Designing Storage and Privacy-Preserving Systems for Large-Scale Cloud Applications

by

Adriana Szekeres

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

Doctor of Philosophy

University of Washington

2020

Reading committe:
Henry M. Levy, Co-Chair
Arvind Krishnamurthy, Co-Chair
Dan R. K. Ports

Program Authorized to Offer Degree:
Computer Science & Engineering

©Copyright 2020

Adriana Szekeres

University of Washington

Abstract

Designing Storage and Privacy-Preserving Systems for Large-Scale Cloud Applications

Adriana Szekeres

Chair of Supervisory Committee:

Henry M. Levy, Arvind Krishnamurthy
Computer Science & Engineering

Today's mobile devices sense, collect, and store huge amounts of personal information, which users share with family and friends through a wide range of cloud applications. Many of these applications are *large-scale* – they must support millions of users from all over the world. These applications must manage a massive amount of user-generated content, they must ensure that this content is always available, and they must protect users' privacy.

To deal with the complexity of managing a large amount of data and ensure its availability, large-scale cloud applications rely on *transactional, fault-tolerant, distributed storage systems*. Latest hardware advancements and new design trends, such as *in-memory* storage, *kernel-bypass* message processing, and *geo-distribution*, have significantly improved the transaction processing times. However, they have also exposed overheads in the protocols used to implement these systems, which hinder their performance; specifically, existing protocols require unnecessary cross-datacenter and cross-processor coordination. This thesis introduces *Meerkat*, a new replicated storage system that *avoids all cross-core and cross-replica coordination for transactions that do not conflict*. Moreover, replica recovery protocols that do not require persistent storage devices, are proposed; these protocols can be easily integrated with existing storage systems.

Almost all applications offer their users a choice of privacy policies; unfortunately, they frequently violate these policies due to bugs or other reasons. Therefore, this thesis introduces SAFE, a *privacy-preserving system* that *automatically enforces users' privacy policies on untrusted mobile-cloud applications*. Our results demonstrate that SAFE is a practical way for users to create a chain of trust from their mobile devices to the cloud.

Acknowledgements

I would have not been able to write this thesis without the invaluable help I received along the way from many people who guided me, provided lots of feedback and insights, and have been my role models in the pursuit of becoming a good researcher.

First of all, I am extremely fortunate that I got to spend most of my PhD time having discussions and brainstorming sessions with **Irene Zhang**, **Jialin Li**, **Naveen Sharma**, and **Dan Ports**. We all started our programs at UW at the same time and, soon after, they became my collaborators, teachers, mentors, advisors, and close friends. Thank you for your invaluable help with my research, for your numerous insights that brought clarity to my often chaotic thoughts, for sharing your knowledge and wisdom, for your support, and for all the fun activities we shared – all of these kept me motivated and excited about research.

I am extremely grateful to my two PhD advisors, **Hank Levy** and **Arvind Krishnamurthy**, who gave me guidance and enough space to discover the topics in systems research I am most passionate about. Thank you for your constant encouragement, patience, feedback, and contagious optimism – this helped both to improve my work and balance my PhD life.

A huge thank you to my amazing collaborators from whom I learned a lot and whithout whom this thesis would have not been possible: **Irene Zhang**, **Jialin Li**, **Naveen Sharma**, **Dan Ports**, **Ellis Michael**, **Michael Whittaker**, **Arvind Krishnamurthy**, **Hank Levy**, **Katelin Bailey**, **Isaac Ackerman**, **Haichen Shen**, and **Franzi Roesner**. I am especially grateful to **Irene** and **Dan**, who significantly helped with navigating through the hard research problems and with the writing for the papers from which this thesis derives, to **Naveen** and **Jialin**, who helped me debug countless performance issues, to **Ellis**, who significantly contributed to our work on diskless recovery, and to **Michael**, who significantly contributed to our Meerkat work.

A big thank you to many other faculty and students at UW who enriched my PhD experience: **Antoine Kaufmann**, **Pedro Fonseca**, **Danyang Zhuo**, **Yuchen Jin**, **Niel Lebeck**, **Helgi Sigurbjarnarson**, **Kaiyuan Zhang**, **Tom Anderson**, **Pete Hornyack**, **Yoshi Kohno**, **Eunsol Choi**, **Camille Cobb**, **Victoria Lin**, **Brandon Holt**, **Lilian de Greef**, **Deepali Aneja**, **Simon Peter**, **Anna Kornfeld Simpson**, **Helga Gudmundsdottir**, **Liang Luo**, **Pratyush**

Patel, **Lequn Chen**, **Elizabeth Clark**, **Karl Koscher**, **Amrita Mazumdar**, **Seungyeop Han**, **Qiao Zhang**, **Ed Lazowska**, **Jacob Nelson**, **Angli Liu** , **Raymond Cheng**, **Henry Schuh**, **Xi Wang**, **Emina Torlak**, **Arun Byravan**, **Jialin Li**, **Ashlie Martinez**, **Samantha Miller**, **Tianyi Cui**, **Ming Liu**, and **Kevin Zhao**. I am especially grateful to **Xi**, always a friendly presence, providing useful feedback and advice, and to **Antoine**, **Arun**, **Kaiyuan**, **Yuchen**, **Danyang**, **Niel**, **Pedro**, **Anna**, **Liang**, for the often long and cheerful discussions.

I am extremely grateful to **Erik van der Kouwe**, **Andy Tanenbaum**, and **Nicolae Țăpuș** for sharing their knowlege and encouraging me to take the PhD path.

I am really grateful to all UW staff that took care of the administrative issues, so that I could focus solely on my PhD, especially **Lindsay Michimoto**, **Melody Kadenko**, **Dali Grubisa**, **Elise deGoede Dorough**, and **Lisa Merlin**.

I would like to thank the committee members **Hank Levy**, **Arvind Krishnamurthy**, **Radha Poovendran**, **Dan Ports**, **Xi Wang**, whose feedback substantially improved the clarity of this thesis.

Finally, I would like to thank my family for their endless support and encouragement.

To my mother, Antonica, my father, Adrian, and my sister, Anca.

Contents

Acknowledgements v

1 Introduction 1

1.1 Storage systems for large-scale cloud applications . . . 2

1.1.1 Desired design . . . 3

1.1.2 New hardware advancements and design trends . . . 4

1.1.3 Challenges . . . 6

1.2 Privacy-preserving systems for untrusted large-scale cloud applications . . 8

1.2.1 Challenges . . . 8

1.3 Contributions . . . 9

1.3.1 Fast and multicore scalable replicated transactions: Meerkat . . . 10

1.3.2 Fast recovery for in-memory replicated storage: Diskless Recovery . 10

1.3.3 Confidentiality system for untrusted cloud applications: SAFE . . . 11

1.4 Outline . . . 11

2 Background 13

2.1 Distributed storage systems . . . 13

2.1.1 Concurrency control . . . 14

2.1.2 Atomic commitment . . . 15

2.1.3 Replication . . . 16

2.1.4 Multicore support . . . 18

2.1.5 Over-coordination in existing solutions . . . 18

2.2 Privacy-preserving systems . . . 20

2.2.1 Information flow control . . . 20

2.3 Concluding remarks . . . 21

3 Meerkat 23

3.1 The Zero Coordination Principle . . . 25

3.2 Meerkat Approach . . . 27

3.3 Meerkat Overview . . . 28

3.3.1 System Model and Architecture . . . 28

3.3.2 Meerkat Data Structures . . . 29
3.4 Meerkat Transaction Protocol . . . 30
3.4.1 Protocol Overview . . . 30
3.4.2 Meerkat Transaction Processing . . . 31
3.4.3 Meerkat Failure Handling and Record Truncation . . . 36
3.4.4 Correctness . . . 51
3.5 Evaluation . . . 52
3.5.1 Prototype Implementation . . . 52
3.5.2 Experimental Setup . . . 54
3.5.3 Impact of ZCP on YCSB-T Benchmark . . . 56
3.5.4 Impact of ZCP on Retwis Benchmark . . . 57
3.5.5 ZCP and Contention . . . 58
3.6 Related Work . . . 59
3.7 Summary . . . 61

4 Diskless Recovery 63
4.1 Background and Related Work . . . 65
4.2 System Model . . . 67
4.3 Achieving Crash-Consistent Quorums . . . 69
4.3.1 Unstable Quorums: Intuition . . . 70
4.3.2 Crash-Consistent Quorums . . . 71
4.3.3 Communication Primitives in DCR . . . 71
4.3.4 Correctness . . . 74
4.4 Recoverable Shared Objects in DCR . . . 78
4.4.1 Multi-writer, Multi-reader Atomic Register . . . 78
4.4.2 Virtual Stable Storage . . . 79
4.5 Recoverable Replicated State Machines in DCR . . . 80
4.5.1 Viewstamped Replication . . . 81
4.5.2 Paxos Made Live . . . 84
4.5.3 JPaxos . . . 86
4.6 Evaluation . . . 88
4.7 Summary . . . 90

5 A practical privacy-preserving system 91
5.1 The SAFE Framework . . . 93
5.1.1 SAFE System Model and Concepts . . . 93
5.1.2 SAFE Policies . . . 96

5.1.3 Trust and Threat Model . . . 97
5.2 Agate, a SAFE Mobile OS . . . 98
5.2.1 Agate Architecture . . . 98
5.2.2 Agate Policies . . . 99
5.2.3 Agate Syscall Interface . . . 99
5.2.4 Agate User Interface . . . 100
5.2.5 Agate SAFE Enforcement . . . 100
5.3 Magma, a SAFE Runtime System . . . 101
5.3.1 Magma Architecture . . . 101
5.3.2 Magma IFC Model . . . 102
5.3.3 Magma Flow Tracking . . . 103
5.3.4 Magma SAFE Enforcement . . . 104
5.4 Geode, a SAFE Storage Proxy . . . 105
5.4.1 Geode Interface and Guarantees . . . 105
5.4.2 Geode Architecture . . . 105
5.4.3 Geode SAFE Enforcement . . . 106
5.5 Evaluation . . . 107
5.5.1 Implementation . . . 107
5.5.2 Security Analysis . . . 108
5.5.3 User Experience . . . 110
5.5.4 Programmability and Porting Experience . . . 111
5.5.5 Performance . . . 112
5.5.6 Application Performance . . . 113
5.6 Related Work . . . 115
5.7 Summary . . . 116

6 Conclusions and Future Research Directions 119

References . . . 123

1

INTRODUCTION

OVER the last decades, *cloud computing* has emerged to allow businesses to easily develop and deploy *cloud applications* to make their services available to users all over the world. Today's cloud giants – Google, Microsoft, Amazon, Facebook, VMWare – provide the necessary hardware and software resources to manage a *cloud*, which normally spans multiple *datacenters*, and expose it to businesses under the form of *services*, such as Software-as-a-Service (SaaS), Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS).

Many cloud applications, such as social networking, online games, email, and online shopping, are *large-scale* – they have millions of users which frequently access them from all over the world, since the ubiquitous mobile devices facilitate the access to such applications, and are entrusted with a massive amount of user-generated content. Therefore, three of the most important challenges for developing large-scale cloud applications are: 1) storing and managing the large amount of application state, including the user-generated content, 2) ensuring that this content is always accessible, and 3) preserving the privacy of users' data, since part of the application state is usually data owned by users, which is either private or shared with other users at the owner's discretion.

Fortunately, storage systems have been developed for a long time precisely to mask the complexity of storing and managing a large amount of data and ensuring that this data is always accessible. Over the last decades, many storage system designs have been proposed, fuelled by varied application requirements and workload characteristics, and

by varied hardware capabilities, all changing dramatically over the years. As we argue in Section 1.1 below, the requirements of many large-scale cloud applications today together with the latest hardware and network infrastructure advancements and trends are pushing for a storage system with several specific design characteristics and properties – an *in-memory*, *geo-distributed*, *fault-tolerant*, *multi-threaded* storage system that provides *strong correctness guarantees*, specifically *serializable* distributed *transactions*. Although traditional protocols, introduced a few decades ago, can be used to implement such a system, we show a few ways in which they are being challenged by the latest advancements and trends.

Ensuring user data privacy (by enforcing user approved policies over their personal data) has always been important and is becoming even more so since the last years have witnessed a massive amount of personal information being shared through large-scale cloud applications. Today's ubiquitous mobile devices sense, collect, and store huge amounts of personal information, which users share with family and friends through a wide range of applications. Once users give applications access to their data, they must implicitly trust that the apps correctly maintain data privacy. As we know from both experience and all-too-frequent press articles, that trust is often misplaced. Recently, GDPR (General Data Protection Regulation) has recognized the growing concern and importance of ensuring data privacy and aims to pressure businesses to give users control over their personal data. While users do not trust applications, they generally trust their mobile devices and operating systems. Unfortunately, sharing applications are not limited to mobile clients but must also run on cloud services to share data between users. Today, the burden to ensure data privacy falls on the application developers. In Section 1.2 we argue for the need to delegate this responsability to a trusted third party privacy-preserving system and present the challenges that must be addressed in order to provide such a system.

1.1. STORAGE SYSTEMS FOR LARGE-SCALE CLOUD APPLICATIONS

The extreme demands of large-scale cloud applications point to several design characteristics and properties that a suitable storage system must have (Section 1.1.1). Moreover, a few notable hardware advancements motivated new design trends that, if adopted by the storage system in use, can benefit large-scale cloud applications (Section 1.1.2). However, despite significantly improving their performance, these hardware advancement and design trends also introduce new challenges by exposing significant overheads in the classic protocols used to implement these storage systems (Section 1.1.3).

1.1.1. DESIRED DESIGN

Distributed. Storage systems for large-scale application must generally be *distributed* (deployed on multiple storage servers) in order to meet the large-scale requirement. A single storage server is usually not sufficient to store all the large-scale application's data or provide the necessary throughput. For this reason, the data is partitioned across multiple storage servers which can serve requests in parallel. Even after data partitioning, a partition can still become a "hotspot" if it must be accessed at a higher rate than the storage server can support. In this case, the partition can be replicated on multiple storage servers with a replication protocol that is able to load balance the read workload across the replicas.

Fault-tolerant. Large-scale cloud applications must be online at all times and thus require their storage system to remain available despite server failures. To meet this requirement, distributed storage systems replicate the data on multiple servers – thus, if a server fails, the other remaining servers can still continue to serve client requests.

Multi-threaded. Large-scale cloud applications have stringent prformance requirements. This strongly motivates storage systems for large-scale cloud applications to pursue an efficient and scalable *multi-threaded* design, to take full advantage of the computational power on the storage servers.

Provide serializable distributed transactions. For many large-scale cloud applications a simple, NoSQL design (key-value) suffices; for some, a SQL design (relational tables) is still necessary. Regradless of the base interface chosen, we believe providing *transactions* on top of it is necessary to ensure the correctness of the majority of these complex applications. Transactions have been introduced to simplify the process of developing applications that must deal with concurrent accesses to the stored shared data. A transaction is a program that encapsulates one or more accesses to the stored shared data. Transactions are processed through the means of a concurrency control mechanism that ensures that transactions execute *atomically*, which means that transactions do not interfere with eachother and if a transaction terminates normally then all its effects are made permanent, otherwise it has no visible effects. The notion of transactions interference has been formalized through a correctness property called *isolation*. *Serializability* is an isolation level that offers strong guarantees: transactions appear to have executed in some serial order. Providing strong isolation gurantees generally results in lower per-

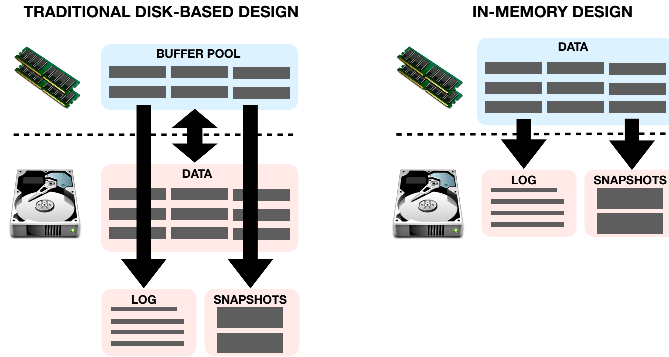


Figure 1.1: Comparison between the traditional disk-based storage system design and the in-memory storage system design. The most notable difference in the two designs is where the data is stored. In the traditional design, the data resides on the secondary storage, in a specialized data structure (e.g., a tree optimized for block disk accesses). In the in-memory design the data resides in memory, in a simpler data structure (e.g., hashmap, tree). An in-memory design can speed up the storage system by up to a few orders of magnitude by allowing the use of simpler data structures to hold the data in memory, as opposed to complex data structures specialized for the secondary storage, by reducing the number of secondary storage accesses required, and by eliminating the need for random access to secondary storage.

formance. However, since the latest hardware advancements and design trends can significantly improve the performance of distributed storage systems (as we describe later), strong isolation guarantees, such as *serializability*, are increasingly preferred.

1.1.2. NEW HARDWARE ADVANCEMENTS AND DESIGN TRENDS

Since the design of the first storage systems, hardware capabilitites have consistently improved over the years. Today, the servers running these systems are much more powerful – their processors are significantly faster, with many more cores, and they are equipped with better storage (memory, HDD, SSD), both faster and with higher capacity. The network infrastructure (NICs, switches, routers, cables, networking protocols) has more that kept up with the computational power of the servers – they are fast enough to process enough packets per second to saturate the server's cores. A few notable hardware and network infrastructure trends are further motivating the particular storage design we mentioned above: *big memory*, *multi-processors*, *worldwide datacenters*, and *fast networks.*

Big memory and the in-memory design. Memory capacity has increased significantly in the last decades. Today, a single server can be equipped with 24TB of RAM. This has motivated storage systems to switch to the much faster *in-memory* design, which as-

sumes that the data can fit into the memory at all times and thus it can use in-memory data structures, without the complexity of designing data structures specialized for the secondary storage (SSDs, HDDs). Figure 1.1 shows how an in-memory storage design differentiates itself from the traditional designs that store the data on the secondary storage. As it can be observed, in the in-memory design accesses to the secondary storage are required only when logging update operations or, periodically, when saving snapshots. Apart from being less frequent, these secondary storage accesses can be implemented more efficiently since they do not require random access, which is usually very expensive for existing secondary storage drives. As a consequence, the in-memory design can increase the speed of the storage system by up to a few orders of magnitude.

Multi-processors. To be able to equip a single server machine with more and more cores, multi-processor architectures were developed; these architectures have evolved signinficantly in the last decades. Today, a single server can support an 8-socket configuration, where each socket can hold a multicore processor comprising up to 28 cores, for a total of 224 cores.

Worldwide datacenters and the geo-distributed design. Cloud providers today own multiple datacenters worldwide, each datacenter consisting of multiple server machines. Storage systems for large-scale applications, which normally have users all around the world, perform better and are more reliable if they are *geo-distributed* and *geo-replicated* – this design not only allows applications to bring data closer to users' locations, but also tolerate an entire datacenter failure.

Fast networks and the kernel-bypass design. Networks are becoming faster and faster. Today's network cards can attain 200 Gbps Ethernet speeds and process around 200 million messages per second. These high throughput rates have forced some innovations in the way messages are processed by the processor, since the processor's speeds have not improved as dramatically over the last decade. Traditionally, network messages are being processed by the OS kernel, which incorporates a network processing stack, before ultimatelly being presented to the application. Shifting to the kernel space is very expensive and processing a single message on the Linux's kernel stack can take around 5 μs. Kernel-bypass libraries have been introduced to alleviate this problem. As shown in Figure 1.2 these libraries fully operate in the user space and can reduce the cost for processing a single message to 0.3 μs.

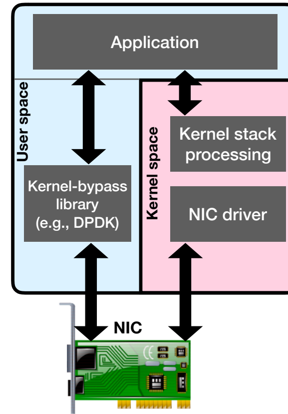


Figure 1.2: Comparison between the traditional method for processing network messages which uses the kernel network stack and the new kernel-bypassing libraries which operate entirely in user space.

1.1.3. CHALLENGES

To implement the distributed storage system design motivated above – namely an *in-memory*, *geo-distributed*, *fault-tolerant*, *multi-threaded* storage system that provides *strong correctness guarantees*, specifically *serializable* distributed *transactions* – previously proposed solutions have been layering three protocols that solve three different sub-problems which have been formalized and studied in isolation a few decades ago: an *atomic commitment* protocol which ensures the atomicity of transactions (i.e., if a transaction commits then all or none of its updates are applied), a quorum and consensus-based *replication* protocol which provides fault-tolerance (i.e., the system continues to operates normally without external intervention despite sever crashes), and a *concurrency control* protocol which ensures that transactions appear to have been executed in some global serial order. These solutions are now being challenged by the latest advancements in hardware and network infrastructure, presented above, which have altered the major source of latency and the throughput bottleneck. Here are a few challanges that we address in this thesis:

Cross-datacenter coordination is expensive. Communication latencies between two datacenters (round trip times) can reach hundreds of miliseconds. While in the past transaction processing times were more concerning, since processing transactions required frequent accesses to a very slow secondary storage and the storage systems were not geo-distributed, today's in-memory design and much faster secondary storage (with

accesses in the order of a few miliseconds) make the cross-datacenter coordination latencies the dominant term in the end-to-end latency costs. Previously proposed solutions require at least two cross-datacenter communication rounds and to contact a certain datacenter (where a *leader* server is situated) to commit transactions; in many cases, this results in a few hundred ms latencies.

