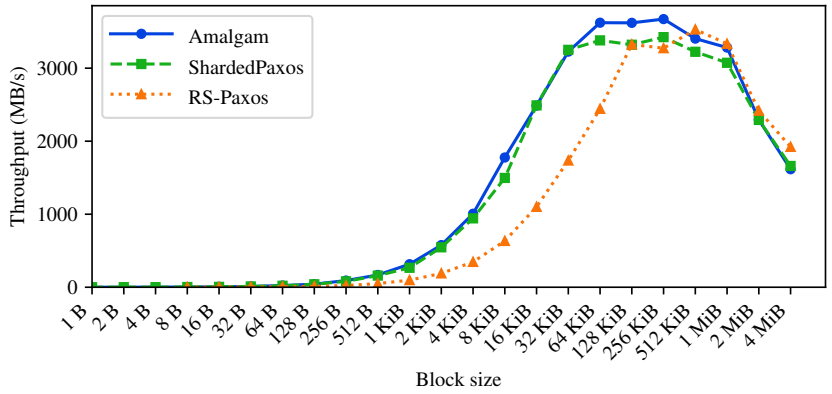

Figure 3.7: *The maximum throughput attained by Amalgam, ShardedPaxos, and RSPaxos as a function of the amount of data written with each request.*

Unless otherwise noted, each experiment is run allowing for two complete machine failures (i.e., $f = 2$). For Amalgam, this means that two parity shards are designated for each stripe, while ShardedPaxos is run with $f + 1$ full replicas in each group. RSPaxos is run with the required $3f + 1$ replicas in each group.

3.6.2 Results

Block Size Amalgam, ShardedPaxos, and RSPaxos are all sensitive to the size of the blocks being written. First, we evaluate all three systems across a range of block sizes to determine the optimal size to use to attain the highest throughput. We run each system with 15 groups on 15 machines, co-locating processes from different groups on the same machine.

Figure 3.7 shows that throughput is quite limited for all systems at lower block sizes — the messaging overhead dominates. Throughput then increases dramatically before

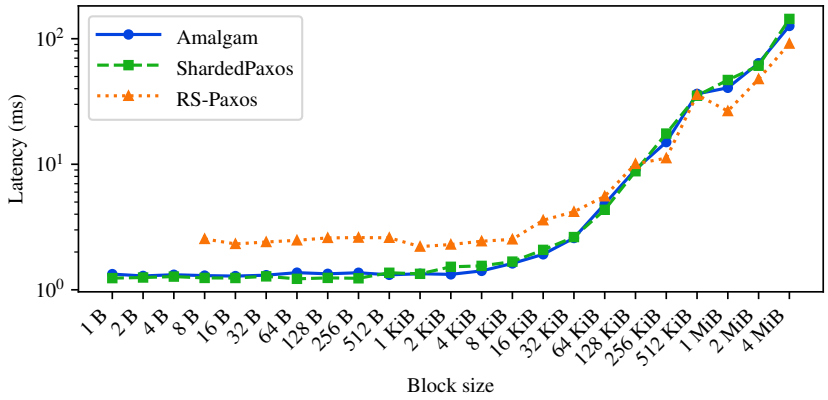

Figure 3.8: *The latency of writing a single block of data to Amalgam, ShardedPaxos, and RSPaxos as a function of the amount of data being written with each request.*

peaking around 128 KiB blocks for all systems. Across a range of block sizes, however, we see that Amalgam does indeed achieve throughput similar to ShardedPaxos.[7]

Figure 3.8 shows that the latency of Amalgam and ShardedPaxos are comparable across a range of block sizes. On the other hand, the latency of RSPaxos is significantly higher when the block size is small — again, the messaging overhead dominates and the leader of an RSPaxos group must send and receive more messages due to its larger group size. However, with 256 KiB blocks and larger, RSPaxos is able to lower its latency below that of the other two protocols because it reduces its bandwidth usage by splitting apart data items.

Scalability Next, we fix the block size at 128 KiB, which was optimal for all three systems, and examine the performance of Amalgam, ShardedPaxos, and RSPaxos as the number of groups increases. For each configuration, we allocate server machines equal to the

[7] In fact, we see that Amalgam's peak performance *exceeds* that of ShardedPaxos. This is not experimental noise but is rather due to subtle differences in implementation and the experimental design. In particular, in a deployment of Amalgam, each machine hosts one data server and one parity server process. However, in our implementation of ShardedPaxos, each replica is run in its own process, and each machine hosts g replicas, where g is the number of replica groups — one leader and $g - 1$ followers. This could introduce slightly more inefficient scheduling. On a write-only workload, there are no protocol-level reasons Amalgam's performance should be better than a replicated system.

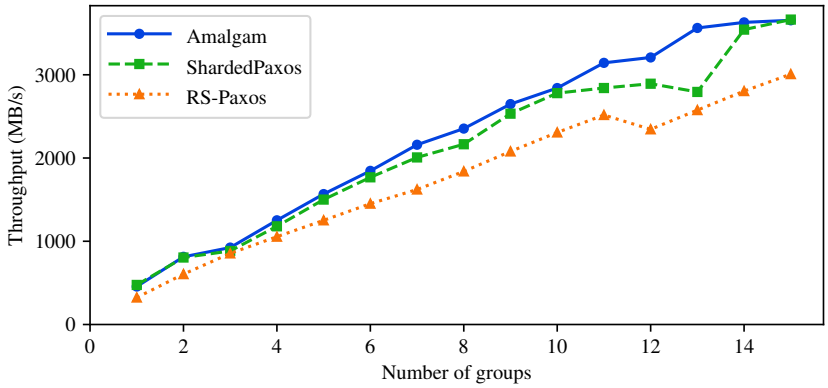

Figure 3.9: *The maximum throughput attained on a write-only workload as the number of groups increases.*

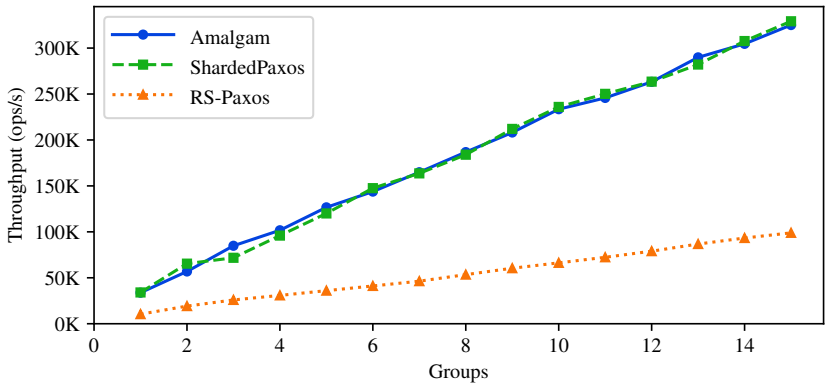

Figure 3.10: *The maximum number of operations from a write-only, small block workload each system is able to process as the number of groups increases.*

number of groups or the number of replicas in each group, whichever is larger. Figure 3.9 shows the results of this experiment. All three systems scale linearly (or close to linearly), and once again Amalgam exhibits very similar performance to ShardedPaxos.

We also test the scalability of all three systems on a small (64 B) block workload. Figure 3.10 shows the results. Once again, all three systems scale well, and Amalgam's performance is very similar to our replicated system. RSPaxos lags behind the other two more significantly as small block workloads are bottlenecked by the message complexity, and RSPaxos groups are larger than Amalgam or ShardedPaxos.

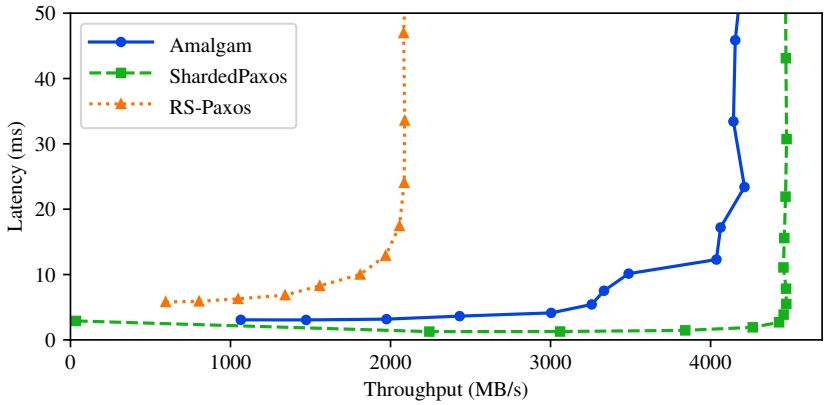

Figure 3.11: *Throughput vs. latency on a read-modify-write (RMW) workload. Clients issue requests that a single 128 KiB block should be modified. Throughput is reported as the total amount of data being read and modified.*

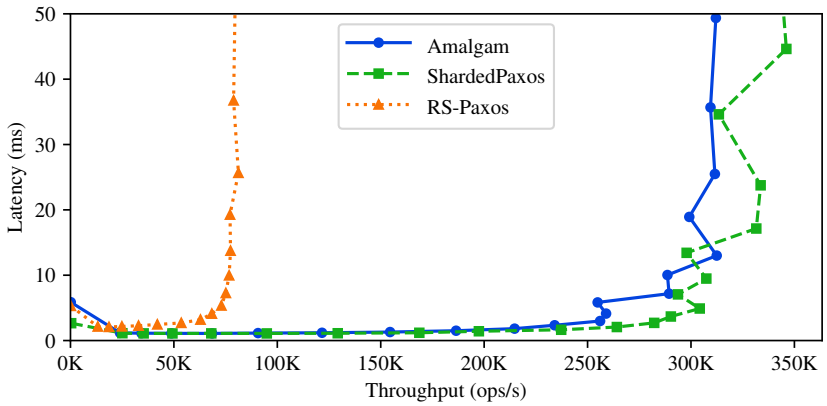

Figure 3.12: *Throughput vs. latency on a read-modify-write workload with 64 B blocks. Clients issue requests that a single block should be modified.*

Read-Modify-Write Operations Next, we examine how all three systems perform on more complicated workloads. For these experiments, clients send requests to read a block of data, modify it (in this case, we XOR the block with some junk value, but the operation is treated as a black box), and write back the results to the system's storage. In Amalgam, this operation is treated as any other operation; the data server computes the change to the data blocks and propagates those changes as normal. In ShardedPaxos, however, an operation can be committed to a Paxos group's shared log as a stored procedure which each

replica can then independently apply to its state. When operations can be represented compactly, this allows ShardedPaxos to reduce inter-node network traffic compared to Amalgam, which is an important trade-off between Amalgam and replicated approaches. RSPaxos does not support RMW operations. We implement these here by making them client-coordinated; the client issues sequential `GET` and `PUT` operations to simulate a single RMW operation. If isolation between concurrent RMW was desired, this approach would have to be augmented with a locking mechanism.