Cross-processor coordination is expensive. In multi-socket architectures, processors are situated in different NUMA nodes, where each node has fast access to parts of the memory but slow access to other parts of memory (i.e., the ones attached to other NUMA nodes). These processors utilize a cache coherence mechanism to serialize cores' writes to memory and, thus, sharing cache lines among cores from different processors can be very expensive if they must be frequently written. Multi-threaded designs must usually protect against race conditions by using low-level atomic operations, provided by the hardware (like compare-and-swap, test-and-set), or locks. If the same lock(s) must be taken by all threads when processing transactions, as is the case with previously proposed solutions, then the throughput of the storage system deployed on multiprocessor architectures is limited to around 2-3 million transactions per second. In the past, transaction processing was expensive and the throughput bottleneck was the secondary storage; today's fast transaction processing on faster secondary storage expose this new bottleneck.

Recovering efficiently from replica failures is hard in the absence of persistent storage devices. Distributed storage systems that are required to provide durability, log a lot of data to the persistent secondary storage device so that, even if all replica servers are shut down at the same time, they can easily recover, independently, next time they restart. However, there are applications, such as statistical processing, graph processing, data analytics, machine learning, online gaming, that do not necessarily require or afford durability, and, thus, they trade it off for performance; for these systems in-memory replication is sufficient to keep them online for a long enough time. Providing a fast recovery mechanism in these circumstances, where the replica servers cannot rely on local persistent storage devices to recover their latest state, is more challenging: replica servers must recover their state by contacting the other replica servers and they must do so by making sure no correctness invariants are being invalidated. Previously proposed solutions are either not correct or are too expensive.

1

1.2. PRIVACY-PRESERVING SYSTEMS FOR UNTRUSTED LARGE-SCALE CLOUD APPLICATIONS

TODAY, mobile devices are ubiquitous. Smartphones, smartwatches, tablets, etc., constantly collect information about their users. Most of this data is shared with other users through mobile-cloud applications. To encourage users to use them as much as possible, these applications must provide guarantees regarding the privacy of the users' data. In fact, the recently introduced GDPR (General Data Protection Regulation) actually forces these applications to integrate privacy by design, as an effort to give users control over their personal data.

Providing privacy guarantees is not an easy task and, currently, the mobile-cloud application developers are the ones responsible for implementing the mechanism that enforces the privacy policies desired by the user. Therefore, users are left to blindly trust that applications will respect their privacy even as the applications move their data across a complex landscape of backend servers, storage systems and cloud services. Unfortunately, this trust is often misplaced due to bugs, such as missing policy checks, or vulnerabilities in the code that attackers can exploit in order to leak private information.

Therefore, there is an urgent need to provide a trusted third-party privacy-preserving system that would automatically enforce existing privacy policies on unmodified and untrusted sharing applications across mobile devices and cloud services. Both users and application developers would benefit from outsourcing privacy policy enforcement to a separate system. Users can trust that their privacy policies are upheld even if the application is untrustworthy or buggy – an important benefit in a world of millions of largely anonymous developers and applications. And developers no longer have to be security experts, perfectly implementing complex privacy features to avoid leaks; thus, they benefit from increased trust and assurance, and can use the privacy-preserving system as a "safety net" to prevent unintentional data disclosure.

1.2.1. CHALLENGES

Designing a general-purpose privacy-preserving system to automatically enforce users' privacy policies on existing untrusted mobile-cloud applications is challenging. We address the following in this thesis:

How to determine what sharing policies the user intended, while providing the same user experience. Each application provides its own graphical interfaces that users use to introduce their data and share it with the application, and set the desired privacy policies (e.g., allowing the application to share the data with only a certain group). Because these interfaces have been implemented by the application developers, they cannot be trusted. Providing trusted graphical interfaces to collect users' data and policies without changing the user experience is challenging. First, the trusted graphical interfaces must emulate the ones provided by the applications themselves, and, second, the application and the privacy-preserving system must somehow agree on the semantic meaning of the objects being transferred from the trusted interfaces to the application.

How to enforce user policies against untrusted applications, with minimal performance cost. While developers understand the functionality of the application code and can better decide on the minimum amount of policy checks they need to add to the code in order to enforce the policies attached with the data, a more automatic, general-purpose, privacy-preserving system must track the data as it traverses through the application and make sure the policy is enforced at every step. Mobile-cloud applications are a complex piece of software, deployed both on the mobile devices and multiple servers in the cloud. Therefore, naively tracking the data flow and performing the policy checks could impose an unacceptable performance penalty.

1.3. CONTRIBUTIONS

THIS thesis addresses the challenges we presented in the previous sections and proposes new storage and privacy-preserving systems for large-scale cloud applications. Parts of this thesis have been published in the following works:

1. **Adriana Szekeres**, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, Irene Zhang, *Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle*, Proceedings of the 15th European Conference on Computer Systems (EuroSys, 2020).

2. **Adriana Szekeres**, Ellis Michael, Naveen Kr. Sharma, Dan R. K. Ports, *Recovering Shared Objects Without Stable Storage*, Proceedings of the 31st International Symposium on Distributed Computing (DISC, 2017).

3. **Adriana Szekeres**, Irene Zhang, Katelin Bailey, Isaac Ackerman, Haichen Shen, Franziska Roesner, Dan R. K. Ports, Arvind Krishnamurthy, Henry M. Levy, *Making*

Distributed Mobile Applications SAFE: Enforcing User Privacy Policies on Untrusted Applications with Secure Application Flow Enforcement, arXiv:2008.06536.

In the rest of this section we give an overview of the new systems and protcols we proposed; they are covered extensively in the next chapters of the thesis.

1.3.1. FAST AND MULTICORE SCALABLE REPLICATED TRANSACTIONS: MEERKAT

First, this thesis contributes *Meerkat*, an in-memory, multi-threaded key-value storage system that provides fault-tolerance and one-copy serializable distributed transactions *without any cross-core and cross-replica coordination when executing transactions that do not conflict*. Meerkat does not require a leader to replicate transactions, which commit in a single round trip in the absence of failures and conflicts (the theoretical minimum); this significantly reduces latency in a geo-ditributed setup. Meerkat also eliminates the cross-processor coordination bottleneck described above by requiring memory contention only when processing transactions that conflict.

Our experiments found that Meerkat is able to scale up to 80 hyper-threads and execute 8.3 million transactions per second. Meerkat represents an improvement of 12× on state-of-the-art, fault-tolerant, in-memory, transactional storage systems built using leader-based replication and a shared transaction log.

1.3.2. FAST RECOVERY FOR IN-MEMORY REPLICATED STORAGE: DISKLESS RECOVERY

Second, this thesis contributes a set of communication primitives that help design correct and fast replica recovery protocols for fault-tolerant storage systems that do not use persistent storage devices (i.e., storage devices that retain data even after their power is shut off). The communication primitives implement a new abstraction, the *crash-consistent quorum*, where no recoveries happen during the acqusition of quorum responses – this guarantees that once the data has been replicated at this quorum, all replicas that successfully recover afterwards see that data. We show that relying on crash-consistent quorums enables a recovery procedure that can recover all operations that successfully finished.

Crash-consistent quorums can be easily identified using a mechanism we term the *crash vector*, which tracks the causal relationship between crashes, recoveries, and other operations. We apply crash-consistent quorums and crash vectors to build two storage primitives. We give a new algorithm for fault-tolerant multi-writer, multi-reader atomic

registers that guarantees safety under all conditions and termination under a natural condition. It improves on the best prior protocol for this problem by requiring fewer rounds, fewer nodes to participate in the quorum, and a less restrictive liveness condition. We also present a more efficient fault-tolerant single-writer, single-reader atomic set—a *virtual stable storage* abstraction. It can be used to lift any existing algorithm from the traditional model that uses persistent storage devices for recovery to the model without persistent storage devices. We examine a specific application, state machine replication, and show that existing protocols can violate their correctness guarantees, while ours offers a general and correct solution.

1.3.3. CONFIDENTIALITY SYSTEM FOR UNTRUSTED CLOUD APPLICATIONS: SAFE

Finally, this thesis contributes SAFE (*Secure Application Flow Enformcement*), a data privacy-preserving system for mobile-cloud applications. SAFE requires cloud services to attest to a system stack that will enforce policies provided by the mobile OS for user data. We implement a mobile OS that enforces SAFE policies on unmodified mobile apps and two systems for enforcing policies on untrusted cloud services. Using these prototypes, we demonstrate that it is possible to enforce existing user privacy policies on unmodified applications.

1.4. OUTLINE

THE rest of the thesis is organized as follows. In Chapter 2 we cover the necessary background and related work. In Chapter 3 we present Meerkat, our new in-memory, multicore scalable distributed storage system. Chapter 4 details our new primitives that help design fast and correct recovery mechanisms for in-memory storage systems that do not use persistent storage devices. In Chapter 5 we present the new privacy-preserving system for untrusted mobile-cloud applications. Finally, we conclude with some future directions in Chapter 6.

2

BACKGROUND

Before delving into the details of our new systems, we give some background on the protocols and mechanisms we built upon, provide pointers to some recent related work, and argue why previous work fails to address all the challenges we introduced in the previous chapter. Detailed related work sections are provided in each of the subsequent chapters.

2.1. DISTRIBUTED STORAGE SYSTEMS

Distributed storage systems, formed by networking together multiple storage server machines, are expected to achieve three goals: high *availability*, high *performance* – exhibit high throughput and low latency –, and provide the *functionality* – application programming interface and correctness guarantees – desired by the application.

Designing a distributed storage system to achieve all these three goals is inherently difficult – the design of such systems must take into consideration varied application requirements and workload characteristics, varied hardware capabilities, and the fundamental tensions between avalability, performance and functionality. The fundamental tension between availability and functionality is formulated by the CAP theorem which states that a distributed storage system cannot guarantee both availability and corectness if the network is prone to partitions. Since in a realistic scenario the network is indeed prone to partitions, a distributed storage system must sacrifice either availability or

correctness – it either blocks until the partition is repaired or returns inconsistent results, thus weakening the correctness guarantees. Besides this fundamental tension between functionality and availability, there is also a tension between functionality and performance – distributed storage systems that provide weaker correctness guarantees generally produce higher throughput and have lower latency. Also, a performance penalty is usually incurred when increasing the availability of the system.

The right trade-offs to address these tensions are dictated by the application requirements and workload characteristics, and by the hardware capabilities, which have all changed dramatically over the years, fuelling numerous distributed storage system designs. In this thesis, we focus on a set of specific design characteristics and properties, suitable for large-scale cloud applications, as we described in Chapter 1; specifically, we are interested in designing an in-memory, geo-distributed, fault-tolerant, multi-threaded, transactional storage system that guarantees strong corectness properties. Additionaly, we want the system to be multicore scalable and to require minimal cross-datacenter communication.

Designing such a system is a very complex task, but, fortunately, we do not need to start from scratch; we can build on decades of previous research in this space. Esentially, this complex task of building such a system can be broken down into smaller pieces: 1) scheduling transactions, 2) ensuring transaction atomicity, and 3) replicating the system. All these three pieces have been formalized into now well-known problems and many solutions have been proposed over the years: to schedule transactions, *concurrency control* protocols have been proposed; to ensure that all of a transaction's changes are visible or none at all, *atomic commitment* protocols have been proposed; and, finally, to consistently replicate the system on multiple storage server machines, *replication* protocols have been proposed (which include recovery from various failure senarios and membership reconfiguration mechanisms). Combining these proposed protocols to obtain the final solution is not always straight-forward and can lead to novel designs.

2.1.1. CONCURRENCY CONTROL

As the name suggests, the concurrency control protocol is used to control the concurrent execution of transactions such that all successfully executed transactions adhere to a well-defined specification that describes, in an implementation agnostic way, all the permissible interleavings between the transactions. These specifications are known as *isolation levels*. *Serializability* is, arguably, the most well known and studied isolation level provided by transactional storage systems. Serializability offers a strong guarantee,

making it easy for developers to reason about and implement the desired correctness properties for their applications: all successful transactions appear to have executed in some serial order (a formal definition and methods to prove that a concurrency control provides serializability can be found in [1]).

There are two main classes of concurrency control protocols: pessimistic and optimistic (hybrids between the two have also been proposed). Pessimistic concurrency control, like 2PL (two-phase-locking), prevents conflicts by acquiring locks before accesing the data item, while optimistic concurrency control allows transactions to execute "optimistically" by accessing data unconstrained, while conflicts are detected at the end, with a validation check. Many studies have been conducted to understand the differences between pessimistic and optimistic concurrency control (see [2] for a recent study). In designing our storage system we chose to provide a variant of optimistic concurrency control, which is also the preferred option in many previously proposed multi-threaded transactional storage systems. Optimistic concurrency control generally preforms better for low to medium contended workloads, which we believe are consistent with the workloads of large-scale cloud application.

In order to provide serializable distributed transactions (transactions that access data from multiple storage servers), the concurrency control protocol must ensure that the execution of succesful transactions is equivalent to a serial order that is consistent across partitions. Designing distributed optimistic concurrency control protocols is more challenging [3], and often requires partitions to share concurrency control information through an atomic commitment protocol descibed below.

2.1.2. Atomic commitment

The atomic commitment protocol ensures the atomicity of distributed transactions despite storage server failures and recoveries. The most commonly used atomic commitment protocol, 2PC (two-phase-commit), is driven by a transaction coordinator, for each transaction individually, and operates in two phases to ensure that each participant (storage server holding a data partition that was accessed by the transaction) that prepared to commit the transaction (in the first phase) will not later un-prepare the transaction unless the transaction is aborted by the transaction coordinator (in the second phase).

2PC allows the transaction coordinator to simply abort a transaction if it doesn't hear back in time from all the participants. This decision is propagated to the other participants only after it has been made durable at the coordinator (which might have replied

back to the client). Since in a fault-tolerant distributed storage system each participant is replicated (with a replication protocol we describe below) it is reasonable to assume that all participants will eventually reply, after they replicated the prepared transaction, and thus the transaction coordinator does not need to ever abort a transaction that prepared at all participants. This avoids the need to replicate the transaction coordinator as well, as described in more detail in MDCC [4] and TAPIR [5].

The atomic commitment protocol is tightly integrated with the distributed concurrency control protocol and sometimes, concurrency control information is piggybacked on the messsages send as part of the atomic commitment protocol.

2.1.3. REPLICATION

As we argued in the previous chapter, large-scale cloud applications must be online at all times. Thus, it is critical for the distributed storage system in use to be highly available and be able to overcome server and network failures. In order to improve on its availability (and prevent the system from stalling for an indefinite amount of time, until the failure has been repaired), a distributed storage system integrates a replication protocol that provides *fault-tolerance*, i.e., the ability of the system to continue to operate despite a certain number of potentially permanent replica server failures.

The most popular replication scheme for transactional distributed storage systems, due to its simplicity, is a *centralized* one, where only one designated replica server, the *primary*, is executing and scheduling all the transactions (it is the only one actively running the concurrency control protocol and deciding the outcome of the transactions). Transactions that successfully commit at the primary are then synchronously propagated to the other replicas; they are propagated either in the form of commands that must be re-excuted in a way that is consistent with the execution at the primary or in the form of updates that must be applied in a primary-specified order.

So how is this replicated distributed system fault-tolerant? What happens if a replica server fails? Even worse, what if that replica server was the primary? We want the system to be able to recover from such failures without external intervention, such as an admin that manually reconfigures the system, as it would take too much effort and time: the admin must ensure the old primary will never recover in the future thinking it is still the primary, and then he must configure another replica server as the new primary. In other words, we want the system to be able to *automatically* reach a configuration where it can make progress again, without violating correctness (ex., losing completed transactions or violating serializability).

Automatically electing a new primary or performing a membership reconfiguration to eliminate the failed machines (as required by some replication schemes in order to be fault-tolerant, as we elaborate later), while maintaining correctness, is hard to do due to the asynchronous nature of the network (which can drop or delay indefinitely messages) and of the server machines (which can take an indefinite time to perform a computation) – for example, the system might erroneously detect that the primary failed; if the system then elects, naively, a new primary, it might reach a situation where there are two primaries trying to execute transactions in parallel.

Fortunately, these failure scenarios have been studied extensively in order to solve the problems of *consensus* and *state machine replication*. The basis for clasic state machine replication protocols is a consensus-based protocol that implements a very powerful abstraction – a fault-tolerant log (ordered set). This protocol, which is leader-based, where a leader is the designated replica server that can append elements to the log, analogous to the primary in the replication scheme above, offers solutions to the failure scenarios we described above and can be easily integrated with the primary-based replication scheme (the leader is merged with the primary; after a successful leader change, the new leader will inherit a consistent log which the new primary can use to ensure serializability).

So how many potentially permanent failures can the system tolerate? This depends on whether we use an external reconfiguration service or not, as we elaborate next. Suppose our distributed storage system uses N replica storage servers. If we do not use any additional service, then our system using the consensus-based fault-tolerant log can tolerate at most $\frac{N-1}{2}$ replica failures, like any classic state machine replication protocol. With this scheme the primary needs to synchonously propagate the committed transactions to *any* majority of replica servers (including itself), failures of replicas that are not the primary are *seamlessly* tolerated, and primaries that are suspected to have failed are replaced with a primary election protocol. If an external configuration service is used, as described in Vertical Paxos [6], Chain Replication [7], then up to $N-1$ storage server replica failures could be tolerated. However, since the external configuration service uses a state machine replication protocol itself, let's say with M replicas, in reality the whole system can tolarate $\min(N-1, \frac{M-1}{2})$ failures out of $N+M$ servers. In this scheme the primary needs to synchronously propagate the committed transactions to *all* the other replicas and, upon *any* failure, the external configuration service is used to reconfigure the system and eliminate the failed machine from the new configuration and, if needed, pick the new primary. Both schemes can use reconfiguration protocols to replace the failed machines and restore the fault-tolerance of the system.

As we explain later, in Section 2.1.5, this *centralized* replication approach, although simple, it has some disadvantages. The new distributed storage system we introduce in this thesis uses a *decentralized* replication approach, where every storage server replica executes transactions and runs the concurrency control protocol. There is very little previous work that follows this scheme. The first such scheme was introduced in Majority Consensus [8]. The only other work in this space we are aware about are some very recent systems: MDCC [4], Janus [9], and our previous work, TAPIR [5]. None of these systems discuss multicore scalability and some require certain information about transactions to be known apriori.

We also chose to follow the scheme that does not use an external reconfiguration service to tolerate failures, which is preferred by recent systems like Spanner [10], since it can *seamlessly* tolerate a number of failures and it can offer better tail latency since the primary needs to wait only for the fastest $\frac{N}{2}$ replies.

2.1.4. MULTICORE SUPPORT

Distributed storage systems take advantage of the multicore architecture by adopting either a *shared-nothing* or a *shared-everything* design. In a shared-nothing design, the set of data items is partitioned across cores (similar to how the set of data items is partitioned across storage servers) such that only one core is allowed to access a given partition, which can completely eliminate the need for shared-memory synchronization between cores. This design is great if the system runs certain workloads, where most of the transctions need to acces a single partition and partitions do not form a hotspot; otherwise, its performance quickly degrades with the number of cores.

In contrast, a shared-everything design allows all cores to access any data item, which usually requires synchronization mechanisms to avoid race conditions. We chose to provide a shared-everything design since it has the potential to offer more parallelism. As we show later in this thesis, we can design our shared-everything distributed storage system to require cross-core synchronization only when transactions conflict, in which case the differences between the the two design approaches can be roughly seen as analogous to the differences between coarse-grained and fine-grained locking.

2.1.5. OVER-COORDINATION IN EXISTING SOLUTIONS

For a long time, the performance of ditributed storage systems has been hindered by slow persistent storage devices and slow networks, with high message processing costs.

These bottlenecks have been hiding the overheads associated with the protocols used to implement such systems. Today's hardware advancements (Section 1.1.2) are exposing these overheads, forcing us to re-think the way we design such systems.

The overhead of cross-replica coordination. The traditional primary-based replication scheme that uses the fault-tolerant log abstraction as we described above, imposes a serious performance penalty on transactional distributed storage systems, because it enforces strict serial ordering using expensive distributed coordination in two places: the replication protocol enforces a serial ordering of operations across replicas in each shard, while the distributed concurrency control protocol enforces a serial ordering of transactions across partitions. This redundancy impairs latency and throughput for systems that integrate both protocols. The replication protocol must coordinate across replicas on every operation to enforce strong consistency; as a result, it takes at least two round-trips to order any read-write transaction. Further, to efficiently order operations, these protocols typically rely on a replica leader, which can introduce a throughput bottleneck to the system.

The overhead of cross-processor coordination. As we mentioned above, we are interested in a shared-everything design, where all the threads are allowed to access all data items. To avoid race conditions, the algorithms implementing this design generally require thread synchronization mechanisms like atomic regions (implemented using locks) or low-level synchronization instructions provided by the processor, like compare-and-swap. These synchronization mechanisms can be very expensive on any shared-memory multicore processor that uses a cache coherence protocol because they require sharing cache lines between cores. Writing a cache line that was last read or written by another core does not scale since the coherence protocol serializes ownership changes for each cache line. On multi-processor architectures, the overheads are much higher due to the higher cost of cross-processor communication [11]. Previous designs suffer from these synchronization overheads due to shared structures that require coordination even for non-conflicting transactions and hence become points of contention. For example, the log data structure employed by the replication protocol must be accessed by the threads for every transaction and many optimistic concurrency control protocols use shared data structures, like lists of validated transactions, timestamp counter, which must also be accessed for every transaction.

2.2. PRIVACY-PRESERVING SYSTEMS

Ensuring users' data privacy is still largely the responsability of application developers. Many modern mobile OSes incorporate *access control* which allows users to restrict application access to various resourses (images, location, etc.). However, once the application gets access to that information, users are forced to trust that the application meets their privacy policies. Unfortunately, often times this trust is misplaced; developers are prone to making mistakes, such as forgetting to place critical privacy policy checks.

Desigining a system to automatically enforce users' privacy policies over applications is thus very appealing. Several such proposed systems use the *sandboxing* mechanism. These systems run applications in a constrained environment such that each user's data is managed by a *sandbox* that doesn't allow the data to be leaked. While data isolation makes it easy to enforce privacy for applications where users do not interact, social applications where users want to selectively share their data do not work well. For these, another mechanism, based on *information flow control*, would be more appropriate.

2.2.1. INFORMATION FLOW CONTROL

Information flow control (IFC) tracks how information propagates through the application and makes sure the application does not handle the information in a way that violates its confidentiality and integrity. IFC research started several decades ago, primarily in the context of military systems. Since then, the field has matured dramatically and the mechanism has been rigorously defined (in the context of many programming languages), in order to be able to understand the precise security guarantees it provides. A detailed overview of the state-of-art in IFC can be found in [12].

A system that integrates an IFC mechanism to enforce users' sharing privacy policies must provide solutions to a few challenges. First, the system must be able to *automatically* translate users' high-level privacy policies into IFC *labels*, pieces of metadata used by every IFC mechanism to describe the security level of the data (as opposed to IFC-based platforms for developers, where developers can directly attach the desired labels to the relevant program variables). Second, the system must identify all the points where data transfer happens to make sure that the IFC *rules* are not violated (these IFC rules, specified as part of every IFC mechanism, describe how the information is allowed to flow between IFC labels). Finally, the system must deal with the fundamental tension between security and functionality that characterizes IFC mechanisms – the problem of *declasification*.

2.3. CONCLUDING REMARKS

Distributed storage systems have been extensively studied over the last several decades. Researchers have broken them down into multiple problems that have been formalized and studied in isolation – this significantly increased our understading of these systems. Still, continuous hardware advancements motivate new designs which challenge the existing solutions. On the other hand, privacy-preserving systems that automatically enforce user policies on untrusted applications are not as mature – in the current world we still largely rely on application developers to enforce privacy policies. Information flow control is a promising idea but it is really hard to put into practice.

3

MEERKAT

As we argued in Chapter 1, replicated, in-memory, multi-threaded, distributed transactional storage systems have emerged as an important piece of infrastructure for large-scale cloud applications because they combine fault-tolerance and strong semantics with high throughput and low latency. The performance demands placed on these systems are extreme; e-commerce, social media, and cloud-scale storage workloads, among others, can require tens of millions of transactions per second [13, 14]. Meeting these demands requires a system structured to eliminate coordination bottlenecks both within each server and across the cloud.

For many years, distributed or replicated storage systems have had the luxury of ignoring the cost of *cross-core* coordination within a single node, as it has been masked by the far higher cost of *cross-replica* coordination. Even in a local-area network, the principal bottlenecks have come, by a substantial margin, from network round trips and the cost of processing packets [15, 16]. Indeed, the gap between communication and computation cost has been so large that some have proposed restricting execution to a single core to simplify protocol design [17, 18]. But the era of slow networks has come to an end. The past decade has brought into the mainstream both faster network fabrics and kernel-bypass network technologies that dramatically lower the cost of packet processing. The result is that it is possible to build a system that pushes the limits of cross-core coordination.

In fact, the effects are already visible on today's widely available hardware. Figure 3.1

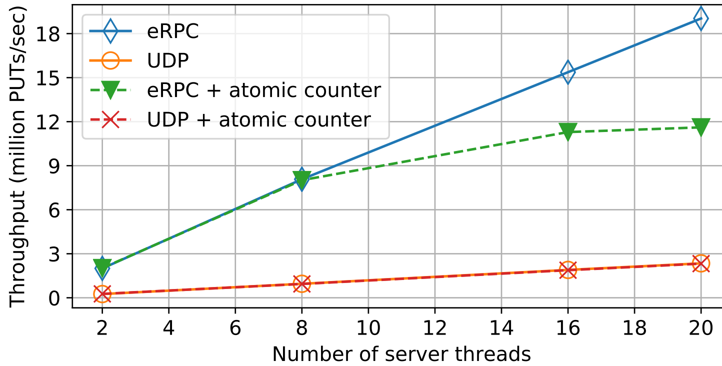


Figure 3.1: Peak throughput comparison of PUT operations for a simple key-value storage system implemented on both a traditional Linux UDP stack and on a recent kernel-bypass stack, eRPC. The 8× improvement in performance uncovers cross-core scalability bottlenecks in the application – a simple shared atomic counter incremented on every PUT operation bottlenecks the system at 11 million operations/second.

compares the performance of a simple key-value storage system implemented using the traditional Linux UDP network stack on a 40 Gb Ethernet network and with eRPC [19], a recent kernel-bypass network stack. In both cases, increasing the number of cores used allows the system to handle more PUT requests, but the eRPC implementation has 8× higher throughput than the UDP implementation. More importantly, when we introduce an artificial scalability bottleneck in the application – a simple atomic shared counter, incremented on every operation – the effect is quite different. The application bottleneck has no discernable effect (up to 20 cores) with the traditional UDP implementation, as it is masked by bottlenecks in the Linux network stack. With the optimized eRPC stack, however, application-level scalability bottlenecks have a major effect. That is, for the first time, the distributed system causes the bottleneck at higher core counts.

Can replicated storage systems scale with increasing core counts? A fruitful line of recent work [20–25] has made great strides in allowing *single-node* transaction execution to take full advantage of multicore processors. Extending this to the distributed environment, however, runs afoul of some fundamental challenges. Keeping replicated systems consistent requires ensuring that each replica reflects the same order of execution. Multicore execution, however, is inherently non-deterministic, a problem for the state machine approach [26]. Other replicated systems are commonly built with shared-memory data structures (e.g., logs) that are themselves challenging to scale.

In this chapter we take a principled approach to multicore design that systematically eliminates bottlenecks in replicated storage systems. We introduce the *Zero-Coordination Principle* (ZCP), which states that a replicated storage system with no coordination be-

tween replicas *or* cores will be multicore-scalable. ZCP extends disjoint access parallelism, a classic architecture property, to the distributed systems realm.

We demonstrate that this approach yields multicore-scalable replicated storage systems: we present *Meerkat*, the first replicated, in-memory, multi-threaded distributed transactional storage system that is multicore scalable. Meerkat uses a mix of new and old design techniques to achieve a replicated transaction protocol with no cross-core or cross-replica coordination. Moreover, Meerkat transactions commit in a single round trip in the absence of failures or conflicts (the theoretical minimum), cutting on the expensive round trip latencies in a geo-distributed setting. Our Meerkat prototype must further ensure that its entire software stack is coordination-free: it uses eRPC for kernel-bypass, x86 atomic instructions, and other optimizations to minimize cross-replica and cross-core latency. Experiments with our prototype demonstrate that Meerkat scales up to 80 hyperthreads and executes 8.3 million YCSB-T [27] transactions/second.

To summarize, this chapter makes the following contributions:

- A new *Zero-Coordination Principle* (ZCP) that guides the design of multicore-scalable, replicated systems.
- The first multicore-scalable, replicated, in-memory, transactional storage system, Meerkat, designed using this principle.
- A performance comparison of Meerkat with existing systems to evaluate the impact of violating ZCP on multicore-scalability and performance.

3.1. THE ZERO COORDINATION PRINCIPLE

OUR goal is to build replicated systems that are both *multicore-scalable* and *replica-scalable*. More precisely, if the system is executing non-conflicting transactions – those whose read and write sets are disjoint – **the number of completed transactions per core should not decrease** even as the total number of cores in the system increases. It should be possible to scale up the system throughput by using nodes with more cores or increase the system fault tolerance by adding more replicas without a performance penalty. That is, the only barriers to scalability should be ones that are fundamental to the workload – those that arise from conflicts between transactions.

Most systems do not achieve both multicore scalability and replica scalability due to shared structures that require coordination for non-conflicting transactions and hence become points of contention. The *Zero-Coordination Principle* states that scalable systems can be built by following two rules:

Multicore scalability: DAP. The first half of ZCP is a familiar one from previous work on designing multicore-scalable applications, namely *disjoint access parallelism* (DAP) [28]. The DAP principle states that a system will be able to achieve ideal scalability on multicore processors if non-conflicting transactions – ones that access disjoint sets of memory locations – access disjoint memory regions. This means that the implementation requires no cross-core coordination. Later work extended the DAP principle to transactional memory [29, 30].

DAP means that there should be no centralized points of contention beyond the data items themselves, such as transaction ID management or database lock managers. Recent work has built single-node transaction processing systems that minimize or eliminate these central contention points [20–25]. Replicated systems tend to involve further points of contention, such as shared logs or other state tables, that must be eliminated for a replicated system to support ZCP.

Replica scalability: coordination-free execution. Our second performance goal, that performance should not decrease as the number of replicas increases, is addressed by the second half of ZCP. Much as DAP required that non-conflicting transactions not access the same memory regions, ZCP imposes an additional requirement that non-conflicting transactions do not require *cross-replica* coordination, i.e., replicas do not need to send messages to each other in order to commit non-conflicting transactions.

For example, a traditional state machine replication protocol [31–34] (or shared log), the basis for many transaction processing architectures [10, 35–37], does not meet this requirement. It requires establishing a total order of operations through a distributed protocol, e.g., by serializing them through a leader replica, a form of cross-replica coordination. This generally causes $O\left(\frac{1}{n}\right)$ throughput scaling in the number of replicas, though some replication and transaction processing protocols have been designed to avoid this [5, 7, 15].

Although sufficient to meet our second performance goal, this ZCP requirement is stricter than necessary and implies a design where the task of replicating transactions is offloaded to the clients. For example, one could potentially employ chain replication [7] to build an equally scalable system but at a higher latency cost (i.e., the message delay is proportional to the number of replicas). The stricter requirement leads to better designs in cases where it is important to reduce the number of message delays.

To summarize, ZCP requires that non-conflicting transactions (1) do not access overlapping memory regions on a single node, and (2) do not require cross-replica coordination. A system that satisfies both requirements will be both multicore-scalable and

replica-scalable.

ZCP is inspired by rules about when multicore scalability is achievable on a single node. Beyond DAP itself, the Scalable Commutativity Rule [38] takes the DAP principle a step further by saying that an interface will have a multicore-scalable implementation if its operations commute; it then applies that rule to the design of operating system calls. ZCP extends this approach to the replicated systems context.

3.2. MEERKAT APPROACH

ENFORCING ZCP is a tall order for any distributed system. Meerkat uses the following design techniques to ensure zero coordination.

Replicate transactions, not arbitrary operations. State machine replication, used by popular replication protocols (e.g., Paxos [31], VR [32, 33] and Raft [34]), is a poor match for multicore scalability. Not only does it require determinism, which is impossible to enforce on multicore replicas without coordination, but it also requires a single ordering of operations, typically using a shared log.

Meerkat takes a different approach and directly replicates transactions. This design ensures that only conflicting transactions require coordination, maintaining ZCP. While other protocols use this approach to minimize coordination (e.g., Generalized Paxos [39], EPaxos [40]), they are more general in the way that they detect dependencies between operations and are not designed to minimize cross-core coordination.

Use timestamp-ordered optimistic concurrency control for parallel concurrency control checks. Without a single order of operations among replicas, Meerkat must use a different approach to detect conflicts in transaction ordering. Meerkat uses timestamp ordering and an optimistic concurrency control mechanism to allow conflict detection to happen in parallel. Timestamps directly order transactions, and replicas and cores can independently check for conflicts using only the timestamp without the need for coordination mechanisms. This approach has been used by multicore-scalable single-node transaction protocols [20–22]; Meerkat extends it to work with a decentralized replication protocol.

Use client-provided timestamps based on loosely synchronized clocks. An obvious challenge for timestamp-ordering is how to efficiently select a timestamp for each transaction. Timestamp selection can be both a multicore bottleneck (contention on the

next-assigned timestamp) and a source of coordination between replicas. Meerkat leverages loosely synchronized clocks as a way to avoid this coordination: clients select and propose a timestamp for a transaction, and replicas determine whether they are able to execute the transaction *at that timestamp* without conflicts. Variants of this approach have been used in a variety of recent systems [5, 10, 41]. Importantly, Meerkat does not require clock synchronization for correctness (unlike Spanner [10], for example) but only for performance.

Versioned backing storage. Versioned storage further eliminates the need to coordinate updates to the same key. Timestamp-based concurrency control requires versioned storage, so Meerkat can execute transactions on different versions of the same key out of order: it can execute a read operation at an earlier timestamp without conflicting with a later write, or process certain writes under the Thomas write rule [8].

3.3. MEERKAT OVERVIEW

IN this section, we describe the design of our new fault-tolerant, multicore transactional storage system dubbed Meerkat. Meerkat provides serializable transactions, replicating operations across multiple commodity servers for fault tolerance. Meerkat follows the Zero-Coordination Principle in its design, combining both a parallelism-friendly storage implementation and a new transaction-oriented replication protocol. Like prior single-node systems, it follows DAP to provide multicore scalability by avoiding cross-core coordination for non-conflicting transactions. Unlike these systems, it also avoids cross-replica coordination for these transactions, providing replica scalability and achieving improved performance and liveness compared to existing replicated storage systems.

Before we detail the Meerkat transaction processing protocol in the next section, we give an overview of the system model and key data structures.

3.3.1. SYSTEM MODEL AND ARCHITECTURE

We assume a standard asynchronous model of a distributed system. Formally, the system consists of $n = 2f + 1$ multicore replica servers and an unspecified number of client machines, where f is the number of replica failures that the system can tolerate. Replicas and clients communicate through an asynchronous network that may arbitrarily delay, drop, or re-order messages. Replicas can fail only by crashing and can recover later.

Meerkat guarantees one-copy serializability for all transactions: from each client's perspective, the results are equivalent to a serial execution of the transactions on a single system. Like all replicated systems, Meerkat cannot ensure liveness during arbitrary faults; it makes progress as long as fewer than half of the replicas are crashed or recovering, and as long as messages that are repeatedly resent are received in some bounded (but unknown) time.

Each transaction is managed by a distinct *Meerkat transaction coordinator*, which typically runs on the client machines (usually application servers), and each replica runs a local instance of the *Meerkat multicore transactional database*. A Meerkat multicore transactional database instance is a three-layered system consisting of a versioned storage layer, a concurrency control layer, and a replication layer. Every instance can process transactions independently of other instances, perform recovery from various failure scenarios, and synchronize with the other instances to ensure consistency.

3.3.2. MEERKAT DATA STRUCTURES

	TID	ReadSet	WriteSet	Status	Timestamp
Core 1	455	<a,3>,<b,9>	a,b	COMMITTED	10
	630	<a,10>	a	VALIDATED-OK	11
Core 2	121	<a,3>,<c,5>	c	VALIDATED-ABORT	
	845	<a,10>	a	ABORTED	

Figure 3.2: An example trecord, partitioned on transaction id among cores. Every Meerkat multicore database manages its own trecord.

Each replica (i.e., Meerkat multicore transactional database instance) runs an algorithm built around two main data structures: the *vstore* and the *trecord*. These structures are organized to preserve ZCP: all state is either partitioned per-key (as in the *vstore*) or purely transaction-local and hence only accessed from a single core (the *trecord*). The result is that cross-core coordination is only necessary between transactions that access the same data.

vstore. The *vstore*, shared among all cores at the replica, implements the versioned storage layer. The vstore can be implemented either as a concurrent hash table (to support efficient gets and puts), or a concurrent tree (to support efficient indexing and range queries). Our implementation uses a hash table; other research has developed suitable multicore-friendly concurrent tree structures [42].

The data stored in the *vstore* is augmented with a per-key version number, or write timestamp, wts, which is the timestamp of the transaction that most recently wrote the

value of key, and a read timestamp, rts, which is the timestamp of the transaction that most recently read the value of key. (Since Meerkat serializes transactions in timestamp order, as we show later, wts and rts essentially track the largest timestamps of transactions that wrote and, respectively, read the value of key.) Additionally, each key maintains two lists of all pending transactions that accessed it. $readers[key]$ stores the timestamps of all uncommitted transactions that read key and were successfully validated. Likewise, $writers[key]$ stores the timestamps of all pending transactions that wrote key. These are used in Meerkat's validation protocol.

trecord. As shown in Figure 3.2, the *trecord* maintains a set of transaction records for recovery and synchronization. A transaction record contains the following fields: a unique transaction id (*tid*), the transaction's read and write sets constructed during the execution phase (*ReadSet* and *WriteSet*), a (proposed) commit timestamp (*Timestamp*) chosen after the transaction is validated, and the status of the transaction (*Status*). A transaction record also contains two additional fields, *View* and *AcceptView*, that are used to recover from a failure of the transaction's coordinator; these are not illustrated in Figure 3.2. We discuss their use in Section 3.4.3.

To avoid unnecessary synchronization and preserve DAP, the trecord is horizontally partitioned among cores by transaction id. That is, each core operates on its own local trecord partition that contains a subset of the transactions. Because synchronous replication inherently requires a multi-round algorithm, a replica is required to process multiple messages for the same transaction, one in each round. We use a mechanism based on the NIC's forwarding capabilities to efficiently route transactions to the correct core. Naively routing transactions to cores can lead to spurious interrupts and unnecessary cross-core communication.

3.4. MEERKAT TRANSACTION PROTOCOL

WE begin with an overview of Meerkat's transaction execution, then follow with the protocol in the absence of failures. Finally, we describe failure handling and give a short proof.

3.4.1. PROTOCOL OVERVIEW

Meerkat uses an optimistic protocol for executing and replicating transactions. Like traditional optimistic concurrency control protocols [43], Meerkat's transaction protocol

follows a three-phase execute-validate-write protocol. Meerkat integrates its replica coordination protocol with the validation phase to ensure that transactions are executed consistently across replicas. Importantly, this is the only phase that requires coordination, and on its fast path (when there are no conflicts between transactions), it is able to execute without coordination between replicas.

Phase 1: Execute. During the execute phase, or read phase, clients read versioned data from any replica. They buffer any write operations locally; these are not installed until they are ready to commit and have been validated.

Phase 2: Validate. Once the client completes executing the transaction, the transaction enters the validation phase. This is a combined OCC-style validation, ensuring that the transaction commits only if no conflicting transactions have occurred, and a replica coordination protocol, ensuring that each replica has consistent state. In this phase:

1. A transaction coordinator (located either on a client or replica) selects a proposed timestamp for the transaction and sends a VALIDATE request to all replicas.
2. Each replica independently performs OCC validation, checking whether they have committed any other transactions that would conflict with the specified timestamp, and returns either OK or ABORT.
   (a) If a supermajority ($> \frac{3}{4}$) of the replicas send matching responses, the transaction commits or aborts under the fast path; the coordinator sends a COMMIT or ABORT message.
   (b) Otherwise, an additional round of coordination (the slow path) is needed to ensure consistency.

Phase 3: Write. Once a transaction has been committed, replicas update their versioned storage with the transaction's writes.

3.4.2. MEERKAT TRANSACTION PROCESSING

Meerkat's transaction protocol includes subprotocols for each of its three stages of execution. We first describe these three subprotocols assuming the data is not partitioned across multiple servers. We then discuss Meerkat support for distributed transactions.

3.4.2.1. EXECUTION PHASE

The execution phase is orchestrated by a *Meerkat transaction coordinator*. During the execution phase, the transaction coordinator sends every read to an arbitrary replica.

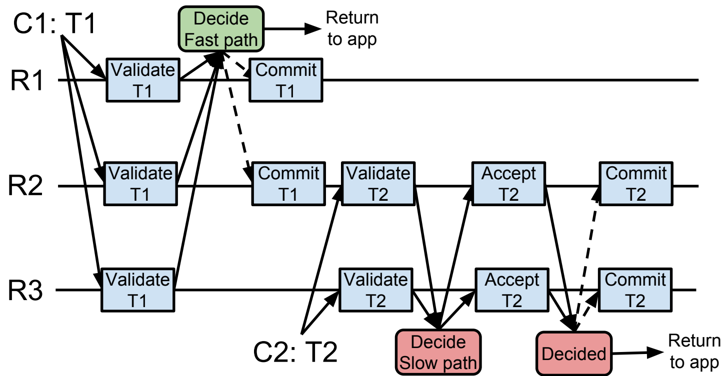


Figure 3.3: The commit protocol in normal operation mode.