Figure 3.11 shows the result of our experiment, again run with 15 groups and server machines with 5 additional machines providing client load. Here, Amalgam does underperform ShardedPaxos but only by a small margin; this is due to ShardedPaxos's ability to compactly represent stored procedures and commit them as such. RSPaxos, on the other hand, significantly lags behind the other two due to the multiple protocol rounds necessary to commit a single operation. Figure 3.12 shows the result of the same experiment run with 64 B blocks rather than 128 KiB blocks. The results largely mirror the previous experiment with RSPaxos's performance being further reduced compared to the other two protocols.

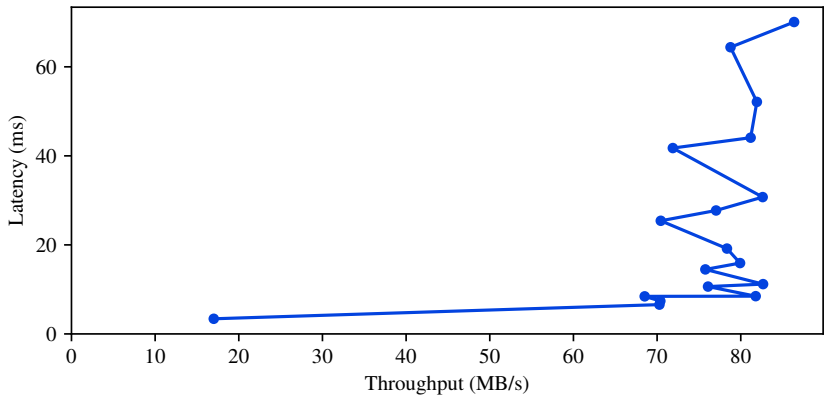

Figure 3.13: *Throughput vs. latency of stripe recovery operations issued by a recovering server as it is rebuilding.*

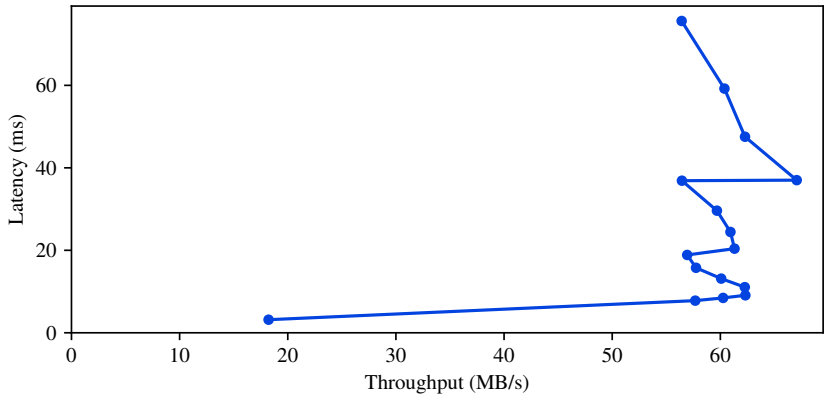

Figure 3.14: *Throughput vs. latency of stripe recovery operations issued by a recovering server as it is rebuilding while 1024 closed-loop clients are issuing writes to data blocks randomly chosen among the 15 data servers* (*including the recovering one*).

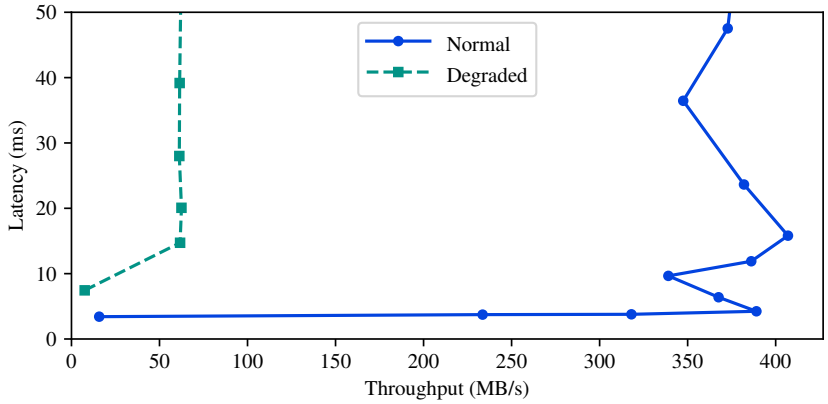

Figure 3.15: *The throughput and latency a single Amalgam server can provide normally and when it is in a completely degraded state.*

Recovery Speed and Degraded Performance We are also interested in how well Amalgam can recover from a failure and restore the system from a degraded state. First, we want to know how quickly a new data server (or parity server) can recover the blocks for its shard. Figure 3.13 shows the throughput and latency a recovering server is able to achieve in a group of 15 servers, where each stripe consists of 13 data blocks and 2 parity blocks, and each block is again 128 KiB. Of course, in realistic scenarios, there will be some amount of traffic to the system during recoveries. Figure 3.14 shows the result of the same

experiment with some amount of background traffic writing to randomly chosen data blocks (including those held at our recovering server, which need to be recovered first before being written). In these experiments, we see that Amalgam can achieve rebuild rates of approximately 80 MB/s and 60MB/s respectively, compared to the more than 3 GB/s at which it is able to write data blocks. Data reconstruction time is one of the major costs of erasure coding, and these speeds are significantly slower than how quickly ShardedPaxos can initialize a replacement replica, which should be limited only by the data transfer throughput between replicas.

Figure 3.15 shows the throughput and latency a single data server in our 15 server system can provide in its normal state and when it is degraded and must rebuild each block on-the-fly. The throughput this server can serve is significantly decreased when it is in a degraded state. By contrast, ShardedPaxos replicas hold full copies of each shard of data. If the leader of a ShardedPaxos replica fails, a backup can take over immediately and does not need to go through the expensive process of rebuilding its state. As soon as the leader election is completed, the new leader replica of a ShardedPaxos shard can serve requests with the same performance characteristics as before. If a new follower is to be initialized, it will not be on the critical path for committing new operations, and its state can be initialized by another follower.

Stripe Size Lastly, we examine the effects of stripe size on Amalgam. For this experiment, we run Amalgam on 15 servers and keep the number of parity blocks per stripe constant at 2 while increasing the number of data blocks per stripe. Figure 3.16 and Figure 3.17 show the results. We see that, for these modest stripe sizes, the effects on throughput are not noticeable while latency increased but not dramatically. The storage overhead for a $13 + 2$ stripe is just 0.15.

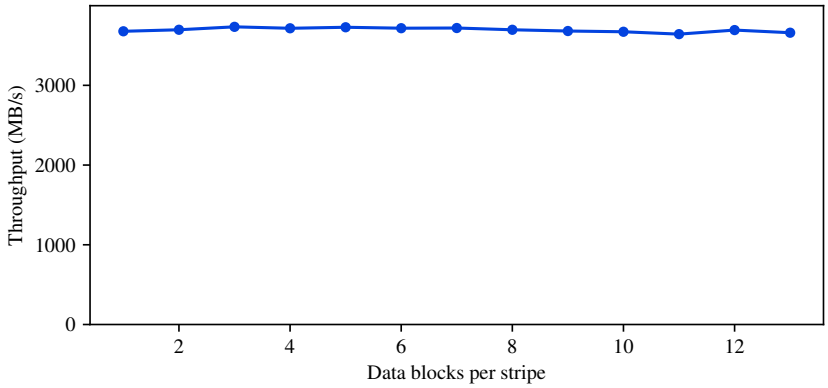

Figure 3.16: *The throughput attained by Amalgam run on 15 servers as a function of the number of data blocks per stripe while the number of parity blocks per stripe remains constant at 2. Clients issue write requests to 128 KiB blocks.*

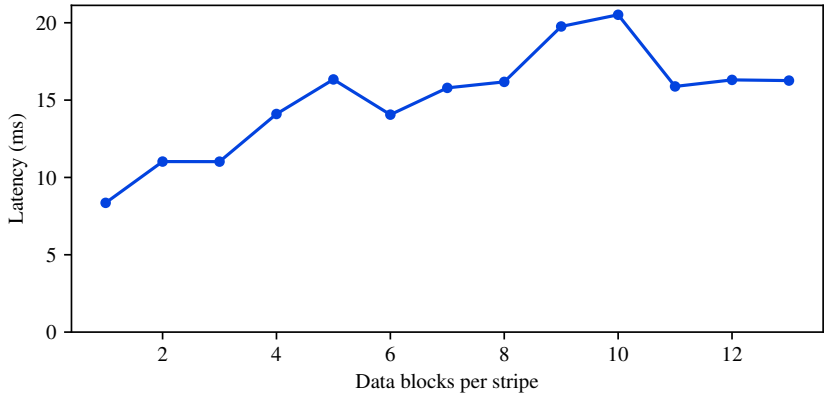

Figure 3.17: *The latency attained by Amalgam run on 15 servers as a function of the number of data blocks per stripe while the number of parity blocks per stripe remains constant at 2. Clients issue write requests to 128 KiB blocks.*

3.6.3 Discussion

Amalgam is able to achieve performance similar to ShardedPaxos, including on read-modify-write workloads. However, we do see one interesting trade-off between the two systems when ShardedPaxos was able to represent RMW operations more compactly and could achieve slightly higher performance than Amalgam. RSPaxos, on the other hand,

while it can achieve considerable throughput on write-only workloads, was not able to provide the same kind of performance on RMW workloads.

Amalgam, in our testing environment, was able to rebuild the state of a single data server at a maximum rate of 80 MB/s. Moreover, when a data server is in a degraded state, the throughput and latency it can process regular operations at dramatically worsened. Long rebuilding times and worse performance during the rebuilding process are one of the biggest drawbacks of any system which utilizes erasure coding and must be carefully considered before any such system is deployed.

3.7 Related Work

The most closely related work to Amalgam is Cocytus [12], which also uses linearity of erasure codes to update parity blocks. Cocytus does not support RMW operations and, unlike Amalgam, runs separate data server and parity server processes for each coding group. That is, whereas Amalgam data servers might communicate with different parity servers depending on the stripe of the data they are modifying, Cocytus data servers always communicate with the same parity servers. The number of coding groups is dependent on the number of physical machines, so the number of data server processes (and consequently the number of shards of data) increases with the square of the number of machines. The Amalgam protocol uses a single data server and parity server process per machine and therefore allows for a much coarser partitioning of data. This enables many more opportunities for multi-block RMW operations while Amalgam scales to larger deployments.

RSPaxos [55] utilizes erasure codes to provide a strongly consistent key–value interface. RSPaxos utilizes $3f+1$ machines in its normal configuration with a quorum of size $2f+1$, ensuring that and committed writes can be successfully read. However, RSPaxos splits data items into separate chunks and distributes them among servers, which is incompatible with the goal of supporting efficient RMW operations.