The replica performs the read and responds with the read value and version. These versioned values are buffered by the transaction coordinator into a *ReadSet*. Similarly, the transaction coordinator buffers every write into a *WriteSet*. Writes are not sent to any replica.

3.4.2.2. VALIDATION PHASE

The execution phase is followed by a commit protocol that performs the validation phase, and, at the same time, replicates the outcome of a transaction. The protocol combines aspects of atomic commitment and consensus protocols. Like atomic commitment protocols, such as two-phase commit, the protocol performs decentralized transaction validation (i.e., each replica is seen as a distinct participant able to validate transactions independently). It uses consensus to allow backup coordinators (in case of coordinator failures) to eventually reach a unique decision to either commit or abort every validated transaction.

A high-level view of the commit protocol and its communication pattern in the normal operation mode is illustrated in Figure 3.3. The two illustrated coordinators, *C1* and *C2*, try to commit transactions *T1* and *T2*, respectively, at roughly the same time. After the coordinators receive validation replies from the replicas, they can either decide and commit the transaction on the fast path, if enough replicas already agree on the outcome of the transaction, or must take the slow path (e.g., coordinator *C2* did not receive enough validation replies), on which they must get the replicas to agree on the decided outcome.

Throughout this protocol, we assume that all messages concerning a particular transaction are always processed by the same core on any replica. This core affinity allows the transaction state to be partitioned across different cores in the *trecord*, preventing spu-

Algorithm 1 Meerkat validation checks

```
1:  procedure VALIDATE(txn, ts)
2:      ▷ Validate the read set
3:      for r ∈ txn.readSet do
4:          lock(r.key)
5:          e ← vstore[r.key]
6:          if e.wts > r.wts or ts > MIN(e.writers) then
7:              unlock(r.key)
8:              go to abort
9:          end if
10:         e.readers.add(ts)
11:         unlock(r.key)
12:     end for

13:     ▷ Validate the write set
14:     for w ∈ writeSet do
15:         lock(w.key)
16:         e ← vstore[w.key]
17:         if ts < e.rts or ts < MAX(e.readers) then
18:             unlock(w.key)
19:             go to abort
20:         end if
21:         e.writers.add(ts)
22:         unlock(w.key)
23:     end for

24:     return VALIDATED-OK

25: abort:
26:     cleanup_readers_writers(txn)
27:     return VALIDATED-ABORT
28: end procedure
```

rious synchronization across different cores. In practice, the way we achieve this is to have the coordinator select a *coreid* for the replicas to use, and select a UDP port for communication based on the *coreid* to ensure that Receive-Side Scaling NICs deliver messages to the same core.

The commit protocol proceeds as follows:

1. The coordinator first selects a core, *coreid*, to process the transaction, a proposed timestamp *ts* for the transaction, and a unique transaction id *tid*. The proposed timestamp *ts* indicates the proposed serialization point of the transaction. To avoid

the need for coordination to select the next available timestamp, the coordinator instead proposes a timestamp using its local clock: ts is a tuple of the client's local time and the client's unique id. tid is a tuple of a monotonically increasing sequence number local to the client and the client's unique id. By including the client's unique id, we ensure that both ts and tid are globally unique.

The transaction coordinator then sends $\langle$VALIDATE, $txn, ts\rangle$ to all the replicas,[1] where txn contains the tid as well as the read and write sets of the transaction.

2. Each replica creates a new entry in its core-local *trecord* partition and validates the transaction using the OCC checks illustrated in Algorithm 1.

   The OCC checks begin by validating the transaction's reads. The condition $e.wts > r.wts$ checks that the read has read the latest committed version. The condition $ts > MIN(e.writers)$ checks that even if all pending transactions were to commit, the read would still have read the latest committed version as of ts. If either of these conditions does not hold, the transaction is aborted. Otherwise, the transaction's timestamp is added to $readers$.

   The replica core then validates the transaction's writes. The conditions $ts < e.rts$ and $ts < MAX(e.readers)$ check that a write will not interpose itself between a pending read or committed read and the version read by that read. Again, if either condition does not hold, the transaction is aborted. Otherwise, the transaction's timestamp is added to $writers$.

   We designed the parallel OCC checks to have small atomic regions at the cost of precision (i.e., certain valid serializable histories may be rejected). Further optimizations may be possible. For example, in certain abort cases it may be possible to assign a different commit timestamp and still commit the transaction, as in TAPIR [5]. We have not yet explored these.

   Ultimately, the replica core replies to the coordinator with $\langle$VALIDATE-REPLY, $tid, status\rangle$ where $status$ is either `VALIDATED-OK` or `VALIDATED-ABORT`. If aborted, any changes to the $readers$ and $writers$ set are backed out.

3. If the coordinator receives a supermajority consisting of $f + \lceil \frac{f}{2} \rceil + 1$ VALIDATE-REPLYs with matching $status$ (this is the **fast path** condition), it notifies the client that the transaction is complete. If $status$ is `VALIDATED-OK`, then the transaction is committed, and if $status$ is `VALIDATED-ABORT`, then the transaction is aborted. The coordinator then asynchronously sends $\langle$COMMIT, $tid, status\rangle$ to all replicas. Note that this

[1] Throughout the protocol, any messages that receive no reply are resent after a timeout. For simplicity, we do not describe this process for each message.

COMMIT message can be piggy-backed on the client's next message.

4. If the fast path condition is not met, the coordinator takes the **slow path**. It must receive VALIDATE-REPLYs from a majority ($f+1$) replicas, resending the VALIDATE message if necessary. The coordinator then sends a ⟨ACCEPT, tid, $status$⟩ request to replicas, where, if $f+1$ or more VALIDATE-REPLYs have $status$ = VALIDATED − OK, the *status* argument is ACCEPT − COMMIT (i.e., the coordinator proposes to commit the transaction), else the *status* argument is ACCEPT − ABORT.

   The ACCEPT request has the same role as the phase 2a message in Paxos – to ensure that a single decision is chosen, even when there are multiple proposers. As described so far, there is only one proposer, namely the transaction coordinator; however, the coordinator recovery protocol (Section 3.4.3.3) introduces *backup coordinators* and the synchronization protocol (Section 3.4.3.2) introduces *synchronization coordinators* that also act as proposers.
5. On receiving ACCEPT, the replica updates the status of the transaction to $status$ and replies to the coordinator with ⟨ACCEPT-REPLY, tid⟩.
6. Once the coordinator receives a majority ($f+1$) of ACCEPT-REPLYs, the transaction is completed. It notifies the client, and asynchronously sends ⟨COMMIT, tid, $status$⟩ to all replicas. As before, this COMMIT message can also be piggy-backed on the client's next message.

To keep the description of the commit protocol simple, we defer some details, mostly involving concurrent proposals, which do not occur in the failure-free case, to the failure handling section (Section 3.4.3).

3.4.2.3. WRITE PHASE

On receiving a ⟨COMMIT, tid, $status$⟩ message, the replica marks the transaction as committed by setting its status in the *trecord* entry to COMMITTED if $status$ is VALIDATED-OK, or ABORTED if $status$ is VALIDATED-ABORT. It then performs OCC's write phase: if $status$ is VALIDATED-OK, the replica updates the data items to their new values, and sets the version of each item to the commit timestamp. Regardless of whether the transaction commits or aborts, the replica simply cleans up $readers$ and $writers$ for tid.

3.4.2.4. DISTRIBUTED TRANSACTIONS

The protocol we described so far can easily be extended to support distributed transactions (when data is partitioned across servers) since it already includes aspects of atomic commitment protocols (i.e., decentralized validation of transactions). The transaction coordinator would just need to perform, in parallel, the validation phase in all partitions involved in the execution of the transaction.

3

3.4.3. Meerkat Failure Handling and Record Truncation

Thus far, we have not considered replica or network failures, and have assumed that transaction coordinators do not fail. Of course, Meerkat must remain resilient to such failures.

First, Meerkat runs a periodic *synchronization* protocol (Section 3.4.3.2) that: (1) completes all unfinished transactions, (2) allows replicas to bring themselves up-to-date, and (3) safely truncates *trecord*s. Although the synchronization protocol is enough to recover from transaction coordinator failures, since it is able to complete unfinished transactions, it might be costly to wait until the next synchronization in order to recover a pending transaction, since a validated transaction not committed for a long enough period has high chances to interfere with the validation of concurrent transactions. Therefore, Meerkat also supports a *transaction coordinator recovery* protocol (Section 3.4.3.3) that allows a backup coordinator to step in, without affecting any other transaction. In most cases, backup coordinators can safely complete an unfinished transaction, without relying on the synchronization protocol.

Second, Meerkat handles replica failure transparently, but for a recovering replica to rejoin the system, Meerkat must run a *replica recovery* protocol (Section 3.4.3.4) that guarantees that a replica recovers in a state at least as recent as it was before the crash.

We first give an overview of the three sub-protocols (Section 3.4.3.1); we introduce a few important concepts and describe the additional state required to implement them. We then give complete specifications of the three sub-protocols, and, finally, we describe the few changes required to the original commitment protocol to interoperate with the new sub-protocols (Section 3.4.3.5).

3.4.3.1. Overview

Eras, Epochs, and Views. Each replica traverses multiple time periods delimited by *Eras* and *Epochs*. During an Era, a *synchronization coordinator* periodically moves the system to the next Epoch: it decides which transactions belong to the Epoch that just ended and commits all uncommitted transactions in that Epoch. A transaction can commit in at most one Epoch and all transactions committed in an Epoch happen after all transactions committed in all previous Epochs. Thus, Epochs can be seen as a sequence of checkpoints. If the current synchronization coordinator fails, the system moves to the next Era and elects a new synchronization coordinator.

If a transaction coordinator fails before informing replicas of its decision, the system must wait until a synchronization coordinator changes the Epoch and commits the transaction. To allow the system to try to commit the transaction as soon as possible, we

Table 3.1: State maintained by every Meerkat replica.

Variable	Meaning
$vstore$	Shared versioned key-value map.
$trecord[c][e]$	Set of transaction records maintained by core c for epoch e.
$.status$	The current status of the transaction stored in this $trecord$ entry. One of: `VALIDATED-OK`, `VALIDATED-ABORT`, `COMMITTED`, `ABORTED`.
$.view$	Latest view number this replica switched to. The view number identifies a unique proposer for the status of the transaction, as the $(view \bmod n)^{th}$ replica.
$.acceptView$	Latest view this replica accepted a status in for the transaction stored in this $trecord$ entry.
$.acceptStatus$	The latest status this replica accepted for the transaction stored in this $trecord$ entry, from the proposer identified as $(acceptView \bmod n)^{th}$ replica. One of: `COMMIT`, `ABORT`.
$ctable[c]$	For each client, the latest sequence number used in tid of a committed transaction.
$status$	The current status the replica is in. One of: *Normal*, *EpochChanging*, *Recovering*.
$epoch$	Identifies the epoch in which the replica last operated in.
era	Identifies the latest synchronization coordinator the replica is aware of, as the $(era \bmod n)^{th}$ replica.
$acceptEra$	Identifies the latest synchronization coordinator this replica accepted an epoch proposal from, as the $(acceptEra \bmod n)^{th}$ replica.
$acceptTrecord$	The latest $trecord$ proposal accepted by this replica for the current epoch, $epoch$, from the proposer identified as $(acceptEra \bmod n)^{th}$ replica. It is a set of $trecord$s, where $acceptTrecord[c]$ refers to core c's partition

use *Views*. During an Epoch a transaction may traverse multiple Views. If the transaction coordinator fails to commit the transaction in the initial View, the system will move to the next View for this transaction and elect a new *backup transaction coordinator* that will try to commit the transaction in this View.

When a replica recovers, it first forces the system to move to the next Era, and then waits until the next Epoch. This and the changes added to the original commitment protocol ensure that the replica recovers all transactions for which it has participated in the commitment protocol and that ended up being committed.

Additional protocol state. Table 3.1 gives a description of all the state variables used by the sub-protocols. Eras and Epochs are identified by two natural numbers, which replicas maintain in variables *era* and *epoch*, shared among all cores. Each replica starts in era 0 and epoch 0. As the system advances through eras and epochs, the era and epoch numbers only increase. Each replica core now maintains separate *trecords* for Epochs. Current Views are also identified by a natural number, maintained per transac-

tion, as a field, *view*, in the respective $trecord$ entry. Here, replicas also maintain the latest view number in which a coordinator proposed an outcome for the transaction and was accepted by the replica, *acceptView*. Replicas also maintain, in a per-core variable, the latest era number for which a coordinator proposed a $trecord$ and was accepted by the replica, *acceptEra*. Purging *trecord*s is necessary for practical reasons, but losing all track about committed transactions might violate correctness. Therefore, each core also maintains a client table, *ctable*, which saves the latest transaction sequence number of committed transactions used by clients, essentially a summary of the transactions that had already been committed. We assume transaction coordinators issue transactions with increasing sequence number; if a transaction coordinator recovers, we assume it will start using a bigger sequence number than all previous ones it used (e.g., by splitting the $tid.seq$ number into a $tid.seq.epoch$ and $tid.seq.ct$, and incrementing on disk the $tid.seq.epoch$ on every recovery) or it will start using a different client id.

3.4.3.2. SYNCHRONIZATION AND TRUNCATION (EPOCH CHANGE AND ERA CHANGE)

Due to network failures, Meerkat's replicas can lag behind. Unlike many other replication protocols, meerkat does not use sequencing to totally order the transactions. Therefore, replicas do not know if they lag behind. Reading from a replica that lags behind for a very long time, without knowing, can cause many transactions to abort. At the same time, a practical implementation should have a way to safely truncate the *trecord*s.

We solve these issues with an *epoch change* protocol that brings the system to a consistent state, allows replicas to bring themselves up-to-date, and allows for safe truncation of *trecord*s. The epoch change protocol is driven by a *synchronization coordinator*, which polls the replicas to determine the state of ongoing transactions. The synchronization coordinator is in charge of deciding the outcome of all ongoing transactions. After a successful synchronization, at least a majority of replicas will have a consistent *trecord*, which can be safely purged after the updates have been applied on the *vstore*. However, the trecords from multiple epochs can be kept for longer to help replicas that lag behind bring themselves up-to-date faster (i.e., without requiring to copy the entire $vstore$).

The synchronization coordinator first sends an $\langle$EPOCH-CHANGE, $era, epoch\rangle$ request to all replicas (meaning the synchronization coordinator want to close the current epoch, $epoch$, and start the next one, $epoch+1$). The synchronization coordinator does not initiate an epoch change from epoch e unless the previous epoch change, from $e-1$ to e, has been completed. Unlike transactions, epochs are totally ordered. On receiving an epoch change request, every replica in era stops processing transactions until the epoch change completes. If a replica receives an epoch change request to an epoch that

Algorithm 2 Meerkat re-validation checks

```
1:  procedure RE-VALIDATE(txn, ts, trecord)
2:      ▷ Validate against all the committed and accepted transactions
3:      ▷ Validate the read set
4:      for r ∈ txn.rSet do
5:          if ∃t ∈ trecord : (t.Status is COMMITTED) ∧ (t.Timestamp < ts) ∧ (∃w ∈
        t.wSet : w.key = r.key ∧ r.wts < t.Timestamp) then
6:              return VALIDATED-ABORT
7:          end if
8:      end for

9:      ▷ Validate the write set
10:     for w ∈ txn.wSet do
11:         if ∃t ∈ trecord : (t.Status is COMMITTED) ∧ (t.Timestamp > ts) ∧ (∃r ∈ t.rSet :
        r.key = w.key ∧ r.wts < ts) then
12:             return VALIDATED-ABORT
13:         end if
14:     end for
15:     return VALIDATED-OK
16: end procedure
```

is not the immediately next one in sequence (i.e., if e does not match its local $epoch$), it first brings itself up-to-date with an *epoch upgrade* protocol which involves copying from other replicas the trecords for all previous epochs that it missed after its current one. Replicas then acknowledge the new epoch and respond to the synchronization coordinator with their current *trecord*, aggregated across all cores. Upon receiving at least a majority ($f+1$) of replies, the synchronization coordinator creates a new trecord (preserving the per-core partitioning), in which it adds transactions such as to preserve any potential previous decisions.

We mask synchronization coordinator failures with era changes. When the synchronization coordinator fails, a backup synchronization coordinator initiates an era change procedure to become the new synchronization coordinator. An era change procedure gets at least $f+1$ replicas to reject messages from previous synchronization coordinators and allows the new synchronization coordinator to resume the sequence of epoch changes.

Epoch change protocol. Every message sent as part of the epoch change protocol contains era and $epoch$, identifiying the current $epoch$ and era in which the sending replica operates. If the $epoch$ and era in the message match the current $epoch$ and era at the

receiving replica, we say that the sending and receiving replicas are *contemporary*.

In the current specification every epoch change protocol message is sent and processed by a single pre-determined core (e.g., core 0), the coordinating core, which, in most cases, must communicate with all the other cores when processing the message (optimizations can be made where cores can send and process epoch change message in parallel, as long as certain atomicity invariants are preserved). The coordinating core communicates with the other cores by sending them messages (or events). These messages and the normal network messages received by the core are processed sequentially (e.g., no interleaving between message handlers).

On receiving any message, msg, sent as part of the epoch change protocol, from a non-contemporary sending replica, the receving replica takes the following steps:

1. If $msg.era < era$ or $msg.epoch < epoch$ then the replica ignores the message (and eventually informs the sender about the more recent era or epoch).
2. Else:
   (a) If $msg.era > era$ then the replica updates its current era number to $msg.era$ and discards all messages received from the older era it had ever stored.
   (b) If $msg.epoch > epoch$ then the replica performs an epoch update using the *epoch update* protocol, described later in this section.
   (c) It re-delivers the message (this time the message will be from a contemporary sending replica)

We proceed with the complete specification of the epoch change protocol (here we assume that all received messages are from contemporary senders; we covered the non-contemporary sender cases above):

1. If the synchronization coordinator, identified as the $(era \bmod n)^{\text{th}}$ replica, has $status = Normal$ and notices the need for an epoch change, i.e., when a local timer expired, sends an $\langle$EPOCH-CHANGE, $era, epoch\rangle$ request to all replicas.
2. On receiving an EPOCH-CHANGE message, msg, the replica sets its status to $EpochChanging$ and returns $\langle$EPOCH-CHANGE-REPLY, $era, epoch, trecord\rangle$, where $trecord$ is the set of $trecord[c][epoch]$ for all cores c. The $status = EpochChanging$ operation and reading $trecord[c][epoch]$ must appear to have been executed atomically by every core c. To achieve this, after setting $status$ to $EpochChanging$, the coordinating core sends a message to all the other cores and waits until they finish processing this message before sending the EPOCH-CHANGE-REPLY (the coordinating core does not process any new messages in the meantime). Each core c first finishes processing the current message, then processes the message from the coordinating core and returns $trecord[c][epoch]$.

The coordinating core the sends the EPOCH-CHANGE-REPLY message to the synchronization coordinator.

3. On receiving an EPOCH-CHANGE-REPLY message, msg, the synchronization coordinator takes the following steps:
   (a) It stores msg.
   (b) If it stored at least $f+1$ EPOCH-CHANGE-REPLYs, including one from itself, it either decides to wait for more replies or takes the following steps:
      i. It merges the received $trecord$s (including its own), such as to preserve the per-core partitioning, using the following rules:
         - It first adds all committed or aborted transactions (i.e., those for which at least one reply returns a $status$ of COMMITTED or ABORTED).
         - If any replica has accepted a decision for a transaction from the coordinator (or a backup coordinator, as described in Section 3.4.3.3), i.e., a reply contains a $trecord$ with a valid $acceptView$ for the transaction, it adopts the decision, stored in $acceptStatus$, with the latest view number, $max(acceptView)$, for that transaction, and adds it to the *trecord* with the corresponding status (if the respective $acceptStatus$ is COMMIT then the new $trecord$'s entry for that transaction will have $status$ COMMITTED, else ABORTED).
         - All remaining transactions for which at least a majority ($f+1$) of replies reported the same $status$ in their respective trecords (either VALIDATED-OK or VALIDATED-ABORT) are added to the *trecord* (with $status$ as COMMITTED or ABORTED).
         - All remaining transactions that might have committed on the fast path, i.e., those where there are at least $\lceil \frac{f}{2} \rceil + 1$ VALIDATE-OK replies, are re-validated using the OCC checks in Algorithm 2, on the newly created $trecord$, and added with the corresponding status.
         - Any other transactions are added with $status$ set to ABORTED (or re-validated in the same manner as the ones above, but with the new $trecord$).
      ii. It then sends an $\langle$ACCEPT-EPOCH$, era, epoch, trecord\rangle$ request to all replicas where $trecord$ is the newly formed transaction record.
4. On receiving ACCEPT-EPOCH message, msg, the replica sets its status to $EpochChanging$, sets $acceptEra$ to era and saves $msg.trecord$ in $acceptedTrecord$. It then sends an $\langle$ACCEPT-EPOCH-REPLY$, era, epoch\rangle$ to the synchronization coordinator.

5. On receiving ACCEPT-EPOCH-REPLY message, msg, the synchronization coordinator takes the following steps:
   (a) It stores msg.
   (b) If it received at least $f+1$ replies (including from itself), then it sends a $\langle$START-NEW-EPOCH, $era, epoch\rangle$ request to all replicas.
6. On receiving a START-NEW-EPOCH message, msg, the replica moves the transactions from $trecord[c][epoch]$ that are not in $msg.trecord[c]$ to $trecord[c][epoch+1]$, for all cores c and then replaces $trecord[c][epoch]$ with $msg.trecord[c]$ for all cores c; it then re-validates all the transactions in $trecord[c][epoch+1]$ against $msg.trecord$, for all cores c (more precisely, the coordinating core sends a message to all cores and each core does the transaction migration and re-validation on its own $trecord[c][epoch]$). Finally, the replica (the coordinating core) sets $epoch$ to $epoch+1$ and $status$ to $Normal$.

Epoch update protocol. We proceed with the complete specification of the epoch update protocol:

1. If a replica notices it is in a lower epoch, it sends a $\langle$EPOCH-TRANSFER, $era, epoch\rangle$ request to the synchronization coordinator or a replica it knows is in a newer epoch.
2. On receiving an EPOCH-TRANSFER message, msg, the synchronization coordinator returns $\langle$EPOCH-TRANSFER-REPLY, $era, epoch, trecord\rangle$, where $trecord$ is a set of $trecord[c][e]$s for all cores c and for all epochs e from $msg.epoch$ to $epoch-1$. If the synchronization coordinator already purged epoch records in that range, the replica will have to recover using the replica recovery protocol.
3. On receiving an EPOCH-TRANSFER-REPLY message, msg, the replica takes the following steps:
   (a) If $epoch \geq msg.epoch$, then the replica ignores the message.
   (b) If $msg.epoch < epoch$ then the replica moves the transactions from $trecord[c][epoch]$ that are not in $msg.trecord[c]$ (where $msg.trecord[c]$ refers to the union of all sets $msg.trecord[c][e]$ for all epochs e recovered in $msg.trecord$) to $trecord[c][msg.epoch]$, for all cores c (in other words, each core discards the transactions from its current $trecord$ which have been completed in older epochs) and then replaces $trecord[c][e]$ with $msg.trecord[c][e]$ for all cores c and all epochs e recovered from msg; it then re-validates all the transactions it just moved to $trecord[c][msg.epoch]$ against $msg.trecord$, for all cores c, using the re-validation checks from Algorithm 2, and updates $vstore$ by applying the

transactions in $trecord[c][e]$ and $ctable[c]$ for all cores c and recoverd epochs e (more precisely, the coordinating core sends a message to all the cores, which process the message atomically and do the described migration, re-validation and write phase in parallel). Finally, the replica sets $epoch$ to $msg.epoch$ and allows the cores to resume their operation.

Era change protocol. The goal after an era change is to get a majority of replicas in the latest operational epoch and era. Again, as before, there is a designated coordinating core (e.g., core 0) that processes all era change messages, sequentially. The complete specification of the era change protocol follows:

1. A replica noticing the need for an era change – a local timer expired and the replica still cannot contact the synchronization coordinator in its era – increments era and resets the timer. If the replica is the synchronization coordinator in this new era, i.e., it is the $(era \bmod n)^{\text{th}}$ replica, then it sends $\langle$ERA-CHANGE, $era\rangle$ request to all replicas, else it sends $\langle$ERA-CHANGE-REPLY, $era, epoch, acceptEra\rangle$ to the synchronization coordinator, where era, $acceptEra$
2. On receiving an ERA-CHANGE message, msg, the replica takes the following steps:
   (a) If $msg.era < era$ then the replica ignores the request.
   (b) Else the replica updates era to $msg.era$ and returns $\langle$ERA-CHANGE-REPLY, $era, epoch, acceptEra\rangle$.
3. On receiving an ERA-CHANGE-REPLY message, msg, the synchronization coordinator takes the following steps:
   (a) If $msg.era < era$, then the synchronization coordinator ignores the message.
   (b) Else if $msg.era \geq era$, then the synchronization coordinator updates era to $msg.era$, if necessary, discarding any replies from the older era, and then stores msg.
   (c) If the synchronization coordinator stored at least $f+1$ ERA-CHANGE-REPLYS, it decides the latest epoch, e, to be the largest it learned of, i.e., $max(msg.epoch), \forall$ stored msg. If $epoch < e$, then the synchronization coordinator first proceeds with the epoch update protocol, bringing itself up to the latest epoch (if the update protocol fails, e.g., the replica in the latest $epoch$ crashes, then the syncrhonization coordinator removes the ERA-CHANGE-REPLY reply from that replica and resends the ERA-CHANGE requests to replicas from which it didn't get a reply). Then, to preserve any potential decisions from previous synchronization coordinators with respect to the closure of the current epoch, if it notices any valid $msg.acceptEra$, $\forall$

stored msg such that $msg = e$, then the synchronization coordinator transfers the $acceptTrecord$ over from the corresponding replica (if the transfer fails, e.g., the corresponding replica crashes, then the synchronization coordinator removes ERA-CHANGE-REPLY from that replica and resends the ERA-CHANGE requests to replicas from which it didn't get a reply) that returned the largest $acceptEra$ in epoch e, and takes steps 3(b)ii - 6 from the epoch change protocol. Finally, the synchronization coordinator sets the epoch change timer.

3.4.3.3. TRANSACTION COORDINATOR FAILURE AND RECOVERY (VIEW CHANGE)

The failure of a Meerkat transaction coordinator presents a more subtle problem. In addition to preventing a client from learning the outcome of a transaction, a coordinator failure can leave unfinished transactions on the replicas. These unfinished transactions may degrade performance by preventing other transactions from being successfully validated.

Meerkat addresses this by using a consensus-based coordinator recovery protocol. Meerkat's coordinator strategy follows the approach used in TAPIR [5, 44] and is an instance of Bernstein's cooperative termination protocol [1]. In this protocol, in addition to its normal coordinator, each transaction also has a set of $2f + 1$ *backup coordinators*. The backup coordinators, which are only invoked on coordinator failure, can be shared among all transactions; each replica can run a backup coordinator process, or they can be deployed on a separate cluster. In the event of a coordinator failure, a replica can initiate a coordinator change to activate a backup coordinator and complete (either commit or abort) the transaction.

To ensure that transaction outcomes are consistent even in the presence of backup coordinators, Meerkat uses a consensus protocol. The consensus algorithm guarantees that all (backup) coordinators eventually reach the same decision – to either commit or abort the transaction – even if they concurrently propose different transaction outcomes.

Like most consensus algorithms, the coordinator recovery protocol uses *views* to uniquely identify proposals. Unusually, each view is specific *to a given transaction* (i.e., tid). For each view, one coordinator acts as the proposer. Each transaction starts with $view = 0$, and in this view, the original transaction coordinator is the unique proposer. The unique proposer in $view > 0$ is the $(view \bmod n)^{\text{th}}$ replica.

To support this protocol, for each transaction, we include two additional fields in the transaction's trecord entry: (1) the current view number, $view$, initially 0; and (2) if this is the case, the view number in which a proposal was last accepted, *acceptView*. Note that this is the same information maintained by Paxos to solve one instance of consensus.

When a replica notices the potential failure of the coordinator of the current view, is starts a view change procedure, similar to Paxos' prepare phase, where a new backup coordinator is established (a majority of replicas agree to ignore proposals from previous coordinators, i.e., containing lower view numbers). After receiving a quorum of replies for the coordinator change request, the new coordinator analyzes the replies to determine a safe outcome for the transaction. This means that it selects any outcome that, in order of priority, has (1) been completed (COMMITTED or ABORTED) at any replica, (2) been proposed by a prior coordinator and accepted by at least one replica, or (3) been VALIDATED-OK or VALIDATED-ABORT by a majority of replicas. It then attempts to complete the transaction with that outcome on the slow path, using a procedure similar to Paxos' accept phase.

3

The complete specification of the view change protocol (including several optimizations to reduce the number of messages sent, by exploting the fact that the backup coordinator is now also a replica and might have already learned the outcome of the transaction):

1. A replica in the Normal $status$ noticing that a transaction, tid, hasn't been completed in due time – a local timer expired and $trecord[c][e][tid].status$ is not COMMITTED or ABORTED, where c is the core that processed the transaction and e is the current $epoch$ – increments $trecord[c][e][tid].view$ and resets the timer. If the replica is the backup coordinator in this new view, i.e., it is the $(trecord[c][e][tid].view \bmod n)^{th}$ replica, then it sends $\langle$VIEW-CHANGE$, tid, view\rangle$ to all replicas, else it sends $\langle$VIEW-CHANGE-REPLY$, tid, era, view, acceptView, status\rangle$ to the backup coordinator, where $view$, $acceptView$ and $status$ are the current values in the corresponding entry in the current $trecord$ and era is the current value of era.
2. On receiving a VIEW-CHANGE message, msg, the replica takes the following steps. Let c be the core processing msg:
   (a) If $status$ is not $Normal$, the replica ignores or postpones the processing of the request.
   (b) If $msg.tid.seq \leq ctable[c][msg.tid.clientid]$, then the replica sends $\langle$VIEW-CHANGE-REPLY$, tid, status\rangle$ to the backup coordinator, where $status$ is $trecord[c][e][tid].status$ if such an entry is found for an older epoch, e, or else PURGED.
   (c) Else if $msg.tid.seq > ctable[c][msg.tid.clientid]$, then, if there is not already an entry $trecord[c][e][msg.tid]$, where e is the current epoch number, $epoch$, then the replica validates the transaction as

before (after learning the read and write sets), creating a new entry in the trecord, t, and sets $t.view$ to $msg.view$. If there is already an entry $t = trecord[c][e][tid]$ then, if $msg.view \geq t.view$ then the replica updates $t.view$ to $msg.view$. The replica then returns $\langle$VIEW-CHANGE-REPLY, $tid, era, view, acceptView, status\rangle$, where $view$, $acceptView$ and $status$ are the current values stored in t and era is the current value of era.

3
3. On receiving a VIEW-CHANGE-REPLY message, msg, the backup coordinator, which is now also a replica itself, takes the following steps:
   (a) If $status$ is not $Normal$, the coordinator ignores or postpones the request.
   (b) If $msg.tid.seq \leq ctable[c][msg.tid.clientid]$, then if $msg.status$ is not `COMMITED`, `ABORTED` or `PURGED`, then the coordinator sends $\langle$COMMIT, $tid, status\rangle$ to the replica that sent msg, where $status$ is $trecord[c][e][tid].status$ if such an entry is found for an older epoch, e, or else `PURGED`.
   (c) Else if $msg.tid.seq > ctable[c][msg.tid.clientid]$, then the backup coordinator takes the following steps:
      i. If $msg.status$ is `PURGED`, then the backup coordinator is in an older epoch and needs to bring itself up-to-date, following the epoch transfer procedure described in the next section.
      ii. Else if $msg.status$ is either `COMMITTED` or `ABORTED`, the backup coordinator commits the transaction accordingly (if $msg.status$ is `COMMITTED` and the transaction has not been validated locally, then the backup coordinator first retrieves the read and write sets from other replicas), updates the $trecord$ entry accordingly, sends $\langle$COMMIT, $tid, status\rangle$ to replicas for which it stored the VIEW-CHANGE-REPLYs and then it removes these replies.
      iii. Else if there is an entry $t = trecord[c][e][msg.tid]$, where e is the current epoch number, $epoch$, then if $t.status$ is either `COMMITTED` or `ABORTED`, then the coordinator sends $\langle$COMMIT, $tid, status\rangle$, where $status = t.status$ to the replica that sent msg. Else if $t.status$ is not `COMMITTED` or `ABORTED`, then if $msg.view = t.view$, then the coordinator stores msg, else if $msg.view > t.view$ then the coordinator updates $t.view$ to $msg.view$ and removes any view change replies it had stored for older views, else it ignores the message.
      iv. Else if there is not already an entry $t = trecord[c][e][msg.tid]$, where e

is the current epoch number, $epoch$, then the coordinator validates the transaction as before (after learning the read and write sets from another replica), sets $t.view$ to $msg.view$, and stores msg.

4. If the backup coordinator stored at least f view change reply messages (at this point an entry $t = trecord[c][e][tid]$ is guaranteed to exist and $m.view = t.view$ for all those messages and no decision has been learned), it tries to commit the transaction:
   (a) If a supermajority of replicas (including the backup coordinator) return the same $status$ and era, then the backup coordinator takes the fast path and sends a $\langle$COMMIT$, tid, status\rangle$ to all replicas, including itself, where $status$ is accordingly set to `COMMITTED` or `ABORTED`.
   (b) Else if
5. If replicas accepted decisions from previous coordinators (at least one replica returns a *Status* of `ACCEPT-COMMIT` or `ACCEPT-ABORT`), the coordinator adopts the decision with the latest view number (*AcceptView*) and tries to commit it using the slow path protocol. It thus sends $\langle$ACCEPT$, tid, view, status, ts\rangle$ request to replicas. Once the coordinator receives a majority ($f+1$) of ACCEPT-REPLYs, the transaction is completed. It notifies the client, and asynchronously sends $\langle$COMMIT$, tid, status\rangle$ to all replicas.
6. If at least a majority ($f+1$) of replicas reported the same $Status$ in their respective trecords (either `VALIDATED-OK` or `VALIDATED-ABORT`), the coordinator tries to commit the transaction using the slow path protocol as before.
7. In the rare case where the majority cannot be formed as above (e.g., not enough replicas could be contacted), the transaction might still have committed, on the fast path, if at least $\lceil \frac{f}{2} \rceil + 1$ of replicas reported `VALIDATE-OK` $Status$ in their trecords. In this case the coordinator cannot safely decide the outcome for this transaction and must rely on the epoch change protocol described in the next section.
8. In all other cases the coordinator tries to abort the transaction using the slow path protocol.

3.4.3.4. REPLICA FAILURE AND RECOVERY

Meerkat uses a leaderless quorum protocol, so it is inherently resilient to failures of a minority of replicas. Provided that there are at least $f+1$ replicas still available to respond to requests, the system continues to make progress. Depending on the number of replicas in the system, failures may cause the number of available replicas to drop below the $f + \lceil \frac{f}{2} \rceil + 1$ needed for a supermajority, forcing the slow path on every transaction. Note,

however, that this still compares favorably to many commonly-used primary-backup protocols, which must stop processing transactions entirely after any replica failures, until the system is reconfigured.

We assume that a failed replica rejoins the system without its previous state. To recover correctly, we must ensure not only that the recovering replica learns the outcome of all committed transactions, but that the system state remains consistent for any partially-completed transactions the replica might have participated in a quorum in prior to crashing [45].

Given that Meerkat is an in-memory storage system, we wanted to minimize the writes to disk necessary to provide a correct recovery mechanism. For our current recovery protocol to be correct, we require each replica to write to disk the latest era it was operational in (its $status$ was $Normal$); this implies a write to disk is needed whenever the replica notices an era change. When a replica recovers, it sets its $status$ to *Recovering*, then it transfers the entire $vstore$ and the non-purged $trecord$s up to the latest known operational epoch. The recovering replica then forces an era change, to the $era+1$, where era is the value recovered from its disk. During this process it finds out the latest epoch the replicas agreed upon and then waits until the next epoch change. Once it observes the epoch change, it is safe to set its $status$ to $Normal$ (the epoch change makes sure all transactions from older era and epoch in whose commitment protocol the recovering replica participated, completed).

To eliminate completely the need for disk writes, the crash-consistent quorum protocol abstraction from [45] can be applied.

3.4.3.5. UPDATED COMMITMENT PROTOCOL

Here we describe the changes required to the original commitment protocol in order to correctly interoperate with the other sub-protocols:

1. The transaction coordinator sends a VALIDATE request to replicas as before.
2. On receiving a VALIDATE request, the replica takes the following steps. Let c be the core processing the VALIDATE request at the replica.
   (a) If $status$ is not $Normal$, the replica ignores or postpones the processing of the request.
   (b) If $tid.seq \leq ctable[c][tid.clientid]$, it means that this is a request for an already committed transaction, for which we might or might not have the outcome stored (as we detail later, the replica might have purged the $trecord$s from an older epoch that contained this information). In this case, the replica returns $\langle$VALIDATE-REPLY, $tid, status\rangle$, where $status$ is $trecord[c][e][tid].status$ if such an entry is found for an older epoch, e, or

else PURGED.

(c) Else if $tid.seq > ctable[c][tid.clientid]$, then, if there is not already an entry $trecord[c][e][tid]$, where e is the current epoch number, $epoch$, then the replica validates the transaction as before. The replica then returns $\langle$VALIDATE-REPLY, $tid, era, view, status\rangle$, where era and $view$ are the current values of era and $trecord[c][epoch][tid].view$, respectivelly, and $status$ is the current value of $trecord[c][epoch][tid].status$.

3. The transaction coordinator waits until it receives VALIDATE-REPLYs from at least a majority of replicas. It then takes the following steps:

   (a) If any of the replies return a $status$ of COMMITTED, ABORTED or PURGED, then the transaction has been committed by another coordinator, in which case the transaction coordinator returns $status$ to the application.

   (b) Else if a supermajority of replies return matching $status$ of either VALIDATED-OK or VALIDATED-ABORT and matching eras (a stronger fast path condition; but note that they need not return the same $view$), then the transaction coordinator takes the **fast path** and returns $status$ to the application. It then asynchronously sends $\langle$COMMIT, $tid, status\rangle$ to all replicas (where $status$ is either COMMIT or ABORT, respectively).

   (c) Else if there is no majority of replies that returned matching eras, then the transaction coordinator restarts the validation protocol with step (1) above (eventually after waiting some time). The re-sent VALIDATE messages are used to either learn the decision of the backup coordinator(s), in which case the protocol will stop at step (3) (a) above, or to re-validate the transaction on a replica that failed and recovered without any information about the transaction – a transaction is guaranteed to eventually be committed, assuming the failures eventually stop, only if it made it in the record of at least a majority of replicas *in the same era*.

   (d) Else if any of the replies returned a $view$ different than 0, it means another coordinator took ownership of the transaction. In this case, the transaction coordinator waits until it learns the outcome of the transaction as decided by a backup coordinator, e.g., by reading the transaction $status$ from replicas until COMMIT, ABORT or PURGED is returned. It then returns $status$ to the application.

   (e) Else the transaction coordinator tries to commit the transaction on the **slow path**. If at least a majority of VALIDATE-REPLYs have $status = \mathrm{VALIDATED-OK}$ and matching era numbers then the transaction coordinator proposes to

COMMIT the transaction and sends $\langle \text{ACCEPT}, tid, status, view \rangle$ request to replicas, where $status$ = ACCEPT – COMMIT, and $view = 0$. In all the remaining cases the transaction coordinator proposes to ABORT the transaction and sends $\langle \text{ACCEPT}, tid, status, view \rangle$ request to replicas, where $status$ = ACCEPT – ABORT, and $view = 0$.

4. On receiving an ACCEPT request, msg, a replica takes the following steps. Let c be the core processing the VALIDATE request at the replica.
   (a) If $status$ is not $Normal$, the replica ignores or postpones the processing of the request.
   (b) If $tid.seq \leq ctable[c][tid.clientid]$, then the replica returns $\langle \text{ACCEPT-REPLY}, tid, status \rangle$, where $status$ is $trecord[c][e][tid].status$ if such an entry is found for an older epoch, e, or else PURGED.
   (c) Else if $tid.seq > ctable[c][tid.clientid]$, then, if there is not already an entry $trecord[c][e][tid]$, where e is the current epoch number, $epoch$, then the replica creates the entry for tid, $status$ and $view$ as received in the ACCEPT request. If there is already an entry $t = trecord[c][e][tid]$ then, if $msg.view \geq t.view$ and $t.status$ is not $COMMITTED$ or $ABORTED$ then the replica updates $t.status$ to $msg.status$, $t.view$ to $msg.view$ and $t.acceptView$ to $msg.view$. The replica then returns $\langle \text{ACCEPT-REPLY}, tid, era, view, acceptView, status \rangle$, where era, $view$, and $acceptView$ are the current values of era, $trecord[c][epoch][tid].view$, and $trecord[c][epoch][tid].acceptView$, respectivelly, and $status$ is the current value of $trecord[c][epoch][tid].status$.

5. A transaction coordinator that took the slow path waits until it receives ACCEPT-REPLYs from at least a majority of replicas. It then takes the following steps:
   (a) If any of the replies return a $status$ of COMMITTED, ABORTED or PURGED, then the transaction has been committed by another coordinator, in which case the transaction coordinator returns $status$ to the application.
   (b) Else if any of the replies returned a $view$ different than 0, it means another coordinator took ownership of the transaction. In this case, the transaction coordinator waits until it learns the outcome of the transaction as decided by a backup coordinator, e.g., by reading the transaction $status$ from replicas until COMMIT, ABORT or PURGED is returned. It then returns $status$ to the application.
   (c) Else if the coordinator received at least a majority replies with the same era and $view = 0$ then it considers the transaction committed and returns

to the application with the respective $status$ and asynchronously sends ⟨COMMIT, tid, $status$⟩ to all replicas.

(d) Else, the transaction coordinator re-sends the ACCEPT messages and repeats the protocol from step (5) above (eventually after waiting some time).