Linear Erasure Codes for Parity Updates Other distributed systems have utilized linear block codes to update parity blocks. Aguilera et al. [1] propose a protocol for a client-driven erasure-coded distributed block storage system. Clients read and write data blocks held at storage nodes, where write operations swap in the value to be written and return the old value of the data block. Clients use this information to update the parity blocks in a stripe. This protocol only provides a read/write interface, however, and does not support RMW operations.

MemEC [74] is key–value storage system which utilizes erasure codes. MemEC increases the storage savings due to erasure codes by encoding both keys and object metadata in addition to values. MemEC stores data in fixed-size chunks, and servers maintain two indexes locally: one mapping chunk IDs to chunk references and another mapping object keys to locations in data chunks.

Myriad [11] is a system providing geo-redundant storage. Myriad uses a two-phase commit protocol to commit updates and utilizes version vectors to ensure consistency during data recovery.

LH^*_{RS} is a protocol that augments distributed Linear Hashing (LH^*) [44] with erasure codes. As in Amalgam, changes in stored data are propagated as state updates in both LH^*_{RS}, Myriad, and MemEC.

Local Storage Arrays Erasure codes have also been used extensively in storage systems. The original RAID paper [62] described a method for grouping multiple physical disks attached to the same I/O bus into a single logical storage array and described distinct RAID levels with varying degrees of fault tolerance. RAID 6, which was invented later, utilizes block codes (e.g., Reed–Solomon) to allow for the complete failure of two disks. RAID arrays can be controlled in hardware or software; the RAID controller is responsible for updating parity information when new data is written and for rebuilding disks after failures.

AutoRAID [70] expanded on the RAID idea by creating a two-tiered storage hierarchy, wherein commonly accessed data is replicated and less frequently accessed data is erasure

coded. Both tiers, however, are stored on the same physical array of discs. A software controller transparently handles read and write access to the AutoRAID array, migrating data between the storage tiers as necessary.

Network Attached Storage Arrays Petal [38] is an architecture for a network-attached storage system. Clients communicate with a set of storage servers, each of which contains one or more physical disks. Data in Petal is mirrored for redundancy, but Petal utilizes a *chained-declustering* data distribution scheme, where each original data block is mirrored to the next server in a cyclic fashion (a scheme similar to RAID 5's or Amalgam's distribution of parity blocks). Frangipani [66] is a network-based distributed file system built on top of Petal, which utilizes a global locking system based on Paxos to handle simultaneous updates to files.

RAID-x [26, 27] is another example of a distributed software RAID system. Data in RAID-x is mirrored across network attached storage nodes in a scheme called *orthogonal-striping and mirroring*. Data consistency is maintained through locks in device drivers.

Cloud Storage Systems Finally, erasure codes have been used extensively in cloud storage systems [7, 13, 24, 25, 31]. Windows Azure Storage [25] uses local reconstruction codes for its streaming layer. Its design uses an asynchronous approach to erasure coding. The streaming data model is an append-only log, which clients can write to and read from — but not modify. Periodically, sections of these logs are erasure coded and distributed to storage servers by a stream manager service. Local reconstruction codes are of particular interest. They reduce the cost of rebuilding a failed server (though at the cost of increased storage overhead). Local reconstruction codes are generally compatible with Amalgam; though they would have to be taken into account when a recovering server chooses which existing servers to recover from.

Giza [13] is another Microsoft system which works in a similar manner to RSPaxos — by splitting data items into blocks and generating parity blocks, each to be distributed. Giza, however, specifically targets the cross-data center setting for cloud storage.

Pahoehoe is a key–value cloud storage system for storing large values which utilizes erasure codes for durability. Pahoehoe provides a PUT/GET interface; moreover, unlike Amalgam which guarantees linearizability of clients' operations, Pahoehoe only guarantees eventual consistency.

3.8 Conclusion

This chapter presented Amalgam, a new protocol which leverages the linearity of erasure codes to efficiently commit updates to sharded block storage. Amalgam can commit arbitrary transactions spanning multiple data blocks with the same number of network delays as traditional, replicated protocols. It guarantees linearizability, the strongest possible consistency condition and can tolerate a configurable number of complete machine failures. When machines fail, the data server, parity server, and witness server on those processes are replaced by a reconfiguration service, and the newly instantiated data servers and parity servers rebuild the missing state using the decoding function of the erasure code. Amalgam uses a snapshot-style recovery protocol to ensure that the inputs to the decoding function are *consistent* and that the associated protocol metadata is also recovered. Because Amalgam is a state machine replication protocol, it cannot guarantee responses to clients' commands in all cases — even when failures remain absent. However, we have argued that Amalgam does guarantee responsiveness to clients' requests given reasonable assumptions.

When compared to ShardedPaxos, a system composed of multiple Paxos groups, Amalgam performs well across a range of workloads. On write-only workloads, Amalgam's offered latency and throughput is as good while it performs nearly as well on read-modify-write workloads. On read-modify-write workloads, Amalgam significantly outperforms systems utilizing erasure codes which split up data objects because those systems must first reconstitute objects in one location before modifying them. Amalgam does this while achieving a significant storage overhead reduction. Moreover, Amalgam does not sacrifice

fault tolerance in the name of reduced storage utilization. Amalgam can provide data durability comparable to replicated systems while utilizing erasure coding layouts which dramatically reduce its storage footprint.

Amalgam sits in a long line of work on storage systems, fault-tolerant distributed protocols, and the integration of erasure codes with distributed systems. It occupies a unique niche in the existing literature, by providing strong consistency and fault tolerance guarantees while handling arbitrary transactions on erasure coded data with minimal impact to latency and throughput when compared to replication.

CHAPTER 4

DSLABS: TEACHING RIGOROUS DISTRIBUTED SYSTEMS WITH EFFICIENT MODEL CHECKING

Distributed systems are a fundamental element of the modern computing landscape. An increasing number of applications and system services are being designed for the cloud, often involving distribution across multiple data centers, with many services designed to operate at huge scale. Multi-tenant data centers alone are now a $60 billion per year business [65].

However, it is challenging to correctly implement a scalable and fault-tolerant distributed service. These systems must tolerate failing machines and adverse network conditions without violating correctness constraints, losing data, or compromising performance. Moreover, reasoning about distributed systems is notoriously difficult. The behavior of a system is the result of its input along with the network behavior: in a completely asynchronous setting, all possible patterns of message delays, drops, duplications, and reorderings must be considered. In the face of these challenges, even experts frequently make mistakes. For example, significant bugs have been found in published protocols implementing Paxos and Viewstamped Replication [51], as well as in the production code for Sprite [61], Chord [75], Raft [59], and BerkeleyDB [73].

Our motivation is to develop tools and a methodology to help novice distributed systems programmers (i.e., students) design and implement correct and performant distributed systems. Our requirements for such a methodology are that it: (i) can find common bugs in the systems students are asked to implement, (ii) uses tools which run in a timely fashion, (iii) finds errors reliably and repeatably, (iv) helps students understand and fix problems when they are found, (v) has students implement real systems which can run efficiently across multiple machines, and (vi) uses programming languages in wide use and tools which can be quickly and easily learned.

Testing is a standard approach to software validation. However, ad hoc testing is unlikely to uncover all errors that can occur in a distributed system, even for a relatively small system. Initially, we tried providing students a set of hand-written stress tests for each of our lab assignments. Student submissions often passed all of these tests, but some students still found further errors when using their solutions as a component in later labs. Although we added tests to catch specific issues as we learned of them, we found it difficult to keep up with the diversity of possible student errors. Students often (incorrectly) believe their code works once it passes a test suite, leaving them with a false sense of mastery.

Code review is another common approach. Among solutions that passed all of our tests, our course staff was often able to find additional bugs by inspection. Of course, code review is expensive, it requires a high level of training, it is not scalable, and in practice it provides feedback to students far too late to be useful.

At the opposite end of the spectrum, Verdi [69] and IronFleet [21] have demonstrated **formal verification** of distributed system implementations. Formal verification can eliminate entire classes of bugs, but the learning curve for these tools is steep. Correctness proofs often require multiple person-years of effort even for relatively simple implementations.

Faced with the challenges of building robust distributed systems and the inadequacies of other methodologies, both academic research and industry have increasingly turned to

model checking to validate system correctness. Model checking overcomes the weakness of ad hoc testing by systematically exploring *all possible executions*, but without the high labor cost of formal verification. Industry leaders such as Amazon and Microsoft report that they use explicit state model checking [23] to validate protocol specifications before they are implemented [32, 57].

One commonly used tool for specifying and model checking distributed systems is TLA+ [33]. While TLA+ has been used in industry and academic settings to great effect, it is difficult to master within a single term, and distributed systems specifications in TLA+ do not produce executable code. Instead, after model checking in TLA+, developers must re-implement their systems in an efficient runtime language like C or Java, leaving open a potential mismatch between the specification and the implementation.

Some research systems, such as MaceMC [28] and MoDist [73], model check implementations of distributed systems, and we initially thought we could use or adapt their techniques. A challenge for model checking concurrent systems is *state space explosion*: the number of possible executions is often exponential in the number of steps. Moreover, bugs in distributed systems are often complex, requiring many execution steps, making model checking prohibitively expensive. A common technique is to couple exhaustive search with periods of **random exploration** of the state space. Even so, search times can be large: the Mace model checker, which makes use of random exploration, can take over a day to execute [28]. While long-running validation is often appropriate as the last step before production code is released, we needed a solution that can check code and provide feedback to students in near real-time.

This chapter introduces DSLabs, a framework for writing, testing, model checking, running, and debugging distributed systems, along with a sequence of assignments written to use the framework. DSLabs defines a simple, single-threaded message-passing API in Java for students to use. Because each node in our programming model runs in a single-threaded event loop, the DSLabs model checker can systematically explore all possible

executions of student code (including message reorderings, drops, and duplications) at the coarsest granularity possible.

Our approach to model checking integrates a *gray-box* testing paradigm with *guided search* techniques. Gray-box testing allows us, as test developers, to write tests in terms of the problem specification and limited information about the implementation, while leaving most of the design decisions to students. Guided search allows us to leverage domain-specific knowledge to more efficiently model check student implementations. We specifically exercise student code in areas where we expect errors, rather than simply searching randomly or by brute force. We introduce two new techniques, *pruning* likely irrelevant states from the state graph and using a *punctuated search* approach where the model checker first finds states matching some intermediate constraint before restarting the search and continuing deeper into the state graph. Another key component of our approach is to teach students how to design for *model checking efficiency* and require a certain amount of efficiency from their implementations, allowing for deeper and more thorough checking.

Using these techniques, DSLabs makes model checking accessible to students by reliably and quickly finding many common bugs in student implementations of distributed systems, even bugs which are rare or unlikely to occur in practice.

When our model checker discovers a safety violation, it outputs a counterexample trace that yields the erroneous behavior. To simplify the task of understanding and reproducing failures, we developed a visual debugger and integrated it with the model checker; the visual debugger allows students to explore the consequences of alternate executions for a distributed system, much as a sequential step-through debugger enables a developer to reach a deeper understanding of program behavior.