3.4.4. CORRECTNESS

Meerkat guarantees serializability of transactions under all circumstances. We give a brief correctness proof sketch here.

Correctness during normal operation. Consider first the non-failure case. Although Meerkat does not guarantee that every replica executes each transaction – no quorum-based protocol can do so – nor even that *any* replica executes every transaction, its correctness stems from the fact that OCC checks can be performed pairwise [5]. Moreover, every successfully committed transaction must have been `VALIDATED-OK` on a majority ($f+1$) of replicas. For purposes of contradiction, suppose that there are two conflicting transactions that have both successfully committed. Quorum intersection means that there exists at least one replica that must have `VALIDATED-OK` both of the two transactions. However, because the two transactions conflict, whichever one arrived later at the replica must have returned `VALIDATED-ABORT` instead. Thus, no such pair of transactions can exist.

Correctness during replica failure. As noted above, replica failure by itself requires no special handling; it is replica *recovery* that poses challenges. The correctness property for the epoch change protocol is twofold: (1) any client-visible results from previous epochs, either commits or aborts, are reflected as `COMMITTED` or `ABORTED` outcomes in the *trecord* adopted by any replica at the start of the new epoch, (2) no further transactions commit in the old epoch. Property (2) is readily satisfied, as the new epoch only begins once $f+1$ replicas have acknowledged an EPOCH-CHANGE message, and they subsequently process no new transactions in the old epoch; thus, no further transactions can achieve a quorum.

With respect to property (1), consider first a transaction that commits or aborts on the slow path. In this case, $f+1$ replicas must have processed the ACCEPT message from the coordinator. At least one of these replicas will participate in the epoch change, and so the procedure in Section 3.4.3.4 will add it to the *trecord* in the appropriate state. Now consider a transaction that committed on the fast path. At least $f+\lceil\frac{f}{2}\rceil+1$ replica must

have returned `VALIDATED-OK` in the validate phase, and at least $\lceil \frac{f}{2} \rceil + 1$ must have participated in the epoch change. This transaction too will be added to the *trecord*, per the algorithm, *unless* a conflicting transaction has already been committed; however, since the original transaction previously committed successfully, no such transaction can exist.

3.5. EVALUATION

OUR evaluation demonstrates the impact of ZCP on multicore scalability. We break down the cost of cross-core coordination and cross-replica coordination on two workloads with both long and short transactions. We also measure the trade-off between multicore scalability and performance under high contention. Our experiments show the following:

1. Cross-core coordination has a significant impact on multicore scalability regardless of transaction length. Eliminating cross-core coordination improves throughput by 5–7× for up to 80 server threads.
2. Cross-replica coordination depends on transaction length. Eliminating cross-replica coordination can provide a performance improvement ranging from 3% to almost 2× for a three-replica system.
3. ZCP comes at a trade-off in supporting extremely high contention workloads. Using 64 server threads, Meerkat provides better performance on low-to-moderately skewed workloads (up to Zipf coefficients past 0.8), but performance suffers on very highly skewed workloads.

3.5.1. PROTOTYPE IMPLEMENTATION

In addition to our Meerkat prototype, we implemented three other systems to evaluate the relative impact of the two ZCP constraints. First, we implemented a classic, log-based, primary-backup replicated system that requires both cross-core and cross-replica coordination: the primary decides transaction ordering using a shared atomic counter and places each committed transaction into a shared log for replication. Replicas also share the log but read transactions and apply updates in parallel (concurrent log replay). Since the transactions are already ordered, there is no need for determinism at the replicas; however, log replay still requires cross-core coordination for access to the shared log. This mechanism is similar to many primary-backup multi-core databases, like KuaFu [46]. Unlike KuaFu, this variant does not need more synchronization, like

barriers, to deal with read inconsistencies (i.e., reads from backups that might be in an inconsistent state after applying updates out of order) since it relies on OCC validation checks at the primary for correctness. Therefore, we refer to this system as KuaFu++.

	Cross-Core Coordination	Cross-Replica Coordination
KuaFu++	Yes	Yes
TAPIR	Yes	No
Meerkat-PB	No	Yes
Meerkat	No	No

Table 3.2: An overview of our evaluation prototypes. To measure the impact of ZCP, we implemented systems with differing cross-core and cross-replica coordination.

Next, we implement a leader-less replicated system, designed to emulate TAPIR [5]. The replicas do not coordinate, but each replica uses a shared, cross-core transaction record. Based on the TAPIR protocol, clients issue transaction timestamps, so replicas can perform concurrency control checks and apply transaction updates in parallel.

Finally, we implement a primary-backup variant of Meerkat that satisfies DAP, which we label Meerkat-PB. It uses the same data structures and concurrency control mechanism as Meerkat; however, only the primary executes concurrency control checks, i.e., all clients submit their transactions with timestamps to the primary, and the primary decides which conflicting transactions will commit. Since transactions are timestamp-ordered and conflict-free, replicas can commit transactions in any order. To eliminate shared data structures, each backup core is matched to a primary core and processes only its transactions. Meerkat-PB lets us measure the impact of cross-replica coordination without cross-core coordination.

We believe that these four systems are representative for the class of distributed storage systems we target – fast systems that assume low to medium contention. Other approaches are possible, such as deterministic [47] or speculative [48] approaches. However, both of these types of approaches, unlike ours, require that transactions' read and write sets be known a priori. In general, they are most effective for high contention rates, rather than our target.

All of our prototype systems have three layers: (1) a transport layer for message delivery, (2) a storage layer for storing the data and scheduling the transactions, and (3) a replication layer for consistently replicating the data. All systems share the transport layer – ensuring that all systems have access to the same high-speed network library and

avoiding differences due to different approaches to serializing and deserializing wire formats. Meerkat and Meerkat-PB also share the storage layer. All replication layers use the same unordered *record* abstraction. Meerkat and Meerkat-PB use one record per core, while KuaFu++ and TAPIR share a single record per replica. To synchronize accesses to the shared record, these solutions use simple mutexes from the C++ standard library.

The shared transport layer is built on eRPC [19], a fast, reliable RPC library that bypasses the kernel. All storage layers provide the same semi-structured data model and use hash tables to store the key-value pairs. They also all implement interactive transactions, as opposed to stored procedures. Each key-value entry has its own fine-grained lock, and reader and writer lists for concurrency control checks. We use the `unordered_map` container for various data structures and `pthread` read-write locks to protect them for concurrent access.

3.5.2. EXPERIMENTAL SETUP

We use three replica servers in our experiments. Each server has two 20-core Intel Xeon Gold 6138 processors with 2 hyperthreads per core, supporting up to 80 server threads. In all experiments, we pin each server thread to a distinct logical CPU (i.e., hyperthread). Each core has private L1 and L2 caches (of 64 KB and 1024 KB, respectively) and shares the L3 cache (of 28 MB) with the other cores on its processor. The total DRAM size of 96 GB is evenly split between the two NUMA nodes. To eliminate uneven overheads (e.g., L3 contention, database loaded in one NUMA node), each experimental result was generated by an even number of threads, half of which were pinned on distinct cores of one NUMA node and half of which were pinned on distinct cores of the other NUMA node.

Each server machine has a Mellanox ConnectX-5 NIC, connected using a 40 Gbps link to an Arista 7050QX-32S switch. All machines run Ubuntu 18.04 with Linux kernel version 4.15. To mitigate the effects of frequency scaling, we set the scaling governors (power schemes for CPU) to *performance*, which maintains the CPUs running at 2 GHz, for all 80 logical CPUs (hyperthreads), and we always start all 80 server threads to keep all the cores at full utilization (each server thread polls in a loop on a NIC queue) to prevent the Intel Turbo Boost from unevenly boosting up the frequencies of the cores.

We use the flow steering mechanism of the Mellanox NICs to preserve DAP in the networking stack. Each server thread uses its own send and receive queue to which clients can steer packets based on a port number – the NIC will place packets in the corresponding queue managed by a single core, thus avoiding unnecessary cross-core

Transaction Type	# gets	# puts	workload %
Add User	1	3	5%
Follow/Unfollow	2	2	15%
Post Tweet	3	5	30%
Load Timeline	rand(1,10)	0	50%

Table 3.3: Description of the Retwis workload.

synchronization.

We run closed loop clients on ten 12-core machines, each a single NUMA node. Each client machine uses a Mellanox ConnectX-4 NIC, connected using a 40 Gbps link to the same switch as the server machines. The client clocks are synchronized with the Precision Time Protocol (PTP). When running the experiments, we used a 5-minute warm-up period to warm-up the caches and the CPUs. Each data point is the average of the results of 3 identical runs.

We use two benchmarks: (1) YCSB-T [27], a transactional version of Yahoo's popular YCSB [49] key-value benchmark and (2) Retwis [5], illustrated in Table 3.3, a benchmark designed to generate a Twitter-like transactional workload. We use keys and values of size 64 bytes. To illustrate the impact of contention on each system, we perform experiments using a varying Zipf distribution ranging from 0 (uniform, low contention) to 0.6 (medium contention) to >0.9 (highly contended).

We pre-allocate memory for metadata, such as the transaction records and locks, to avoid memory allocation overhead during the measurements. Before each run, we load the entire database into memory, with 1 million data items per core. By increasing the number of keys, we keep the contention level constant as we scale to increasing numbers of cores.

All systems use OCC-based concurrency control, which allows any replica to serve GETs – the OCC concurrency control checks will later establish if they can be ordered at a serializable timestamp. We thus uniformly load balance the clients across replicas and their cores for both GET and COMMIT requests.

Our experiments are focused on measuring throughput – more precisely, goodput, i.e., the number of transactions that successfully commit per second. It should be noted, however, that Meerkat does not sacrifice latency to achieve scalability. Indeed, it achieves low latency because the validation checks are cheap, with small atomic regions – comparable with the ones used by other systems – and the protocol saves one round trip compared to most state-of-the-art systems.

3.5.3. IMPACT OF ZCP ON YCSB-T BENCHMARK

Using the YCSB-T benchmark, we measure the impact of cross-core and cross-replica coordination using our four prototype systems. We use the transactional variant of the YCSB workload F, where transactions consist of a single read-modify-write operation. These transactions are relatively short with an even mix of read and write operations.

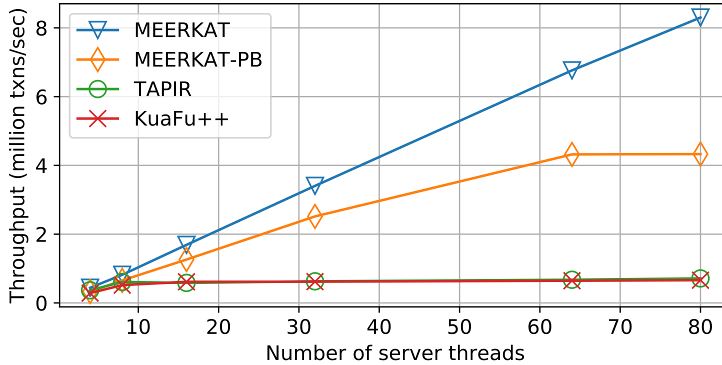


Figure 3.4: Peak throughputs comparison between Meerkat and the three other systems for uniform key access distribution for YCSB-T transactions containing one read-modify-write operation. While Meerkat is able to scale to 80 cores, the other systems all bottleneck at fewer cores due to violating ZCP.

Figure 3.4 shows the performance of each system. The KuaFu++ system has both cross-core and cross-replica coordination; as a result, it bottlenecks at 6 cores with 600,000 transactions per second. Eliminating cross-replica coordination only improves performance slightly: our TAPIR system scales to 8 cores and 800,000 transactions per second. However, cross-core contention prevents it from scaling further, because it still uses a shared record between server threads. This highlights an important point: despite the significant research that has gone into eliminating cross-replica coordination, cross-core coordination may pose a more significant bottleneck for many deployments, particularly using modern networks.

In contrast, eliminating cross-core coordination in our Meerkat-PB system increases throughput by 7x compared to KuaFu++. Despite the need for cross-replica coordination, the fast network communication between replicas using our eRPC transport lets Meerkat-PB scale to 64 server threads.

Finally, eliminating cross-replica and cross-core coordination in Meerkat improves throughput by 12x compared to our baseline KuaFu++ system. Meerkat is able to continue scaling to 80 threads and 8.3 million transactions per second, with the elimination

of cross-core and cross-replica coordination contribution in roughly equal parts to its performance improvements.

3.5.4. IMPACT OF ZCP ON RETWIS BENCHMARK

We perform the same experiment with the Retwis [5, 50] benchmark. Compared to YCSB-T, Retwis offers longer, more complex transactions, a more read-heavy workload and a wider range of transaction types. Figure 3.5 shows the results for all four systems.

Due to longer transactions, the total transaction throughput is lower for all systems. As a result, violating ZCP has less impact on the systems with more coordination, and even the non-ZCP systems are able to scale to more cores. TAPIR and KuaFu++ are both able to scale to 32 cores because there is more work for the cores to do during the execution period where there is less coordination. However, they still are not able to process more than 600,000-700,000 transactions per second.

Since cross-replica coordination only happens during transaction commit, its impact on performance is reduced with longer transactions. In particular, the majority of the execution phase for this benchmark consists of read operations, which can be executed on any replica. TAPIR does not improve much on KuaFu++ by eliminating cross-replica coordination and Meerkat-PB scales almost as well as Meerkat. As a result, with longer transactions, the effects of cross-core coordination on multicore scalability increase while the effects of cross-replica coordination decrease. Meerkat still scales the best, achieving 2.7 million transactions per second on 80 server threads.

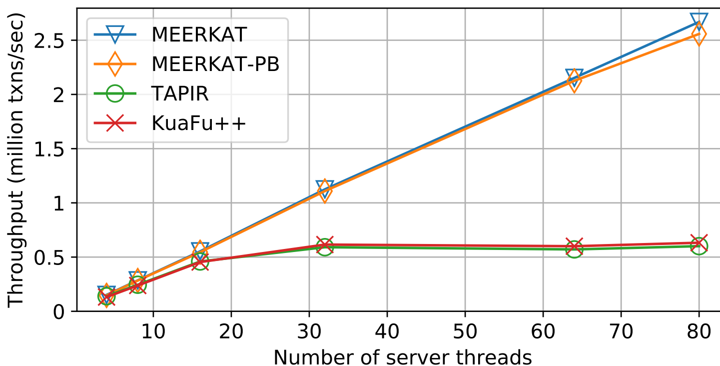
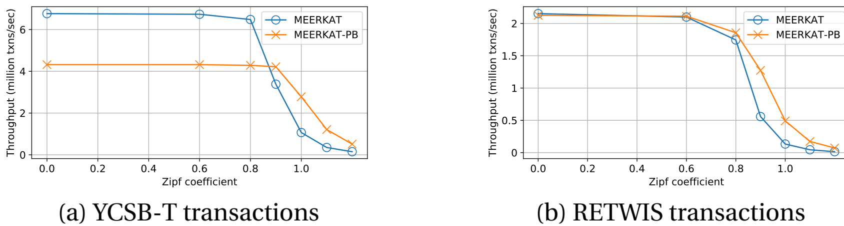


Figure 3.5: Peak throughputs comparison between Meerkat and the three other systems for uniform key access distribution for Retwis transactions. The comparison systems are able to scale better with coordination due to Retwis's longer transactions and read-heavy workload. However, none of the systems quite scale linearly to 80 cores except Meerkat; the non-ZCP systems continue to be limited by their additional coordination.

3.5.5. ZCP AND CONTENTION

In this section, we evaluate the trade-off between ZCP and performance under high contention workloads. Meerkat uses a highly optimistic concurrency control mechanism to avoid cross-core and cross-replica coordination. Meerkat decentralizes OCC checks at all replicas, while Meerkat-PB centralizes them at the primary. As a result, Meerkat is more likely to abort transactions at higher contention rates because replicas cannot agree. Thus, Meerkat trades-off performance under high contention for better multicore scalability.

To show this trade-off, we vary the Zipf coefficient of both the YCSB-T and Retwis benchmarks. We fix the number of server threads at 64, where Meerkat-PB still scales on both workloads and compare their performance across the range of Zipf coefficients.

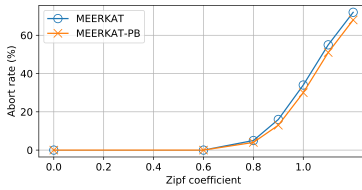


(a) YCSB-T transactions (b) RETWIS transactions

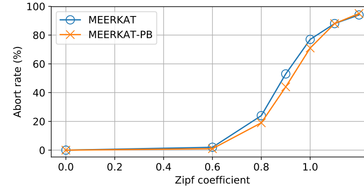
Figure 3.6: Peak throughputs comparison between Meerkat and Meerkat-PB for various zipf coefficients, 64 server threads for (a) YCSB-T transactions containing one read-modify-write operation, and (b) long, read-heavy Retwis transactions. Meerkat outperforms its primary-backup variant for low to medium contended workloads.

Figure 3.6 clearly shows the trade-off between the two systems. For YCSB-T, Meerkat provides 50% higher throughput until the zipf coefficient reaches 0.87. After that, Meerkat's OCC mechanism causes the throughput to drop more sharply than Meerkat-PB, making Meerkat's performance worse at higher contention rates. For Retwis's longer transactions, Meerkat-PB is able to match the performance of Meerkat; however, at higher Zipf coefficients, Meerkat-PB outperforms Meerkat.

Figure 3.7 shows the reason for the drop in performance. At lower contention rates and with shorter (YCSB-T) transactions, the lack of coordination gives Meerkat higher performance while both systems have low abort rates. However, as the abort rate climbs with more contention and longer transactions (i.e., note that the abort rate climbs faster for Retwis), the lack of coordination hurts Meerkat and causes its throughput to drop faster. High contention workloads require more coordination to support efficiently, which makes them fundamentally at odds with multicore scalability.



(a) YCSB-T transactions



(b) Retwis transactions

Figure 3.7: Abort rates at peak throughputs comparison between Meerkat and its primary-backup variant for various zipf coefficients, 64 server threads, and 3 replicas for (a) short YCSB transactions containing 1 read-modify-write operation, and (b) the complex, read-heavy Retwis workload. Meerkat has slightly higher abort rate as it needs to collect multiple favorable votes to commit transactions.

3

3.6. RELATED WORK

THERE is significant previous work in transactional storage systems, replication, and multicore scalability. While past research has focused on one or two of these topics, there exists little work on the combination of all three. As we showed in Figure 3.1, new advances in datacenter networks will make multicore scalability an increasing concern for replicated, transactional storage, forcing researchers to delve into all three topics.

Replicated, transactional storage. Fault-tolerance is crucial for modern transactional storage systems, thus plenty of research has been done to understand how various replication techniques work with various transactional storage systems, but less attention has been paid to the multicore aspect.

Due to its simplicity, the most popular replication technique for transactional storage systems is primary-backup, where only the primary executes and validates transactions and ships updates using a log. Many commercial databases – MySQL [51], PostgreSQL [52], Microsoft SQL Server [53], among others – also support parallel execution at the primary, but the backups consume the log serially, thus becoming a bottleneck, as observed in KuaFu [46]. These primary-backup solutions trivially violate ZCP's cross-replica coordination rule and do not achieve Meerkat's performance/scalability goal. Both KuaFu and Scalable Replay [54] enable more concurrency at replicas, but both solutions also violate ZCP's first rule – KuaFu still requires log synchronization and Scalable Replay requires a global atomic counter.

State-machine replication, where transactions are first ordered then executed in the same order on all replicas, is another popular option, adopted by many previous systems [47, 55]. In general, these solutions violate both tenets of ZCP, requiring both cross-replica and cross-core on the shared operation log. Calvin [47] improves on state of the

art by using deterministic locking on the replicas for better performance but still requires cross-replica coordination.

Chain replication [7] offers an alternative; while it still requires cross-replica coordination, communications between replicas does not increase with increasing numbers of replicas. There has not been a multicore-scalable transactional store deployed on chain replication, but we speculate that it would provide better performance at higher contention than Meerkat at the cost of higher overall latency per transaction.

Several transaction protocols are able to eliminate cross-replica coordination, including the classic Thomas algorithm [8] and recent systems like Janus [9]. However, these systems still use a shared log, as they were generally not designed to be multicore scalable, and so cannot avoid cross-core coordination. It may be possible to redesign these systems to meet ZCP, which represents an interesting future direction.

Multicore Scalability for Unreplicated, In-memory Storage. Meerkat follows a long line of research on unreplicated, multicore, in-memory storage systems. Silo [20], TicToc [21], Cicada [22], ERMIA [23], MOCC [24] all strive to achieve better multicore scalability using a variety of designs. These include both single-version optimistic concurrency control [20, 21] and multi-version concurrency control designs [22, 23]. Some use more sophisticated concurrency control mechanisms to ensure serializability; for example, ERMIA uses a Serial Safety Net to reuse a snapshot isolation mechanism while guaranteeing serializability [23]. Others use careful design to provide scalability properties beyond DAP; for example, Silo not only avoids cross-core coordination for committing disjoint read-write transactions, but it also eliminates cross-core synchronization for all read-only transactions. Wu et al [56] provide a good overview of the design choices involved in these in-memory concurrency control protocols and their relative benefits.