We designed a set of assignments to teach distributed systems concepts and prepare students to build ambitious and correct distributed systems. Our assignments are based on, but go beyond, a similar set of labs developed by Robert Morris and colleagues at MIT [54]. Over the course of a ten-week quarter we ask students to build a transactional,

scalable (sharded), highly available, externally consistent (linearizable) key–value store with key migration and multi-key updates. The labs are specified at a high level; for example, students can choose their own consensus algorithm (e.g., Paxos or Raft), their own leader election algorithms, and their own message formats. The course staff solution is 2641 lines of code.

We have used the framework and labs to scale our undergraduate distributed systems class to 175 students per year; we have also used it to teach 50 professional masters students. This chapter reports on our experiences with guiding students through the DSLabs assignments using our framework. Almost all students were able to produce a working version of replicated key–value storage with dynamic sharding in a quarter; the stronger students also added multi-key transactions.

The rest of this chapter describes DSLabs and our experiences with it in more detail. Section 4.1 describes our programming model and the DSLabs API. Section 4.2 illustrates how our system would find a specific bug. Section 4.3 overviews the techniques we used to make tractable the task of model checking student code, and Section 4.4 discusses how to design distributed systems for efficient model checking. Section 4.5 describes our visual debugger and how it complements our model checker. Section 4.6 describes our experiences with having large numbers of students use our system to develop complex distributed systems code. Section 4.7 discusses DSLabs in relation to previous work, and Section 4.8 concludes.

4.1 The DSLabs Programming Model

In this section, we describe our distributed programming model and the details of the DSLabs API. A distributed system in the DSLabs framework consists of a group of nodes. Each node can access its own memory, communicate with other nodes by sending and receiving messages, and set timers to take some action after a certain amount of time has

```
public abstract class Node {
  /* event handlers, which students implement */
  public abstract void init();

  public abstract void handleMessage(
    Message message, Address sender);

  public abstract void onTimer(Timer timer);

  /* provided methods */
  protected final void send(Message message,
                            Address to) { }

  protected final void set(Timer timer,
                           int duration) { }

  protected final void set(Timer timer,
    int minDuration, int maxDuration) { }
}
```

Figure 4.1: *The DSLabs API. Students create subclasses of* `Node`*, each of which implements 3 handlers to define behavior upon initialization, receiving a message, or receiving a timer. A handler can modify internal state, send messages, and set timers. There is also a sub-node feature, which allows for composition and code reuse.*

elapsed. A programmer implements a distributed protocol by defining message and timer handlers — as well as defining a special handler for the initialization event.

Figure 4.1 shows the Java programming interface. Nodes run as single-threaded event loops; that is, they are I/O automata [46, 47] or distributed actors [22]. Event handlers must appear, from the standpoint of other nodes, to be *atomic actions* and should run to completion without waiting.

Clients vs. Servers Our assignments are based around a client–server model in which there can be an unbounded number of clients, while the core of the system is handled by the servers. These clients' behavior will differ based on the particular implementation of a protocol, however. Therefore, in DSLabs we model clients using *client nodes* (henceforth simply "clients"), relatively thin nodes which implement the client interface, allowing

external code (e.g., end-to-end tests) to send inputs (called commands) and receive outputs (called results) from the system. The client interface is asynchronous; it prescribes methods for sending commands, checking whether a result for the previously sent command exists, and retrieving that result. Students are responsible for implementing both the client and server node classes for each system.

Workers Important to the way the DSLabs testing framework works is our use of *worker* nodes, the implementation for which is provided by the framework itself. A worker is initialized with a client node as well as a workload (a list of commands). Workers run in a closed-loop; upon initialization, the worker sends the first command from the workload through the client. The worker passes all messages and timers it receives to the underlying client, and if one of those events causes the client to report having a result for the previously sent command, the worker sends the next command from the workload. The worker keeps track of the results returned by the client, which can be used by the testing infrastructure to check the correctness of the system. Worker nodes are just nodes; they do not themselves implement the client interface. Rather, they take clients written by students and transform them into nodes which send a pre-defined series of commands.

Network Model The weakest model typically assumed in the distributed computing literature is the asynchronous model in which messages can be delivered out of order, dropped, arbitrarily delayed, and even duplicated. Moreover, in an asynchronous system, there is no bound on the relative speeds of nodes; there is no guarantee that the durations of timers correspond to real time in a way that is meaningful across nodes. Timers are, however, delivered in an order consistent with the monotonicity of the local clock. Other, stronger network models (e.g., exactly-once or FIFO delivery) are compatible with our programming interface and implementable in DSLabs. For generality, however, we assume an asynchronous network for all of the lab assignments.

Failure Model The DSLabs framework allows for crash (i.e., non-Byzantine) failures, which are inherent to the asynchronous network model — a crashed node is equivalent to one whose messages are always dropped. We do not currently support nodes restarting after crashing, however. Support for restarts would require a model of stable storage and an interface to interact with it, which we eschew in order to focus on the core challenges posed by the distributed setting. This restriction is not fundamental, however, and a model of stable storage could be integrated with the DSLabs framework (and may be in the future).

Node Composition Finally, like other actor frameworks, DSLabs provides a way to compose nodes — by adding one node as a sub-node of another. From the standpoint of the rest of the system, these function together as a single logical node. This feature enables re-use of code and lets students build increasingly complicated systems over the course of several labs.

4.2 Testing and Model Checking in DSLabs

Now that we have defined the DSLabs programming model, we will briefly describe the DSLabs testing infrastructure and provide a motivating example for its use of model checking.

There are currently four assignments in DSLabs. The first asks students to implement an exactly-once RPC protocol on top of an asynchronous message-passing layer. The next has students implement a primary-backup system, and in the third lab, students implement the Paxos protocol. Finally, students layer on their Paxos implementation a reconfiguration and atomic commitment protocol (two-phase commit) across multiple groups of servers to create a scalable, transactional key–value storage system. The latter three labs are based on the labs developed by Robert Morris and colleagues at MIT [54]. Our labs go further by asking students to implement multi-key transactions in the fourth

lab, and there are significant differences in all three stemming from the differing programming models.

For each lab, we provide a suite of automated tests. The tests we provide generally fall into two categories: those based on running the system and those based on model checking. All tests will setup a particular configuration (e.g., how many servers and clients there are, the workloads to be used with each client, etc.) and then check that the output meets the assignment specification. Execution-based tests vary, for example, based on the behavior of the network (e.g., by configuring it to drop a percentage of all messages) and the failure pattern specified by the test. Model checking tests, on the other hand, vary based on the initial configuration of the system as well as the way the search is guided (described in Section 4.3.3.2).

As we discuss below, execution-based tests are good for exercising the "normal case" of a particular implementation, as well as testing that progress can still be made even under adverse conditions. Model checking, on the other hand, tests all cases systematically and is well suited to finding violations of safety properties in the asynchronous, message-passing setting.

4.2.1 Example

In the third assignment, students implement a fault-tolerant, linearizable replicated state machine using the Paxos protocol [35] to agree on the sequence of commands to be executed — that is, a shared log. The servers receive commands from clients and place them in consecutive slots in their logs, executing commands once they have been chosen (permanently fixed) for their respective slots. In order to meet the safety requirement (linearizability), no two servers should ever execute different commands in the same log slot. Briefly, each Paxos server has an associated (infinite) set of proposal numbers it can use to propose values (i.e., commands sent by clients); these proposal numbers are totally ordered and each is unique to the node which owns it. Before a node is allowed to use a proposal number, it must first contact a quorum of nodes (usually a simple majority)

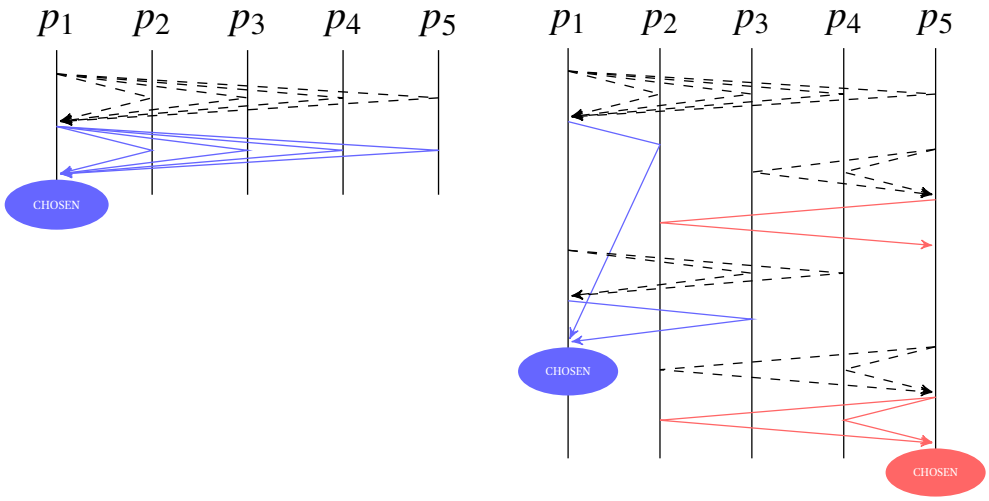

Figure 4.2: *Two executions of an incorrect version of the Paxos protocol in which second phase replies contain only values. The left trace shows p_1 successfully completing both phases of the protocol and choosing the blue value, as it should. The trace on the right actually demonstrates the error, showing p_1 and p_5 choosing for the first slot in the log both the blue and red values, respectively.*

to ensure: (1) that no proposal with a lower proposal number will ever be accepted by those nodes, and (2) that it discovers any proposals already accepted by those nodes. If the node discovers any already accepted proposals, it must use the value corresponding to the highest proposal number seen during phase one; otherwise it can use the client's value. Assuming it receives phase one responses from a quorum, the node can then send a proposal — a tuple with the value, index in the log, and proposal number. If that proposal is accepted by a quorum, then the value is chosen (permanently fixed) for that index in the log.

Now, consider the following bug in an implementation of Paxos. During the second phase of the algorithm when a node accepts a value being proposed, it only includes (and the proposer only checks) the accepted value in the response, rather than the proposal number. While this might seem like a benign modification to the Paxos protocol, it introduces a fatal error. If a node fails to get its proposed value accepted by a majority because of a competing proposer and proposes that same value with a higher proposal number,

it could later receive a delayed message from a node responding to its original proposal (with a smaller proposal number). This would mean the node could decide that a value has been permanently chosen (accepted by a majority with the same proposal number) when, in fact, some other value could have been chosen. Figure 4.2 shows a trace demonstrating the error. This bug is realistic; misunderstanding how to check for responses is a common error in student implementations of Paxos.

While this bug could cause a violation of linearizability, witnessing such a violation would be rare. For an error to occur, there would have to be a precise ordering of message deliveries, including a significantly delayed message. Even an execution-based test in which messages are randomly dropped, duplicated, and delayed would be unlikely to trigger this specific sequence of events. The error would either never be caught or only be caught in a tiny fraction of executions. In light of our goal of providing a thorough suite of tests, not being able to find a common bug like this one is problematic!

On the other hand, model checking can find this bug reliably. Using state graph exploration, along with optimizations described in Section 4.3, we designed a model checking test which can reliably find this particular error. The fact that it is an unlikely outcome, based on runtime characteristics of the system, is irrelevant to the model checker.

Note, however, that both execution-based tests and model checking tests have roles to play in the testing infrastructure. For example, live-lock is always possible in an implementation of Paxos [14]; whether it occurs in practice (under ideal network conditions) is largely a question of how well the first phase is implemented (e.g., tuning the durations of the various timers). Execution-based tests, unlike model checking, are well suited to testing these kinds of liveness properties. Moreover, as we will discuss in Section 4.3, model checking tests using breadth-first search of the state graph are primarily limited in the depth to which they can explore. Execution-based tests give us the ability to check safety properties on much longer runs of students' systems, albeit without the degree of thoroughness and repeatability provided by model checking.

4.3 Designing a Model Checker for Students

Model checking brings many advantages over execution-based testing. However, existing approaches to model checking distributed systems either come with significant learning curves or are not able to find common bugs in distributed systems in a timely fashion. One of the primary goals of DSLabs is to make the framework accessible and useful to novice distributed systems programmers, so that they can spend more of their time focusing on the subject material.

4.3.1 Simplifying Implementation

Like previous work on model checking distributed systems, the DSLabs model checker makes assumptions about the systems it checks (e.g., handler determinism [18]). In order to simplify the student's task of writing acceptable code, we mechanize the process as much as possible.

4.3.1.1 Collapsing Equal States

One of the goals of the DSLabs framework is to make it as frictionless as possible for students to begin writing code; for that reason, we chose to use Java, a mature language most computer science students are already familiar with. However, in order to collapse equivalent states and avoid wasting work during model checking, we need to be able to compare student-created data structures with each other for equality. Having students implement the `equals` and `hashCode` methods themselves is cumbersome and error prone. The Project Lombok [63] library solves this problem; it provides the `@EqualsAndHashCode` annotation for classes which generates those methods at compile time. We annotate all classes in the provided skeleton code and give students a simple rule: if you create a class, add `@EqualsAndHashCode`.

4.3.1.2 Testing Determinism

One requirement of the model checker is that the event handlers students write are *deterministic*. That is, the resulting state after the execution of a handler should only depend on the original state and the event being handled. Without determinism, the DSLabs model checker can miss invariant-violating states. Some sources of non-determinism are obvious (e.g., generating an integer through `new Random().nextInt()`), while others are more subtle (e.g., `HashMap` iteration order). In order to help students write deterministic code, we added an optional flag to our model checker. When enabled, this causes the model checker to clone each state and deliver each event twice, once to each clone. If the resulting states are not equal, there is some sequence of events with a non-deterministic outcome.

We test that students correctly apply `@EqualsAndHashCode` using the same flag. During model checking, we clone every state reached and check that the clone is equal to and has the same hash as the original. Similarly, we also test that message handlers are idempotent, a useful — but not necessary — property for a distributed system running in an asynchronous environment.

4.3.1.3 Supporting Randomness

While determinism is useful for model checking, access to randomness is often useful in distributed protocols [4,40,60]. One potential resolution to this tension is to allow a node to make pseudo-random choices by maintaining a seed as part of its state (bootstrapping the seed using its `Address`, which is guaranteed to be unique). While this approach is safe, it can lead to an unnecessary explosion in the size of the state graph. Another potential resolution is to allow programmers to expose all non-deterministic choices in event handlers to the model checker. However, this would significantly complicate the programming interface.

DSLabs avoids these drawbacks by only supporting the most common and practical use of nondeterminism in distributed systems, random timer durations, which previous

work has shown can yield substantial performance benefits [60]. The framework provides a method for specifying the minimum and maximum durations when setting a timer (see Figure 4.1). During execution-based tests, the framework chooses the actual duration of the timer from a uniform distribution. However, whenever the model checker is running, all locally consistent delivery orders are considered.

4.3.2 Defining Gray Boxes

When designing distributed systems assignments, there is a choice to be made about the amount of detail in the specification and provided skeleton code. At one extreme, we can specify a distributed system only in terms of its externally visible behavior. This can make it difficult to perform anything but brute-force model checking, however. At the other extreme, we can provide students with the full definitions of all message and timer types to be used (and even the data structures to be kept at each node). This would enable fine-grained, isolated testing of each handler but would obviate many of the challenges we would like students to solve. The DSLabs assignments take a middle *gray-box* approach of prescribing limited aspects of the system while leaving most design decisions up to students.

4.3.2.1 Commands and Results

One source of information about the states of distributed systems common to all labs is the workload given to and results returned by worker nodes. The automated tests use this information to check correctness — for all assignments, we require linearizability. For example, for a simple key–value store, we can construct a test where multiple workers are given a sequence of append-and-get operations to the same key. Linearizability of these append-and-get operations can be stated as follows. First, when all workers' result lists are combined and sorted by length, each value should be equal to the previous value concatenated with the value from the corresponding append-and-get operation. And

second, no result in this combined result list should have been returned by its client before the command for a previous result in the list had been sent.[1]

4.3.2.2 Intermediate Information

While the client interface is important for specifying and checking the end-to-end correctness of student implementations, we often want even more insight into system states, either for testing the correctness of individual components or to enable the optimizations discussed Section 4.3.3.2. This information takes two basic forms in DSLabs. In early labs, we define some of the message types used by students. This is primarily done for pedagogical reasons, as a gentle introduction to programming in the DSLabs framework. However, this information can then be used by the tests, and we can write predicates in terms of the messages present in the network. For example, in the primary-backup lab, the provided code completely specifies the messages sent to and received from the view server (the node that maintains configuration information); nodes can only become the primary or backup by receiving a message from this server. Using this information, we can define predicates on states describing whether or not the view server has started a given configuration.

In later assignments, we provide less skeleton code. In order to get more information about the system states, we specify limited informational interfaces to be implemented by student node classes. For instance, in the Paxos lab, we have the Paxos servers implement a method which will return the status (either tentatively accepted, chosen, empty, or garbage collected) of a given slot in their logs, as well as the command in that slot, should it exist.[2] This lets us write invariants in terms of this information. For example, there should never be two different commands chosen in the same slot. Another example invariant

[1] This second condition can usually be elided. For append-and-get workloads, a violation of linearizability but not serializability would require the system to predict a value to be appended before it is sent.

[2] This particular interface is a subset of the one defined in the MIT distributed systems labs [54]. Theirs defined additional methods and was functional, rather than informational, and thus constrained student design decisions.

is that if a node believes a command has been chosen in a slot, it must be present (or already garbage collected) at a majority of servers.

It is worth noting that because students have full information about their implementations, they can go beyond the gray-box approach we describe above. Using the DSLabs testing framework, they can write their own predicates in terms of any piece of their system's state and test their own assumptions about their systems either through model checking or execution-based tests.

4.3.3 Dealing with State Explosion

Systematic search of the state graph, while useful in uncovering some bugs, is not a panacea; model checking is up against a fundamentally hard problem. For most distributed systems, the state graph is infinite, and the number of unique states reachable in n steps is exponential in n. This poses a particular challenge for our use-case. We want the model checker to reliably find issues in student code, guiding them towards a correct implementation, and we want it to do so in a timely fashion. The naïve approach of breadth-first search from the initial state through certain number of states or for a certain amount of time might miss many of the bugs model checking was supposed to catch! Moreover, randomized approaches to search have the same problem as execution-based tests: they can fail to reliably find bugs which only occur in a small fraction of states.

The DSLabs tests use two basic strategies to find as many bugs as possible: searching for progress and using a guided search strategy.

4.3.3.1 Pairing Progress with Safety

One challenge facing the DSLabs model checker is that different implementations of the same basic protocol can have drastically different state graphs, impacting the performance of the model checker. We will discuss the ways we encourage students to design for model checking efficiency in Section 4.4, but we would like to avoid situations in which the model checker, presented with buggy and inefficient code, exits without reporting an error.

One way to gain some certainty that the model checker can get far enough into the state graph to find invariant-violating states, should they exist, is to also search for states in which progress has been made. For instance, in the Paxos lab, we first search for a state where a worker has received results for a small number of sequential commands. Then, we search for invariant-violating states, with similar limitations (e.g., time). The intuition as to why this pairing of progress and safety searches is helpful is that if a buggy system can take "good" actions in a certain number of steps, it is often (but not always) the case that it can take "bad" actions in a similar number of steps.

4.3.3.2 Guiding the Search

While pairing progress and safety searches can help rule out needlessly inefficient implementations, some bugs in distributed systems can require lengthy traces. For instance, the bug described in Section 4.2, when injected into our Paxos implementation, takes a minimum of 36 steps to trigger a violation of end-to-end linearizability. Using the slot-validity invariants defined in Section 4.3.2.2, we can reduce that to 23 steps. However, this is still far deeper than our model checker can search exhaustively within a reasonable time bound (see Table 4.1). In DSLabs, we address this challenge by using knowledge about each system's specification to guide the model checker's search towards interesting, error-prone areas of the state space.

First, we specify *prunes* — predicates telling the model checker which states not to expand. For example, worker nodes are given finite workloads in most model checking tests. When checking properties of their result lists (e.g., linearizability), we can safely prune away states in which all worker nodes have finished.

Second, we take an iterative approach to model checking we call *punctuated search*, in which we first search for a state satisfying some intermediate constraint and then restart the search from there. In the primary-backup lab, for example, little of interest happens until both a primary and backup have been initialized. Therefore, one of our model checking tests first searches for a state in which both have been initialized and all clients

have been informed of this, and then begins a new search from this point, allowing it to search much deeper into a more error-prone region of the state graph.

Furthermore, punctuated search also allows test developers to design model checking tests targeted at specific classes of bugs. For instance, one of our Paxos tests guides the model checker through a sequence of leader changes necessary to produce the bug from Section 4.2. Indeed, our tests can successfully find the bug in student implementations. Reliably finding such a bug would not be possible, at least with reasonable cost, without punctuated search.

In the most extreme case, we can even proceed one step at a time through a particular execution known to be problematic, while still leaving the flexibility to do a full breadth-first search from the end state if necessary.

4.3.4 Improving Understandability

Model checking can uncover bugs in student code, but if students cannot interpret the output of the model checker, then it is of limited utility.

4.3.4.1 Annotating State Predicates

First, if the model checker finds an invariant-violating state, students need to be able to understand the invariant. While they do have full access to the test code, some of our predicates are built out of reusable pieces, and we initially found that students had a hard time reading them. Therefore, we modified the DSLabs test infrastructure so that state predicates return a tuple with a boolean — whether or not the predicate is satisfied — along with an optional explanatory string. This allows predicates to return detailed information about exactly why they were or were not satisfied, and allows us to display more useful information to students.