While these systems perform extremely well, they cannot be easily extended to a replicated environment. In particular, the concurrency control protocol in each system cannot be trivially extended to support coordination between replicas to ensure consistency. Replication further requires data structures that may not be multicore scalable and are not easily made to scale.

General Multicore Replication. Replication for general multicore applications remains a difficult problem. Without insight into application operations, replicas must use some form of determinism to ensure consistency. For example, Paxos Made Transparent [57] captures all system calls and uses deterministic multi-threading. Eve [48] takes a more speculative approach, letting replicas execute in parallel and rolling back operations in

case of inconsistencies. Rex [58] executes a transaction first at a primary, and logs all non-deterministic decisions the primary made during the execution and replays them at the replicas. These heavy-weight mechanisms require significant cross-core coordination, making it impossible to scale these systems as the number of cores grows.

3.7. SUMMARY

With the proliferation of kernel-bypass technologies and faster datacenter networks, existing distributed system designs will be able to scale to the large number of cores being deployed in the datacenters. This chapter presented a new guideline for the design of multicore-scalable distributed systems – ZCP – and a new multicore-scalable, replicated, in-memory, transactional storage system, Meerkat.

As we showed in our experiments, cross-core and cross-replica coordination will increasingly dominate and limit the performance of existing systems. A state-of-the-art replicated, in-memory storage system can only scale to 6-8 server threads, while Meerkat is able to scale to 10× the number of cores with a corresponding increase in throughput. We hope that these results encourage researchers to focus more on the design of multicore-scalable distributed systems in the future.

4

Diskless Recovery

In the previous chapter we described the design of Meerkat, an in-memory storage system that provides fault-tolerance and strong correctness guarantees. Designing a replica recovery protocol for Meerkat is simple if the replica is allowed to log all the requests it receives and their outcomes to a persistent storage device (e.g., HDD, SSD) before taking any subsequent actions (e.g., replying to the sender of the request) – in other words, the replica is allowed to use write-ahead logging techniques. This would not only provide durability and preserve atomicity but would allow the replica to recover locally, from its own persistent storage device, without the need to contact the other replicas. The downside of this recovery mechanism is that it requires many writes to the persistent storage device, which can be too expensive for some applications, and that it prevents replicas that lost access to their persistent storage device from recovering – disk drives failures are a regular occurence in large-scale data centers [59, 60]. Therefore, Meerkat also provides a recovery mechanism that reduces the number of writes to the persistent storage device and requires a very small piece of application state to be written to. In this chapter we explore recovery mechanisms that do not require writes to persistent storage devices, impose minimal changes to the existing protocols, and impose a negligible performance penalty.

More generally, we address the problem of how to build recoverable shared objects even when replicas lose their entire state. We consider the *Diskless Crash-Recovery* model: each replica process in the system may go down at any time; upon recovery, it loses all

state it had before the crash except for its identity. However, processes can run a *recovery protocol* to reconstruct their state before deeming themselves operational again. This model matches the way that many distributed systems are built in practice.

The Diskless Crash-Recovery model (DCR) is more challenging than the traditional Crash-Stop model (CS) or the Crash-Recovery with Stable Storage model (CRSS). The main challenge is that an invariant that holds at one process may not hold on that process's next incarnation after recovery. This leads to the problem of *unstable quorums*: it is possible for a majority of processes to acknowledge a write operation, and yet processes can still subsequently lose that knowledge after crash and recovery.

We provide a general mechanism for building recoverable shared objects in the DCR model. We show that an operation can be made recoverable once it is stored by a *crash-consistent quorum*, which we informally define as one where no recoveries happen during the quorum responses. Crash-consistent quorums can be efficiently identified using a mechanism called the *crash vector*: a vector, maintained by each process, that tracks the latest known incarnation of each process. By including crash vectors in protocol messages, processes can identify the causal relationship between crash recoveries and other operations. This makes it possible to discard responses that are not part of a crash-consistent quorum. We show that this is sufficient to make storage mechanisms recoverable.

The crash-consistent quorum approach is a general strategy for making storage primitives recoverable. We give two concrete examples, both of which are always safe and guarantee liveness during periods of stability; other storage primitives are also possible:

- First, we build a *multi-writer, multi-reader atomic register* by extending the well-known ABD protocol [61] with crash vectors. This improves on the best prior protocol by Konwar et al. [62], $RADON_R^{(S)}$, for this problem: it requires fewer rounds (2 rather than 3), requires fewer nodes to participate in the protocol (a simple majority vs 3/4), and has a less restrictive liveness condition.
- Second, we construct a *single-writer, single-reader atomic set*, which has weaker semantics yet permits a more efficient implementation, requiring only a single round of communication for writes. We refer to this algorithm as *virtual stable storage*, as it offers consistency semantics similar to a local disk. We show that the virtual stable storage protocol can be used to transform any protocol that operates in the traditional CS or CRSS models to one that operates in DCR.

We discuss the application of this work to state machine replication, a widely used distributed system technique. Recovering from disk failures is an important concern in practice, and recent replication protocols attempt to support recovery after complete

loss of state. Surprisingly, we find that each of the three such protocols [33, 59, 63] can lose data. We identify a general problem: while these protocols go to great lengths to ensure that a recovering replica reconstructs the set of operations it previously processed, they fail to recover critical *promises* the replica has previously made, e.g., to elect a new leader. This is due to the fact that these protocols rely on *unstable quorums* to persist these promises. This causes nodes to break important invariants upon recovery, causing the system to violate safety properties. Our approach provides a correct, general, and efficient solution.

To summarize, this chapter makes the following contributions:

- It formalizes a *Diskless Crash-Recovery (DCR)* failure model in a way that captures the challenges of long-lived applications (Section 4.2).
- It introduces the notion of *crash-consistent quorums* and provides two communication primitives for reading from and writing to crash-consistent quorums (Section 4.3).
- It presents algorithms built on top of our communications primitives for two different shared objects in the DCR model: an atomic multi-writer, multi-reader register and an atomic single-writer, single-reader set. The former is a general purpose register which demonstrates the generality of our approach, while the latter provides a *virtual stable storage* interface that can be used to port any protocol in the CRSS model to one for the DCR model (Section 4.4).
- Finally, it examines prior protocols for state machine replication in the DCR failure model and demonstrates flaws in these protocols that lead to violations of safety properties. Our two communication primitives can provide correct solutions (Section 4.5).

4

4.1. Background and Related Work

Static Systems. A static system comprises a fixed, finite set of processes. Fault-tolerant protocols for reliable storage for static systems have been studied extensively in the Crash-Stop (CS) failure model, where processes that fail never rejoin the system, and the Crash-Recovery with Stable Storage model (CRSS). In the latter model, processes recover with the same state after a crash. Consensus and related problems, in particular, have been studied extensively in these settings [64–66]. In CRSS, a crashed and recovered node is no different than one which was temporarily unavailable; asynchronous algorithms that tolerate lossy networks are inherently robust to these types of failures [65].

Our work addresses the challenge of implementing fault-tolerant shared objects in

a *Diskless Crash-Recovery (DCR)* model, i.e., without stable storage. In particular, we demonstrate a communication primitive which persists quorum knowledge even in the presence of diskless recoveries, in the spirit of [61, 67].

Prior work on fault-tolerant shared objects and consensus without stable storage generally requires some subset of the processes to never fail [68, 69]. Aguilera et al. [68] showed an impossibility result for a crash-recovery model: even with certain synchrony assumptions, consensus cannot be solved *without at least one process that never crashes*. The main differentiator between that and this work is that in their model, the states of processes were binary—either "up" or "down." We overcome this limitation by adding an extra "recovering" state. As long as the number of processes which are "down" or "recovering" at any given time is bounded, certain problems can be solved even *without processes that never fail*. This addition is especially well suited to capturing the needs of long-lived applications in which processes occasionally fail and need to rebuild state.

Recently, Konwar et al. [62] presented a set of algorithms for implementing an atomic multi-writer, multi-reader (MWMR) register in a model similar to ours. We generalize and improve on this work using new primitives for crash-consistent quorums. Our techniques are applicable to other forms of shared objects as well, and our MWMR register is more efficient: it requires one fewer phase and a simple majority quorum (vs 3/4). Like that work, we also draw on the ABD protocol [61] for our implementation of an atomic MWMR register. We also draw on the network stability models in [62] to characterize the conditions under which our shared objects guarantee progress.

Recovering without disk state is an important practical concern for real systems. Several recent practical state machine replication systems [33, 59, 63] incorporate ad hoc recovery mechanisms for nodes to recover from total disk loss. The common intuition behind these approaches is that a synchronous write to disk can be replaced with a write to a quorum of other nodes, recovering after a failure by performing a quorum read. However, we show that these protocols are not correct; they can lose data in certain failure scenarios. A more recent design, Replacement [70], provides a mechanism for replacing failed processes. Like our work and the epoch vectors in JPaxos [63], it draws on concepts like version vectors [71] and vector clocks [72] to determine the causal dependencies between replacements and other operations. We build on these techniques to provide generic communication primitives in DCR.

Dynamic Systems. In a dynamic setting, processes may leave or join the system at will. At any given time, the system membership consists of the processes that have joined and have not yet left the system. Although we consider a static system, DCR may

be viewed as a dynamic system with a finite concurrency level [73], i.e, where there is a finite bound on the maximum number of processes that are simultaneously active, over all runs. Here, a recovering process without state is equivalent to a newly joined process, and a process which crashes is equivalent to a process that leaves the system without an explicit announcement .

Many dynamic systems implement *reconfiguration* protocols [33, 74–79]. Reconfiguration allows one to change the set of members allowed to participate in the computation. This process allows both adding new processes and removing processes from the system. Reconfiguration is a more general problem than recovery: it can be used to handle disk failure by introducing a recovering node without state as a new member and removing its previous incarnation. However, general reconfiguration protocols are a blunt instrument, as they must be able to handle completely changing the membership to a disjoint set of processes (potentially even with a different size). As a result, these protocols are costly. Most use consensus to agree on the order of reconfigurations, which delays the processing of concurrent operations [80]. DynaStore [77] is the first proposal which does not require consensus, but reconfigurations can still delay R/W operations [80]. SmartMerge [81] improves on DynaStore by offering a more expressive reconfiguration interface. Recovery is a special case of reconfiguration, where each recovering process replaces, and has the same identity as, a previously crashed process. As a result, it permits more efficient solutions.

Other protocols implement shared registers and other storage primitives in churn-prone systems [82–85]. In these systems, processes are constantly joining and leaving the system, but at a bounded rate. These protocols remain safe only when churn remains within the specified bound, in contrast to our work which is always safe. Most of these protocols also require synchrony assumptions for correctness; our protocols remain correct in a fully asynchronous system. However, under these assumptions they are able to provide liveness guarantees even during constant churn.

4.2. System Model

We begin by defining our failure model: *Diskless Crash-Recovery* (DCR), a variant of the classic Crash-Recovery model where processes lose their entire state upon crashing.

We consider an asynchronous distributed system which consists of a fixed set of n processes, Π. Each process has a unique name (identifier) of some kind; we assume processes are numbered $1, \ldots, n$ for simplicity. Each process executes a protocol (formally,

it is an I/O automaton [86]) while it is up. An execution of a protocol proceeds in discrete time steps, numbered with $\mathbb{N}$, starting at $t = 0$. At each step, at most one process either processes an input action, processes a message, crashes, or restarts. If it *crashes*, the process stops receiving messages and input actions, loses its state, and is considered DOWN. A process that is DOWN can *restart* and transition back to the UP state. We make the following assumptions about a process that restarts: (1) it knows it is restarting, (2) it knows its unique name and the names of the other processes in the system (i.e., this information survives crashes), and (3) it can obtain an incarnation ID that is distinct from all the ones that it previously obtained. Note that the incarnation ID need only be unique among different incarnations of a specific process, not the entire system. These are reasonable assumptions to make for real-world systems: (1) and (2) are fixed for a given deployment, and (3) can be obtained, for example, from a source of randomness or the local processor clock.

Processes are connected by an asynchronous network. Messages can be duplicated a finite number of times or reordered arbitrarily – but not modified – by the network. We assume that if an incarnation of a process remains UP, sends a message, and an incarnation of the destination process stays UP long enough, that message will eventually be delivered.[1]

The unique incarnation ID makes it possible to distinguish different incarnations of the same process. Without unique incarnation IDs, processes are vulnerable to "replay attacks:"

Theorem 1. *Any state reached by a process that has crashed, restarted, and taken steps without receiving an input action or crashing again will always be reachable by that process.*

Proof. Suppose process p has crashed, restarted, and taken some number of steps without crashing or receiving an input action. That is, suppose that after it restarted, p received some (possibly empty) sequence of messages, $\mathscr{M}$. Because p is an I/O automaton without access to randomness or unique incarnation IDs, anytime p crashes and restarts, it restarts into the exact same state. Furthermore, if p crashes, restarts, and receives the same sequence of messages, $\mathscr{M}$, having been duplicated by the network, p will always end up in the same state. □

A corollary to Theorem 1 is that any protocol in the DCR model without unique incarnation IDs satisfying the safety properties of consensus—or even a simple shared object

[1] This model is equivalent to one in which the network can drop any message a finite number of times, with the added stipulation that processes resend messages until they are acknowledged.

such as a register—can reach a state from which terminating states are not reachable (i.e., a state of deadlock). If all processes crash and recover as in Theorem 1 before deciding a value or receiving a write, they can always return to this earlier state, so the protocol cannot safely make progress.

For simplicity of exposition, we assume that the incarnation ID increases monotonically (e.g., as could be achieved using a monotonic clock), although this requirement can be eliminated.

A restarting process must recover parts of its state. To do so, it runs a distinct *recovery protocol*. This protocol can communicate with other processes to recover previous state. Once the recovery protocol terminates, the process declares recovery complete and resumes execution of its normal protocol. We describe a process that is UP as RECOVERING if it is running its recovery protocol and OPERATIONAL when it is running the initial automaton. A protocol in this model should satisfy *recovery termination*: a recovering process eventually completes recovery and becomes OPERATIONAL, as long as it does not crash again in the meantime. This precludes vacuous solutions where recovering process never again participate in the normal protocol.

Using a separate recovery protocol matches the design of existing protocols like Viewstamped Replication [33]. Importantly, the distinction between RECOVERING and OPERATIONAL makes it possible to state failure bounds in terms of the number of OPERATIONAL processes, e.g., that fewer than half of the processes can be either DOWN or RECOVERING at any moment. This circumvents Aguilera et al.'s impossibility result for consensus [68], which does not make such a distinction (i.e., restarting processes are immediately considered OPERATIONAL). We show later how to build a stable storage abstraction that can adapt consensus-based protocols from CRSS to DCR; the resulting protocols are safe, live, and (unlike [68]) do not require certain processes to never crash.

4.3. ACHIEVING CRASH-CONSISTENT QUORUMS

MAKING shared objects recoverable in the DCR model requires a new type of quorum to capture the idea of persistent, recoverable knowledge. A simple quorum does not suffice. We demonstrate the problem through a simple straw-man example, and introduce the concepts of *crash-consistent quorums* and *crash vectors* to solve the problem. We use these to build generic quorum communication and recovery primitives.

4.3.1. UNSTABLE QUORUMS: INTUITION

Consider an intentionally simple example: a fault-tolerant *safe* register that supports a single writer and multiple readers. A safe register [87] is the weakest form of register, as the behavior of READ operations is only defined when there are no concurrent WRITEs. We further constrain the problem by allowing the writer to only ever execute one WRITE operation. That is, the only safety requirement is that once the WRITE completes, all subsequent READs that return must return the value written. READs should always return as long as a majority of processes are OPERATIONAL at any time.

4

In the Crash-Stop model, a trivial quorum protocol suffices: WRITE(val) broadcasts val to all processes and waits for acknowledgments from a quorum. Here, we consider majority quorums:

Definition 1. *A quorum Q is a set of processes such that* $Q \in \mathcal{Q} = \{Q : Q \in 2^{\Pi} \wedge |Q| > n/2\}$.

A subsequent READ would then be implemented by reading from a quorum. The quorum intersection property (i.e., $\forall Q_1, Q_2 \in \mathcal{Q} \;\; Q_1 \cap Q_2 \neq \{\}$) guarantees that at least one process will return val for a READ that happens after the WRITE. It is easy to extend this protocol to the CRSS model simply by having each process log val to disk before replying to a WRITE.

Could we use this same quorum protocol in our DCR model, where processes that crash recover without stable storage, by augmenting it with a recovery protocol that satisfies recovery termination? In fact, for this particular protocol, there is *no* recovery protocol that both guarantees the safety requirement and recovery termination – even if there is a majority of processes which are OPERATIONAL at any instant! In order to tolerate the crashes of a minority of processes and satisfy recovery termination, any recovery protocol must be able to proceed after communicating with only a simple majority of processes. However, if a process crashes in the middle of the WRITE procedure—after acknowledging val—it may recover before a majority of processes have received val. No recovery procedure that communicates only with this quorum of processes can cause the process to relearn val.

We term the resulting situation an *unstable quorum*: the WRITE operation received responses from a quorum, and yet by the time it completes there may no longer exist a majority of processes that know val. It is thus possible to form a quorum of processes that either acknowledged val but then lost it during recovery, or never received the write request (delayed by the network). A subsequent READ could fail by reading from such a quorum.

Although this is a simple example, many important systems suffer from precisely this problem of unstable quorums. We show in Section 4.5 that essentially this scenario can cause three different state machine replication protocols to lose important pieces of state.

4.3.2. CRASH-CONSISTENT QUORUMS

We can avoid this problem – both for the straw-man problem above and in the general case – by relying not just on simple quorums of responses but *crash-consistent* ones.

4

Crash Consistency. We informally define a *crash-consistent quorum* to be one where no recoveries of processes in the quorum *happen during* the quorum responses. More precisely:

Definition 2. *Let $\mathscr{E}$ be the set of all events in an execution. A set of events, $E \subseteq \mathscr{E}$, is* crash-consistent *if $\forall e_1, e_2 \in E$ there is no $e_3 \in \mathscr{E}$ that takes place at a later incarnation of the same process as e_1 such that $e_3 \rightarrow e_2$. Here, $\rightarrow$ represents Lamport's happens-before relation [88].*

In Section 4.3.3, we show how to build recoverable primitives using crash-consistent quorums, in which all quorum replies (i.e. the message send events at a quorum) are crash-consistent.

Crash Vectors. How does a process acquire a crash-consistent quorum of responses? The mechanism that allows us to ensure a crash-consistent quorum is the *crash vector*. This is a vector that contains, for each process, its latest known incarnation ID. Like a version vector, processes attach their crash vector to the relevant protocol messages and use incoming messages to update their crash vector. The crash vector thus tracks the causal relationship between crash recoveries and other operations. When acquiring a quorum on a WRITE operation, we check whether any of the crash vectors are inconsistent with each other, indicating that a recovery may have happened concurrently with one of the responses. We then discard any responses from previous incarnations of the recovering process, ensuring a crash-consistent quorum, and thus avoiding the aforementioned problem.

4.3.3. COMMUNICATION PRIMITIVES IN DCR

We now describe in detail two generic quorum communication primitives, one of which acquires a *crash-consistent quorum*, as well as a generic recovery procedure. These prim-

itives require their users to implement an abstract interface: READ-STATE, which returns a representation of the state of the shared object; UPDATE-STATE, which alters the current state with a specific value; and REBUILD-STATE, which is called during recovery and takes a set of state representations and combines them.

The ACQUIRE-QUORUM primitive writes a value to a crash-consistent quorum (i.e., using the UPDATE-STATE interface) and returns the latest state. The READ-QUORUM primitive returns a *fresh*—but possibly inconsistent—snapshot of the state as maintained at a quorum of processes. If ACQUIRE-QUORUM(val) succeeds, then any subsequent READ-QUORUM will return at least one response from a process that *knows* (i.e., has previously updated its state with) val.

The detailed protocol implementing the two primitives and the recovery procedure is presented as pseudo-code in Algorithm 3. We present the algorithm using a modified I/O automaton notation. In our protocol, **procedures** are input actions that can be invoked at any time (e.g., in a higher level protocol); **functions** are simple methods; and **upon** clauses specify how processes handle external events (i.e., messages, system initialization, and recovery). We use **guards** to prevent actions from being activated under certain conditions. If the **guard** of a message handler or **procedure** is not satisfied, no action is taken, and the message is not consumed (i.e., it remains in the network undelivered).

Each of the n process in Π maintains a *crash vector*, v, with one entry for each process in the system. Entry i in this vector tracks the latest known incarnation ID of process i. During an incarnation, a process numbers its ACQUIRE and READ messages using the local variable c to match messages with replies. When a process recovers, it gets a new value from its local, monotonic clock and updates its incarnation ID in its own vector. When the recovery procedure ends, the process becomes OPERATIONAL and signals this through the op flag. A process's crash vector is updated whenever a process learns about a newer incarnation of another process. Crash vectors are partially ordered, and a join operation, denoted $\sqcup$, is defined over vectors, where $(v_1 \sqcup v_2)[i] = \max(v_1[i],\ v_2[i])$. Initially, each process's crash vector is $[\bot,\dots,\bot]$, where $\bot$ is some value smaller than any incarnation ID.