4.3.4.2 Producing Understandable Traces

Although model checking using breadth-first search produces small traces, they are not necessarily intuitive traces. When humans write explanations of bugs in systems, they tend to pair events with their immediate consequences. However, in traces produced by the model checker, concurrent events will be randomly ordered. These traces are hard to follow, as they can have many "context switches."

Therefore, we implemented a post-processing phase for traces returned by the DSLabs model checker that reorders concurrent events into a more intuitive order. First, it takes the original trace and builds its *execution graph*, the graph of events (rather than states). The edges between events are defined by the happens-before relation [34]. It then performs depth-first, topological sort of this graph to produce an equivalent trace which pairs events with their immediate successors whenever possible.

4.4 Designing Systems for Model Checking

One of the goals of DSLabs is for students to create runnable distributed systems. We would like students to consider the performance characteristics of their systems, and our tests check that their designs attain reasonable run-time performance. We believe this makes our lab assignments compelling and allows students the sense of accomplishment in implementing realistic distributed systems.

It would be nice if students did not have to take model checking into consideration (aside from ensuring that their systems meet the basic requirements described in Section 4.3.1). If this were true, the model checking tests we provide would simply be better tests which reliably caught tricky distributed systems bugs. However, system design decisions can have a large impact on the performance of the model checker, and thus its ability to find bugs within a reasonable amount of time. A fundamental aspect of distributed systems is that nodes often need to reason about the state at other nodes [20]; depending

on how that state is represented and transmitted between nodes, it can easily explode the state space even further and reduce the effectiveness of the model checker.

This is particularly noticeable with certain performance optimizations. There is a natural tension between designing systems which perform well at run time and those that can be efficiently model-checked. Roughly speaking, model checking works better on simpler systems, while run-time optimizations often add complexity. Luckily, code that is readily model checkable usually corresponds to the kind of code we want students to write — code that is as simple as possible with respect to its state graph. For instance, we encourage students to write idempotent message handlers, which generally correspond to both simpler code and checkability. We also encourage students to avoid keeping or sending unnecessary state, as it can create extraneous states for the model checker to examine.[3] However, some common optimizations and techniques can cause poor model checking performance, limiting the design space artificially.

For example, some optimizations have the same information sent over the network in multiple different forms. In Paxos, when a node is attempting to help another node catch up, it is natural to batch decisions and send them in a single message rather than as a series of small messages. However, if those decisions also exist as individual messages in the network, then the batched message can create unnecessary states, and the situation is often worse since each prefix of a list of decisions could be sent in its own message. Generally, the model checker can tolerate some amount of batching; indeed, our reference implementation of Paxos batches decisions as described above. However, incautious application of these techniques can lead to problems.

The number of events required for a system to make progress can also impact model checking performance. Most Paxos implementations elect a leader, where the leader handles all client requests and other nodes only attempt to become leader if they suspect the leader has failed — i.e., if they haven't heard from the leader within a certain amount

[3] In the case of state kept purely for debugging purposes, students can choose to disregard fields when determining hashes and state equality using the `exclude` parameter on the `@EqualsAndHashCode` annotation. We do not recommend this, however, as it is error-prone.

of time. This means that another node would need to receive two timer events in a row before attempting to become leader.

Viewstamped Replication (VR) [42, 58] takes a different approach to leader election; instead of letting each node attempt to elect itself leader, leadership is assigned on a round-robin basis. This has the practical benefit of reducing contention for leadership, but in some cases it can be disastrous for model checking performance. In a deployment with five VR nodes, it can take up to twelve sequential events for a particular node to become leader, making complicated failure patterns unreachable.

In the end, we have found that the best heuristic to give to students to ensure model checkability of their systems is the following: Favor simplicity above all else. Do not keep or send unnecessary state. Explore performance optimizations, but not at the expense of significant added complexity. Finally, consider the number of events it takes for your system to make progress from any state; ensure that number is reasonably close to the minimum.

4.5 Visual Debugger

To help students better understand the behavior of their programs and to make model checking more accessible to students, we developed a graphical, interactive debugger for distributed systems. The visual debugger enables developers to discover and diagnose bugs in their distributed systems by controlling the order in which messages and timers are delivered. The visual debugger supports time-travel, allowing developers to explore multiple executions of the same system. We have included the visual debugger in our distributed systems labs, and integrated it with the testing framework. Students can easily start the debugger on their systems in order to explore their behavior. The visual debugger also starts automatically when the model checker finds an invariant violation; students can then step through the violating trace. Unlike other trace visualization tools, the visual

debugger also enables students to branch off of the violating trace and explore alternate executions in order to better understand the error.

The visual debugger's execution model is compatible with the execution model of the labs: the debugger supports event-based systems with handlers that run in response to messages and timers. Rather than executing these handlers as a result of predefined tests or as part of an exhaustive search through the system, the visual debugger executes them (via a shim layer that connects student handlers to the the visual debugger debugger over the network) in response to the user's commands in the debugger.

Figure 4.3 shows a screenshot of the the visual debugger interface. The debugger's interface uses "inboxes" to model the messages and timers waiting in a network. When a new message is sent, or a timer is set, it immediately goes into the receiving node's inbox. Any message or timer in an inbox can be delivered at any time; messages can also be dropped or duplicated. The user controls the delivery order of messages and timers by clicking on them. The user can also click on a node, message, or timer to inspect its contents, allowing them to investigate how the system's state evolves as it runs. The user can navigate through the history of previously-explored system states using the branching history view. They can easily explore multiple executions, investigating what happens if messages or timers are reordered.

Students completing the DSLabs assignments can use the visual debugger in multiple ways. The first is to simply run their system attached to the the visual debugger debugger and explore from the initial state. This enables hypothesis testing — does the system behave as expected in the normal case, or under various failure scenarios? The visual debugger makes it easy to discover simple bugs: if a node fails to respond to a message, or sends an unexpected message, it is immediately obvious. For instance, using the visual debugger, a student immediately discovered a bug in their Paxos implementation in which after receiving "prepare" replies from a quorum, a leader would send extra "accept" requests after each subsequent "prepare" reply. Since the bug impacted the system's

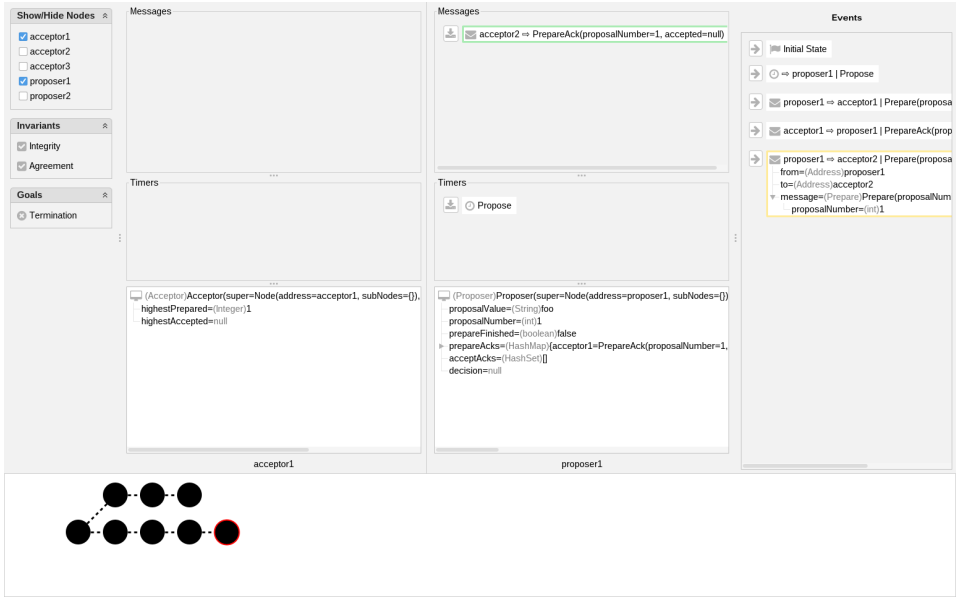

Figure 4.3: *The visual debugger window. Each node is displayed, along with an inbox of messages and timers waiting to be delivered at that node. The user can control delivery by clicking on the button next to timers and messages and can also inspect the contents of any message or timer or the state at any node. Using the branching history view in the bottom of the window, the user can navigate the states of the system they have explored. The user can reset the debugger to a previous state by clicking on it; this resets the system to that state so that the user can explore further from there. The user can also explore the list of events (message and timer deliveries) which led to the current state in the panel on the right side of the window. Finally, the left panel allows the user to control which nodes are currently visible in the main area, and depending on the current settings, lists of invariants (state predicates which should be true of all states) and goals (state predicates that a particular search test is using to look for progress) are also displayed.*

performance but not its safety, they would have been unlikely to discover it without the visual debugger.

The visual debugger is also used to visualize counterexamples to invariants found by the model-checker. When such a counterexample is discovered, the lab framework automatically starts the visual debugger on the system and passes it the execution trace (modified, as discussed in Section 4.3.4.2). The student can then step through the trace in

order to see what went wrong. Since the trace is running in the visual debugger, the student can explore other possible executions simultaneously, perhaps to determine whether a possible bug-fix is actually correct.

4.6 Experiences

In this section we summarize some of our initial experiences using DSLabs to teach distributed systems. We start by evaluating our own reference solution set and then describe some of the student experiences from our most recent offering of the course.

4.6.1 Code Complexity and Performance

To validate the DSLabs assignments and model checking framework, we implemented a solution set in Java. Table 4.1 lists the lines of code (including comments) in our reference implementation of each assignment. For comparison, we also list the lines of code in the framework API, the assignments' automated tests, the model checker itself, and the visual debugger.

First, the code needed for the assignments is tractable for students to complete in a single term. The restriction to deterministic event handlers had minimal effect on code complexity. A solution implemented in TLA+ would likely be smaller, but only modestly so, at the expense of students needing to learn a completely new language. Comparatively, the testing infrastructure is substantial; asking students to implement their own testing framework is likely a non-starter.

Unlike TLA+, our Java implementation can run in production mode. Table 4.1 also gives the maximum throughput attained by the solution set when run on a cluster of server class machines, each with two Intel Xeon E5-2680v3 CPUs (2.5GHz, 30M Cache, 24 hyperthreads) and 64GB DRAM. The primary-backup lab was, of course, run with an active primary and backup. For the Paxos lab, we used a 3 replica cluster. For the dynamic sharding and transactions labs, we used 7 groups of 3 replicas each, distributed across

Assignment	LOC	Kops/s	Test (s)	Search Depth	Guided Depth
Exactly once RPC	164	116	56	Exh.	N/A
Primary-backup	355	64	275	25	41
Multi-Paxos	647	46	293	13	26
Dynamic sharding	878	117	423	12	24
Transactions	597	4.3	355	12	24
Framework API	810				
Automated tests	4313				
Model checker	4322				
Visual debugger	2724				

Table 4.1: *For the solution set to each assignment in DSLabs, lines of code* (*including comments*)*, key–value operation throughput* (*operations per second*)*, total time for all tests in seconds, and maximum search depth* (*breadth first and guided*)*. Model checking for lab 1 was exhaustive. For comparison, we also list the lines of code for the DSLabs framework, the automated tests, the model checker, and the visual debugger.*

7 server machines. The transaction workload consisted of transactional reads to two randomly selected keys and transactional writes to two randomly selected keys, at a 9:1 ratio. There were 10 times as many keys as closed-loop workers, and the keys were selected from a uniform distribution, ensuring a low but non-negligible amount of contention. Our implementations, even though they are not highly optimized, are reasonably performant.

4.6.2 Rapid Feedback

To test the execution time of our automated tests, we ran it against our solution set on one of the aforementioned servers. Table 4.1 lists the wall clock time for testing the solution set for each assignment, along with the maximum search depth reached by the model checker. Note that student code is likely to be less efficient, taking longer to reach a given search depth. The execution time is moderate, in a range of three to seven minutes for the various assignments.

A key reason for our efficiency is guided search. We apply constraints on intermediate states of the execution to focus the model checker on particularly interesting execution

paths. In the primary-backup assignment, for example, we wanted to demonstrate that student implementations correctly handle multiple reconfigurations. With guided search, we were able to walk students' systems through a series of reconfigurations, checking that they can maintain safety throughout and still make progress from the resulting state. Guided search allows the model checker to check student code to a much deeper level than otherwise possible.

This supports our goal of giving students timely feedback on whether their solutions worked. Prior to adding model checking, it was common for students to find bugs in their Paxos implementation only when they tried to use that implementation in a later lab. By catching student errors more quickly, we reduce the amount of re-work needed. For example, one student wrote, referring to the old, pre-model checking version of the labs:

"Just 3 days before the deadline of the project, my partner and I discovered that our Paxos failed 1 of 100,000 tests. Though it's very unlikely that our program will crash when graded, we still decided to debug. However, we realized that the bug comes from our optimization of duplicate request detection before putting request on the Paxos operation log which means we need to rewrite fifty percent of the whole project but we did not give up. Finally, after 30 hours of work in 2 days, we fixed the design flaw and eliminated the bug. We were so excited that we started to dance in the lab."

While we do not have any direct evidence about the incidence of undetected bugs with and without model checking, after adding model checking we had zero reports where students found errors in their earlier assignments when completing later labs.

4.6.3 Thoroughness

A goal of our testing framework was to find likely student errors, even those that would be only rarely encountered in practice. Here, we evaluate the performance of the model checker on our solution set; the unguided and guided depth the model checker reached in each assignment is shown in Table 4.1. For these tests, all searches were time-limited to between 15 and 30 seconds.

For the primary-backup assignment, the protocol is simple enough that we were able to search relatively deeply, even without guided search. In the protocol, configuration state (the identities of the primary and backup) is centralized at a view server and distributed to the participants. Commands are not processed in a certain configuration until both the primary and backup have acknowledged the new configuration. Further, all messages are tagged with the configuration state of the sender; messages from old configurations are discarded. This reduces the set of states to be explored. For example, delivery of old messages to up-to-date participants has no effect, allowing the model-checker to quickly move on to other events. By contrast, in Paxos any node can trigger an election, and the state used for leader election is distributed across all nodes. Lagging nodes can continue communicating with each other long after other nodes have moved on. A consequence of having more options at every step is that the unguided search depth is shallower.

Through our use of gray-box testing (Section 4.3.2) and guided search (Section 4.3.3.2), however, we were able to substantially improve the depth to which we were able to search.

4.6.4 Comparison to Unguided Methods

In order to evaluate the effectiveness of our model checking techniques as compared to black-box, unguided methods, we take the bug described in Section 4.2 as a case-study. Specifically, we compare against a pure breadth-first search as well as random exploration. The previously described Paxos bug is typical of many errors seen in student implementations, and an invariant-violating trace is complicated — requiring a minimum of 36 steps and 4 leader elections — but not abnormally so.

In the DSLabs testing framework, we implemented a simple random exploration strategy which continuously takes random walks starting from the initial state and running for a pre-determined number of steps (in this case, 1000). Random exploration was able to uncover the bug injected into our implementation of Paxos. However, over 5 runs, it took an average of 12 hours to do so.

Pure breadth-first search fared even worse and was not able to uncover the bug. Exhaustively searching the state space up to the required depth of 36 would take an effectively infinite amount of time and space.

On the other hand, the guided search test we designed to find this type of bug was able to find the invariant violation in our implementation in 18 seconds. By designing model checking tests for specific classes of bugs, we are able to find errors in certain portions of the state space efficiently and reliably, something not possible without guided search. While guided and unguided methods are not mutually exclusive — both can be used sequentially or in parallel to check the same system — given our goals of promptness and efficiency, guided search techniques are invaluable and provide the right set of trade-offs.

4.6.5 Debuggability

One of the benefits of our model checker is that it can produce detailed information about invariant violations it finds. We found that this information did help students fix many issues found in their systems.

Out of approximately 500 separate submissions across all four labs, only 25 were found by the model checker to violate invariants. Compared to the number of submissions which failed tests due to liveness or performance issues, this is relatively few. As one point of reference, as a part of a larger user study Oddity [71], the previous iteration of the visual debugger, was outfitted with opt-in telemetry and reported over 150 separate debugging sessions started by students due to invariant violations found by the model checker during development. The real number of bugs uncovered by the model checker during student development is likely to be much higher; the above count does not include students who did not opt-in to our data gathering nor does it include invariant-violating traces which were not inspected using the visual debugger. Furthermore, almost all of these bugs were fixed before submission; only a couple of the reported traces were still present at grading time. Taken together, this data suggests that the bugs found by the model checker during development were readily fixed by students before submission.

4.6.6 Checkability

Model checking adds a constraint for students' implementations: design for model checking performance. Student design decisions can affect the depth to which the model checker can search in a given amount of time, as well as the depth at which errors occur, should they exist. A simple example of this was where a student incremented the proposal number in Paxos when retrying after a lost message. This meant that packet loss was not idempotent, expanding the search space. Another student had each node periodically send its entire state to every other node, as a way of keeping nodes up to date; this drastically expanded the search space.

We gave students advice on how to reduce search complexity and design for checkability (Section 4.4), encouraging students to avoid unnecessary events that would foil the search for safety errors. We also paired searches checking safety with searches for progress, ensuring that the model checker can find within some time bound states in which progress is made (Section 4.3.3.1).

In the end, we were fairly successful in encouraging checkability. Out of the approximately 500 submissions across all four labs, only 38 were unable to pass all of our searches for progress, showing that the vast majority of submissions attained reasonable model checking performance.

4.6.7 Thinking Distributed

An overarching goal is to encourage students to think about their code as inherently distributed. It is not enough to find one code path or event sequence that performs as expected; students need to consider all possible event sequences simultaneously, ideally by thinking in terms of the invariants maintained be their systems. For many students, this is the hardest part of the class.

Model checking helped tremendously with this, by finding bugs that students did not realize were latent in their code, and ones that less rigorous testing would have missed. Students often march through test cases incrementally, fixing problems only once they

occur. A particular student tried this for the primary-backup assignment and got stuck: the fix for a problem found by one test would often break the solution for previous tests. The student found that he could find a version to pass each of the tests, just not the same version. After we encouraged him to start over with a clean design that met all of the criteria simultaneously, he was able to quickly converge on a solution.

The visual debugger also helped: one student reported finding an unintended performance bug by running the code through the debugger and seeing an unexpected flood of messages after certain events.

4.7 Related Work

DSLabs and its model checking framework are preceded by a long line of research on model checking for distributed and concurrent systems.

Explicit-state model checking is, at its core, exhaustive exploration of a state space to a bounded depth. Much previous work has focused on reducing the time and space requirements of this exploration so that bugs, if they exist, can be found in hours or days rather than years. By contrast, since the DSLabs model checker needs to run frequently on student code, its time budget is only a few minutes. To explore a meaningful part of the state space in such a short time, DSLabs exploits the test developer's knowledge of the system's specification to guide the search (see Section 4.3.3.2). As such, most previous techniques are of limited utility in DSLabs. Most notably, partial-order reduction (POR) [17], dynamic partial-order reduction [15], and dynamic interface reduction [19] exploit the commutativity inherent in message-passing systems to reduce the number of redundant traces and the number of states that need to be explored. These techniques could potentially be applied in DSLabs to further improve performance, but would not be a substitute for guided search.

Both CMC [56] and VeriSoft [18] are early model checkers which ran on unmodified implementations of event-driven and concurrent systems. VeriSoft relies on POR

techniques, while CMC stores compressed records of explored states in order to avoid exploring redundant states. The DSLabs model checker also stores the states it explores; since it is always CPU- rather than memory-bound, compression is unnecessary.

SPIN [23] was another early model checker and remains in popular usage. Java PathFinder [68] is another popular model checking tool designed for concurrent Java programs. Both model checkers implement numerous optimizations and support various modes and search strategies. However, as mentioned above, these optimizations are not a replacement for the guided search techniques used in DSLabs. Moreover, because DSLabs uses a stateful model checker specialized for its distributed programming model, it obviates the need for many of the features of general-purpose model checkers designed for concurrent programs.

MoDist [73] checks unmodified distributed systems by interposing on system calls made to the operating system. It can then explore reachable states by controlling the scheduler. Because MoDist is aimed at unmodified code, a large amount of its complexity is in handling the inherent non-determinism of the OS interface, e.g., due to thread scheduling, randomness, and time-based system calls. By contrast, our approach can be simpler because we assume students use the DSLabs programming interface and write deterministic event handlers.

Mace [29] is an actor-based extension to C++ for building distributed systems. Like the DSLabs programming interface, one of Mace's strengths is that it is conducive to model checking based on high-level transitions of systems. In particular, MaceMC [28] is a model checker used with Mace, specifically designed to check liveness, rather than safety, properties in eventually consistent systems. To that end, MaceMC uses a multifaceted approach to model checking. MaceMC begins by searching exhaustively from an initial state up to some depth bound; it then takes long random walks from these states, searching for live states. While it is possible the techniques used in MaceMC could be beneficially integrated into the DSLabs model checker, MaceMC considers distributed systems and predicates on them to be black boxes, and therefore does not consider the guided search

optimizations used in DSLabs model checking tests. MaceMC also includes a debugger, MDB. Like the visual debugger included in DSLabs, MDB allows developers to step through counterexample traces and explore alternative executions; unlike our debugger, it does not include a graphical interface.

SAMC [39] reduces the number of states and transitions that need to be considered by having the distributed systems developer classify messages based on their semantics. This information, however, is highly implementation-specific, and the soundness of the model checker relies its correctness. Rather than expecting students to specify correct semantic information about their implementations, DSLabs uses information about the problem specification, rather than the implementation, to explore interesting subareas of the state space.

CrystalBall [72] steers deployed systems away from potential invariant violations using a model checker started from each global system state. In effect, this results in an exploration of the state space branching out from a single, realized execution path. This is similar, in a way, to the guided search in DSLabs; rather than being guided by the specification, however, it is guided by a single system execution.

The WiDS Checker [45] and Friday [16] allow developers to debug their systems by recording traces in production and replaying them. Both provide facilities for developers to inspect these traces in detail, observing how the state of the system evolves over time. Similar facilities could be integrated into DSLabs in order to help students understand bugs identified by the model checker.

DEMi [64] provides a way to minimize the very long invariant-violating traces found via random fuzz testing. DEMi explores similar traces using a variant of dynamic partial-order reduction [15]. Integrating DEMi with DSLabs by having it analyze and minimize (potentially long) traces produced by execution-based tests could be useful.

4.8 Conclusion

Implementing distributed systems is notoriously difficult. The DSLabs framework and assignments give teachers and students tools to face these difficulties. By creating a simple programming interface in a well-known language, we enable students to begin creating real systems on day one. Through model checking, we built a testing infrastructure capable of meeting the challenges of the asynchronous setting and providing quick and useful feedback to students. We then built a suite of optimized execution-based and model checking tests for each assignment, making use of guided search and other optimizations to overcome the perennial limitations of model checking. Finally, we integrated DSLabs with a visualization tool to help students better understand and debug their systems.

Using the DSLabs framework and assignments, we have successfully guided hundreds of students through the process of building a fault-tolerant, scalable, distributed key–value store. Furthermore, these student-built systems are actually runnable, rather than mere specifications; they can be deployed in a fully distributed fashion and can achieve considerable performance. While implementing systems for both execution and model checking can be idiosyncratic and requires certain compromises, the DSLabs framework streamlines this process, and we have found that an overwhelming majority of students succeed in the class and enjoy the experience. By providing this programming framework and making efficient model checking accessible to students, we believe DSLabs provides students with the tools to become proficient distributed systems programmers.

While the techniques we identified for improving model checking efficiency are particularly suited to the instructional setting, we believe they may also be of interest to more experienced developers. Gray-box testing and guided search allow developers to more effectively and rapidly model check their distributed systems, without tightly coupling tests to an implementation, potentially reducing the time spent identifying errors and increasing developer productivity.

The DSLabs framework and all assignments are open-source [50].

CHAPTER 5

CONCLUSION

Distributed systems are omnipresent, and building reliable, performant, and resource-efficient distributed systems is an important but difficult task. New protocols, programming environments, and tools are needed to ensure that the pieces of software running these critical systems are correct and make the best use of computing resources possible.

To that end, this thesis described Amalgam, a new protocol for erasure-coded, transactional block storage. Amalgam can handle read-modify-write operations executed on each of its shards of data, and it can commit operations with latency and throughput similar to a replicated baseline. This design reduces the storage overhead of fault-tolerant storage while retaining the ability to commit arbitrary transactions. It can tolerate the failure of a configurable number of machines, and a separate configuration service replaces servers as they fail. The Amalgam recovery protocol ensures that data remains consistent across machine failures and recoveries, even as servers restart. On read-modify-write workloads, Amalgam significantly outperforms a class of distributed systems utilizing erasure codes which split data objects.

Amalgam itself is not a database. However, it is a core coordination protocol which other systems can be built atop. Indeed, existing object storage mechanisms [74] which are compatible with Amalgam's fixed-size data model can be readily used with Amalgam.

This thesis also described the DSLabs framework. The framework is designed to introduce programmers to the distributed programming environment and prepare them to write correct distributed systems. It utilizes model checking to provide as much assurance as possible that the systems these programmers build satisfy the specified safety properties and do not violate key invariants. The DSLabs framework also allows more experienced distributed systems programmers to guide the search process of the model checker towards particularly dangerous portions of the state space, mitigating the state explosion problem. When safety violations are found, DSLabs returns a concrete trace which demonstrates the error and provides a powerful visualization tool which allows programmers to explore the problematic trace interactively.

DSLabs is designed for university students taking a first course in distributed systems, giving them the ability to write code which is both runnable and checkable. DSLabs has been used to teach thousands of students at many universities, and work on the project continues to this day.

BIBLIOGRAPHY

[1] Marcos K. Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks* (*DSN '05*), pages 336–345, June 2005.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, September 1987.

[3] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, January 1993.

[4] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing* (*PODC '83*), Montreal, Canada, 1983.

[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery In Database Systems*. Addison-Wesley, Boston, MA, USA, February 1987.

[6] Walter A. Burkhard and Jai Menon. Disk array storage system reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing* (*FTCS '93*), pages 432–441, Toulouse, France, June 1993.

[7] Erasure encoding as a storage backend – Ceph. `https://tracker.ceph.com/projects/ceph/wiki/Erasure_encoding_as_a_storage_backend`.

[8] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[9] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[10] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[11] Fay Chang, Minwen Ji, Shun-Tak Leung, John MacCormick, Sharon Perl, and Li Zhang. Myriad: Cost-effective disaster tolerance. In *Proceedings of the Conference on File and Storage Technologies* (*FAST '02*), Monterey, CA, January 2002. USENIX.

[12] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Transactions on Storage*, 13(3):1–30, September 2017.

[13] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *Proceedings of the 2017 USENIX Annual Technical Conference* (*USENIX ATC '17*), pages 539–551, Santa Clara, CA, July 2017.

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[15] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '05*), Long Beach, CA, 2005.

[16] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (*NSDI '07*), Cambridge, MA, 2007.

[17] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg, 1996.

[18] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '97*), Paris, France, 1997.

[19] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (*SOSP '11*), Cascais, Portugal, 2011.

[20] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[21] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (*SOSP '15*), Monterey, CA, 2015.

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (*IJCAI' 73*), Stanford, CA, 1973.

[23] Gerard Holzman. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[24] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies* (*FAST '21*), pages 233–248, February 2021.

[25] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX Annual Technical Conference* (*USENIX ATC '12*), Boston, MA, June 2012.

[26] Kai Hwang, Hai Jin, and Roy S.C. Ho. RAID-x: A new distributed disk array for I/O-centric cluster computing. In *Proceedings of the 9th IEEE International Symposium on High-Performance Distributed Computing* (*HPDC '00*), pages 279–286, August 2000.

[27] Kai Hwang, Hai Jin, and Roy S.C. Ho. Orthogonal striping and mirroring in distributed RAID for I/O-centric cluster computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):26–44, January 2002.

[28] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (*NSDI '07*), Cambridge, MA, 2007.

[29] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '07*), San Diego, CA, 2007.

[30] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[31] Hairong Kuang. Saving capacity with HDFS RAID. Facebook Engineering. `https://engineering.fb.com/core-data/saving-capacity-with-hdfs-raid/`, June 2014.

[32] Leslie Lamport. Foundations of Azure Cosmos DB. https://www.youtube.com/watch?v=L_PPKyAsR3w.

[33] Leslie Lamport. The TLA home page. https://lamport.azurewebsites.net/tla/tla.html.

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

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

[36] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

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

[38] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS '96*), pages 84–92, Cambridge, MA, September 1996. ACM.

[39] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (*OSDI '14*), Broomfield, CO, 2014.

[40] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '81*), Williamsburg, VA, 1981.

[41] Jialin Li, Ellis Michael, Adriana Szekeres, Naveen Kr. Sharma, and Dan R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '16*), Savannah, GA, USA, November 2016.

[42] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, July 2012.

[43] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (*SOSP '13*), page 226–238, Pacific Grove, CA, 1991.

[44] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. LH*: Linear hashing for distributed files. *ACM SIGMOD Record*, 22(2):327–336, June 1993.

[45] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS checker: Combating bugs in distributed systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (*NSDI '07*), Cambridge, MA, 2007.

[46] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (*PODC '87*), Vancouver, Canada, 1987.

[47] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[48] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '08*), San Diego, CA, USA, December 2008.

[49] David Mazières. Paxos made practical. `https://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, 2007.

[50] Ellis Michael. DSLabs: Distributed systems labs and framework. `https://github.com/emichael/dslabs`.

[51] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage. In *Proceedings of the 31st International Symposium on Distributed Computing* (*DISC '17*), Vienna, Austria, 2017.

[52] Ellis Michael, Doug Woos, Thomas Anderson, Michael D. Ernst, and Zachary Tatlock. Teaching rigorous distributed systems with efficient model checking. In *Proceedings of the 14th European Conference on Computer Systems* (*EuroSys '19*), March 2019.

[53] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (*SOSP '13*), pages 358–372, Farminton, PA, November 2013.

[54] Robert Morris. 6.824: Distributed systems. `http://nil.csail.mit.edu/6.824/2015/`, 2015.

[55] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (*HPDC '14*), pages 61–72, Vancouver, Canada, 2014. ACM.

[56] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '02*), Boston, MA, 2002.

[57] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, March 2015.

[58] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing* (*PODC '88*), Toronto, Canada, August 1988.

[59] Diego Ongaro. Bug in single-server membership changes. `https://groups.google.com/forum/#!topic/raft-dev/t4xj6dJTP6E`, July 2015.

[60] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference* (*USENIX ATC '14*), Philadelphia, PA, 2014.

[61] John Ousterhout. The role of distributed state. In *CMU Computer Science: A 25th Anniversary Commemorative*, 1991.

[62] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (*SIGMOD '88*), pages 109–116, Chicago, IL, 1988.

[63] Project Lombok. `https://projectlombok.org`.

[64] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI '16*), Santa Clara, CA, 2016.

[65] Synergy Research Group. Cloud growth rate increased again in Q1. `https://www.srgresearch.com/articles/cloud-growth-rate-increased-again-q1-amazon-maintains-market-share-dominance`.

[66] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. *ACM SIGOPS Operating Systems Review*, 31(5):224–237, October 1997.

[67] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys*, 47(3), February 2015.

[68] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering* (*ASE '00*), Grenoble, France, 2000.

[69] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '15*), Portland, OR, 2015.

[70] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[71] Doug Woos, Thomas Anderson, Michael Ernst, and Zach Tatlock. A graphical interactive debugger for distributed systems. arXiv Preprint: `https://arxiv.org/abs/1806.05300`.

[72] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Transactions on Computer Systems*, 28(1):1–49, March 2010.

[73] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI '09*), Boston, MA, 2009.

[74] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. Erasure coding for small objects in in-memory KV storage. In *Proceedings of the 10th ACM International Systems and Storage Conference* (*SYSTOR '17*), pages 1–12, Haifa, Israel, 2017.

[75] Pamela Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, March 2012.