The ACQUIRE-QUORUM function handles both writing values and recovering. ACQUIRE-QUORUM ensures the persistence of both the process's current crash vector—in particular the process's own incarnation ID in the vector—as well as the value to be written, val. It provides these guarantees by collecting responses from a quorum of processes and en-

[2]This new ACQUIRE message has the same c and val as before but will have an updated crash vector. Sending it is necessary for liveness.

Algorithm 3 Communications primitives

```
Permanent Local State:
   n ∈ ℕ+                              ▷ Number of processes
   i ∈ [1,...,n]                       ▷ Process number

Volatile Local State:
   v ← [⊥ for i ∈ [1,...,n]]           ▷ Crash vector
   op ← false                          ▷ Operational flag
   R ← {}                              ▷ Reply set
   c ← 0                               ▷ Message number

 1: upon SYSTEM-INITIALIZE
 2:    op ← true
 3: end upon
 4: upon RECOVER
 5:    v[i] ← READ-CLOCK
 6:    Σ ← ACQUIRE-QUORUM(null)
 7:    REBUILD-STATE(Σ)
 8:    op ← true
 9: end upon
10: function ACQUIRE-QUORUM(val)
11:    R ← {}
12:    c ← c + 1
13:    m ← ⟨ACQUIRE⟩
14:    m.c ← c
15:    m.val ← val
16:    for all j ∈ [1,...,n] do
17:       SEND-MESSAGE(m, j)
18:    end for
19:    Wait until |R| > n/2
20:    return {m.s : m ∈ R}
21: end function
22: function READ-QUORUM
23:    R ← {}
24:    c ← c + 1
25:    m ← ⟨READ⟩
26:    m.c ← c
27:    for all j ∈ [1,...,n] do
28:       SEND-MESSAGE(m, j)
29:    end for
30:    Wait until |R| > n/2
31:    return {m.s : m ∈ R}
32: end function
33: function SEND-MESSAGE(m, j)
34:    m.f ← i                         ▷ Sender
35:    m.v ← v
36:    Send m to process j
37: end function
38: upon receiving ⟨ACQUIRE⟩, m
39: guard: op
40:    v ← v ⊔ m.v
41:    m′ ← ⟨ACQUIRE-REP⟩
42:    if m.val ≠ null then
43:       UPDATE-STATE(m.val)
44:    end if
45:    m′.s ← READ-STATE
46:    m′.c ← m.c
47:    SEND-MESSAGE(m′, m.f)
48: end upon
49: upon receiving ⟨ACQUIRE-REP⟩, m
50: guard: m.v[i] = v[i] ∧ c = m.c
51:    v ← v ⊔ m.v
52:    Add m to R
53:    ▷ Discard inconsistent, duplicate replies
54:    while ∃m′ ∈ R where
55:             m′.v[m′.f] < v[m′.f] do
56:       Remove m′ from R
57:       Resend[2] ⟨ACQUIRE⟩ message to m′.f
58:    end while
59:    while ∃m′, m″ ∈ R where
60:             m′.f = m″.f ∧ m′ ≠ m″ do
61:       Remove m′ from R
62:    end while
63: end upon
64: upon receiving ⟨READ⟩, m
65: guard: op
66:    v ← v ⊔ m.v
67:    m′ ← ⟨READ-REP⟩
68:    m′.s ← READ-STATE
69:    m′.c ← m.c
70:    SEND-MESSAGE(m′, m.f)
71: end upon
72: upon receiving ⟨READ-REP⟩, m
73: guard: m.v[i] = v[i] ∧ c = m.c
74:    v ← v ⊔ m.v
75:    Add m to R
76: end upon
```

suring that those responses are *crash-consistent*. It uses crash vectors to detect when any process that previously replied *could have crashed* and thus could have "forgotten" the written value.

4.3.4. CORRECTNESS

We show that our primitives provide the same safety properties as writing and reading to simple quorums in the Crash-Stop model. First, we formally define quorum knowledge in the DCR context.

Definition 3 (Stable Properties). *A predicate on the history of an incarnation of a process (i.e., the sequence of events it has processed) is a* stable property *if it is monotonic (i.e., X being true of history h implies that X is true of any history with h as a prefix).*

Definition 4. *If stable property X is true of some incarnation of a process, p, we say that incarnation of p* knows X.

Definition 5 (Quorum Knowledge). *We say that a quorum Q* knows *stable property X if, for all processes $p \in Q$, one of the following holds: (1) p is* DOWN, *(2) p is* OPERATIONAL *and knows X, or (3) p is* RECOVERING *and either already knows X or will know X if and when it finishes recovery.*

In our analysis of Algorithm 3, we are concerned with knowledge of two types of stable properties: knowledge of values and knowledge of incarnation IDs. An incarnation of a process knows value val if it has either executed UPDATE-STATE(val) or executed REBUILD-STATE with an ACQUIRE-REP message in the reply set sent by a process which knew val. Knowledge of a process's incarnation ID, i, is the stable property of having an entry in a crash vector for that process greater than or equal to i.

Next, we define crash-consistency on ACQUIRE-REP messages stamped with crash vectors.

Definition 6 (Crash Consistency). *A set of* ACQUIRE-REP *messages R is* crash-consistent *if $\forall s_1, s_2 \in R\ s_1.v[s_2.f] \leq s_2.v[s_2.f]$.*

Note that Definition 6, phrased in terms of crash vectors, is equivalent to the sending events of the ACQUIRE-REP messages being crash-consistent according to Definition 2.

Definition 7 (Quorum Promise). *We say that a crash-consistent set of* ACQUIRE-REP *messages constitutes a* quorum promise *for stable property X if the set of senders of those messages is a quorum, and each sender knew X when it sent the message.*

Definition 8. *If process p sent one of the* ACQUIRE-REP *message belonging to a quorum promise received by some process, we say that p* participated *in that quorum promise.*

The post-condition of the loop on line 54 guarantees the crash-consistency of the reply set by discarding any inconsistent messages; the next loop guarantees that there is at most one message from each process in the reply set. Therefore, the termination of ACQUIRE-QUORUM (line 10) implies that the process has received a quorum promise showing that val was written and that every participant had a crash vector greater than or equal to its own vector *when it sent the* ACQUIRE *message*. This implies that whenever a process finishes recovery, it must have received a quorum promise showing that the participants in its recovery had that process's latest incarnation ID in their crash vectors.

Unlike having a stable property, that a process *participated* in a quorum promise holds across failures and recoveries. That is, we say that a process, not a specific incarnation of that process, participated in a quorum promise. Also note that only OPERATIONAL processes ever participate in a quorum promise, guaranteed by the guard on the ACQUIRE message handler.

4.3.4.1. SAFETY

Finally, we are ready to state the main safety properties of our generic read/write primitives.

Theorem 2 (Persistence of Quorum Knowledge)**.** *If at time t, some quorum, Q, knows stable property X, then for all times $t' \geq t$, Q knows X.*

Proof. We prove by strong induction on t' that the following invariant, I, holds for all $t' \geq t$: For all p in Q: (1) p is OPERATIONAL and knows X, (2) p is RECOVERING, or (3) p is DOWN. In the base case at time t, Q knows X by assumption, so I holds.

Now, assuming I holds at all times $t' - 1 \geq t$, we show that I holds at time t'. The only step any process $p \in Q$ could take to falsify I is finishing recovery. If recovery began at or before time t, then because Q knew X, p must know X now that it has finished recovering. Otherwise, if it began after time t, then p must have received some set of ACQUIRE-REP messages from a quorum, all of which were sent after time t. By quorum intersection, one of these messages must have come from some process in Q. Call this process q. Since q's ACQUIRE-REP message, m, was sent after time t and before t', by the induction hypothesis, q must have known X when it sent m. Therefore, p must know X upon finishing recovery since it updates its crash vector and rebuilds its state using m.

Since I holds for all times $t' \geq t$, this implies the theorem. □

Theorem 3 (Acquisition of Quorum Knowledge)**.** *If process p receives a quorum promise for stable property X from quorum Q, then Q knows X.*

Proof. We again prove this theorem by (strong) induction, showing that the following invariant, I, holds for all times, t:

1. If a process receives a quorum promise for stable property X from quorum Q, then Q knows X.
2. If process p ever participated in a quorum promise for X at or before time t, and p is OPERATIONAL, then p knows X.

In the base case, I holds vacuously at $t = 0$. We show that if I holds at time $t-1$, it holds at time t:

First, we consider part 1 of I. If p has received a quorum promise, R, from quorum Q for X, then because R is crash-consistent, we know that *at the time they participated in R* no process in Q had participated in the recovery[3] of any later incarnation of any other process in Q than the one that participated in R. If they had, then by the induction hypothesis(which we can apply as their participation happened before time t) , such a process would have known the recovered process's new incarnation ID when it participated in R, and R would not have been crash-consistent.

Given that fact, we will use a secondary induction to show that for all times, t', all of the processes in Q either: (1) haven't yet participated in R, (2) are DOWN, (3) are RECOVERING, or (4) are OPERATIONAL and know X. In the base case, no process in Q has yet participated in R. For the inductive step, note that the only step any process q could take that would falsify our invariant is transitioning from RECOVERING to OPERATIONAL after having participated in R. If q finished recovering, it must have received a quorum promise showing that the senders knew its new incarnation ID. By quorum intersection, at least one of these came from some process $r \in Q$. We already know r couldn't have participated in q's recovery before participating in R. So by the induction hypothesis, r knew X at the time it participated in q's recovery. Because knowledge of values and incarnation IDs is transferred through ACQUIRE-REP messages, q knows X, completing this secondary induction.

Finally, we know that since p has received R at time t, all of the process in Q have already participated in R, so all of the processes in Q are either DOWN, RECOVERING (and will know X upon finishing recovery), or are OPERATIONAL and know X. Therefore, Q knows X, and this completes the proof that part 1 of I holds at time t.

Now, we consider part 2 of I. Suppose, for the sake of contradiction, that p is OPERATIONAL at time t and doesn't know X, but participated in quorum promise R for X at or before time t. Let Q be the set of processes participating in R. Since p does not know X,

[3] That is, participated in the quorum promise needed by a recovering process, showing that the senders knew the recovering process's new incarnation ID.

p must have crashed and recovered since participating in R. Consider p's most recent recovery, and let the quorum promise it received showing that the senders knew p's new incarnation ID (or a greater one) be R'. Let the set of participants in R' be Q'. By quorum intersection, there exists some $r \in Q \cap Q'$.

It must be the case that r participated in R' before R; otherwise by induction, when r participated in R', it would have known X, and then transferred that knowledge to the current incarnation of p (at time t). r couldn't have participated in R before time t, because then by part 2 of I, it would have known p's latest incarnation ID when participating in R, violating the consistency of R. However, r cannot participate in R at or after time t, either. Because p has received a quorum promise for its new incarnation ID at or before time t, by part 1 of I, Q' knows p's new incarnation ID. By Theorem 2, Q' continues to know this at all later times. Because $r \in Q'$, it must know p's incarnation ID, and thus cannot participate in R without violating its crash-consistency. This contradicts the fact that r participates in R and completes the proof that part 2 holds at time t. □

Since ACQUIRE-QUORUM(val) obtains a quorum promise for val, Theorem 3 implies quorum knowledge of val, and Theorem 2 shows that that knowledge will persist for all future time, subsequent ACQUIRE-QUORUMS and READ-QUORUMS will get a response from a process which knows val.

4.3.4.2. LIVENESS

ACQUIRE-QUORUM and READ-QUORUM terminate if there is some quorum of processes that all remain OPERATIONAL for a sufficient period of time.[4] This is easy to see since a writing or recovering process will eventually get an ACQUIRE-REP from each of these OPERATIONAL processes, and those replies must be crash-consistent. Note that the termination of ACQUIRE-QUORUM implies the termination of the recovery procedure, RECOVER. Therefore, the same liveness conditions are required for ACQUIRE-QUORUM and for recovery termination.

We define a sufficient liveness condition, *LC*, below. It is a slightly weaker version of the network stability condition N_2 from [62]: the period in which processes must remain OPERATIONAL is shorter.

Definition 9 (Liveness Condition (*LC*))**.** *Consider a process p executing either the* ACQUIRE-QUORUM *or* READ-QUORUM *function, ϕ, and consider the following statements: (1) There exists a quorum of processes, Q, all of which consume their respective messages sent from ϕ. (2) Every process in Q either (a) remains* OPERATIONAL *during the interval* $[T_1, T_2]$,

[4]We assume that the application-provided READ-STATE, UPDATE-STATE, and REBUILD-STATE functions execute entirely locally and do not block.

where T_1 is the point in time at which ϕ was invoked and T_2 the earliest point in time at which p completes the consumption of all the responses sent by the processes in Q or (b) becomes DOWN *and remains* DOWN *during the same interval after p consumed its response. If these two statements are true for* every *invocation of a* ACQUIRE-QUORUM *or* READ-QUORUM *function, then we say that LC is satisfied.*

Our protocol implementing the group communication primitives is live if *LC* is satisfied. For *LC* to be satisfied, it is necessary that at most a minority of processes are DOWN *at any given time*. Otherwise, no process can ever receive replies from a quorum again.[5]

4.4. RECOVERABLE SHARED OBJECTS IN DCR

IN this section we demonstrate the benefits of our quorum communication primitives for DCR: generality and efficiency. We present protocols for two different shared objects: a multi-writer, multi-reader (MWMR) atomic register (Algorithm 4) and a single-writer, single-reader (SWSR) atomic set (Algorithm 5). In both protocols, READ and WRITE are intended to be invoked serially.

The first protocol implements a shared, fault-tolerant MWMR atomic register in DCR. It is more efficient and has better liveness conditions than prior work. The second protocol implements a weaker abstraction—a shared, fault-tolerant SWSR atomic set. We use this set as a basic storage primitive to provide processes with access to their own *virtual stable storage* (VSS), an emulation of a local disk. This enables easy migration of protocols to DCR.

4.4.1. MULTI-WRITER, MULTI-READER ATOMIC REGISTER

We present a protocol for implementing a fault-tolerant, recoverable multi-writer, multi-reader (MWMR) atomic register in DCR, which guarantees the linearizability of READS and WRITES [87]. Our protocol is similar to the ABD protocol [61] but augments it with a recovery procedure. Its pseudo-code is presented in Algorithm 4. Timestamps are used for version control, as in the original protocol. A timestamp is defined as a triple $(z, i, v[i])$, where $z \in N$, $i \in [1..n]$ is the ID of the writing process, and $v[i]$ is the incarnation ID of that process. Timestamps are ordered lexicographically. By replacing each quorum write phase in the original protocol with our ACQUIRE-QUORUM function and

[5] In fact, this is true if there is ever a majority of processes that are DOWN *or* RECOVERING, unless there is a set of messages currently in the network that will allow some of them to complete recovery.

Algorithm 4 Multi-writer, multi-reader atomic register in Diskless Crash-Recovery

```
Volatile Local State:
    (t, d) ← (t_0, d_0) ▷ Value of register

 1: procedure WRITE(d_new)
 2: guard: op
 3:   ▷ Get latest timestamp
 4:   Σ ← READ-QUORUM
 5:   (t_max, d_max) ← max(Σ)
 6:   ▷ Write value
 7:   t_new ← (t_max.z + 1, i, v[i])
 8:   ACQUIRE-QUORUM((t_new, d_new))
 9: end procedure

10: procedure READ
11: guard: op
12:   ▷ Get latest register value
13:   Σ ← READ-QUORUM
14:   (t_max, d_max) ← max(Σ)
15:   ▷ Write latest register value
16:   ACQUIRE-QUORUM((t_max, d_max))
17:   return d_max
18: end procedure

19: function UPDATE-STATE(val)
20:   if val.t > t then
21:     (t, d) ← val
22:   end if
23: end function

24: function READ-STATE
25:   return (t, d)
26: end function

27: function REBUILD-STATE(Σ)
28:   (t, d) ← max(Σ)
29: end function
```

each quorum read phase with READ-QUORUM, we guarantee that every successful write phase is visible to subsequent read phases, despite concurrent crashes and recoveries, thus preserving safety in DCR. The REBUILD-STATE function reconstructs a value of the register at least as new as the one of the last successful write that finished before the process crashed.

Discussion. The most recent protocol for fault-tolerant, recoverable, MWMR atomic registers is *RADON* [62]. The always-safe version of *RADON*, $RADON_R^{(S)}$, introduces an additional communication phase after each quorum write to check whether any of the processes that acknowledged the write crashed in the meantime. This increases the latency of both the READ and WRITE procedures. Also, our liveness conditions are weaker: our protocol is live if any majority of processes do not crash for a sufficient period of time, while $RADON_R^{(S)}$ requires a supermajority (3/4) of processes to not crash.

4.4.2. VIRTUAL STABLE STORAGE

Algorithm 5 presents a protocol for a fault-tolerant, recoverable, SWSR set, where the reader is the same as the writer. It guarantees that the values written by completed WRITEs and those returned in READs are returned in subsequent READs. Given the group communication primitives, its implementation is straightforward; the only additional

Algorithm 5 Single writer, single reader atomic set in Diskless Crash-Recovery

```
Permanent Local State:
    owner ▷ Owner of set flag

Volatile Local State:
    S ← {}              ▷ Local set

 1: procedure WRITE(s)
 2: guard: op ∧ owner
 3:     ACQUIRE-QUORUM({s})
 4:     S ← S ∪ {s}
 5: end procedure

 6: procedure READ
 7: guard: op ∧ owner
 8:     return S
 9: end procedure

10: function UPDATE-STATE(val)
11:     S ← S ∪ val
12: end function

13: function READ-STATE
14:     return S
15: end function

16: function REBUILD-STATE(Σ)
17:     ACQUIRE-QUORUM(Σ)
18:     S ← ⋃Σ
19: end function
```

detail is that values read during recovery should be written back to ensure atomicity (Line 17).

Discussion. We can use this set to provide a *virtual stable storage* abstraction. It is well known that any correct protocol in CS can be transformed into a correct protocol in the CRSS model by having processes write every message they receive (or the analogous state update) to their local disk before sending a reply. By equipping each process with VSS, any correct protocol in the CRSS model can then be converted into a safe protocol in the DCR model, wherein processes write to crash-consistent quorums instead of stable storage.

This conversion method, while general, might not be the most efficient for a particular problem; more efficient implementations could utilize the communications primitives in Algorithm 3 directly.

4.5. RECOVERABLE REPLICATED STATE MACHINES IN DCR

WE further extend our study of DCR to another specific problem: state machine replication (SMR). SMR is a classic approach for building fault-tolerant services [26, 88] that calls for the service to be modeled as a deterministic state machine, replicated over a group of replicas. System correctness requires each replica to execute the same set of operations in the same order, even as replicas and network links fail. This is typically achieved using a consensus-based replication protocol such as Multi-Paxos [89] or Viewstamped Replication [32, 33] to establish a global order of client requests. Once

consensus has been achieved, the replicas execute the request and respond to the client.

We examined three diskless recovery protocols for SMR: Viewstamped Replication [33], Paxos Made Live [59], and JPaxos [63]. We found that each of these protocols suffers from the problem illustrated in the example at the beginning of Section 4.3: they use regular quorums of responses (instead of crash-consistent ones) when persisting critical data, which could violate their invariants. This can lead to operations being lost, or different operations being executed at different replicas, both serious correctness violations.

All of these protocols can be correctly migrated to DCR, with little effort, using VSS write operations, as explained in Section 4.4.2. This approach is straightforward, efficient, and requires no invasive protocol modifications. As an example, we implemented the Viewstamped Replication using VSS. We evaluate this protocol experimentally in Section 4.6, showing that it efficiently handles recovery.

We further detail the three protocols we examined and give complete traces that illustrate how a sequence of crashes and recoveries can cause the system to violate the consistency guarantees they claim to provide.

4.5.1. VIEWSTAMPED REPLICATION

Originally introduced by Oki and Liskov in 1988 [32], Viewstamped Replication was one of the first consensus-based SMR protocols. VR provides linearizability [90] and guarantees liveness as long as a majority of replicas stay up for long enough during the periods of synchrony.

VR is a leader-based algorithm: the system moves through a series of numbered views, in which one node is designated as the leader. VR uses 2 protocols. During normal case execution, the leader assigns sequence numbers to incoming client requests, sends PREPARE messages to replicas, and executes the operation once it has received replies from a majority. When the leader is suspected to have failed, a *view change* protocol replaces the leader with a new one. Replicas increment their view numbers, stop processing requests in the old view, then send the new leader a VIEW-CHANGE message with their log of operations. The new leader begins processing only when it receives VIEW-CHANGE messages from a majority of replicas, ensuring that it knows about all operations successfully completed in prior views. These two protocols are equivalent to the two phases of Paxos.

Correctness for the VR protocol hinges on the following promise: once a node has sent a VIEW-CHANGE message, it processes no further requests from the previous view's leader. The protocol ensures this invariant by having processes increment their view

number and only process PREPARE messages with matching view number.

As a result, ensuring that nodes in VR can recover from failures requires that the recovery procedure continue to maintain this promise. That is, *each replica must recover in a view number at least as high as the view number in any* VIEW-CHANGE *message it has ever sent.* The original version of VR [32] achieved this invariant by writing view numbers to stable storage during view changes. A later version, "VR Revisited" [33], claimed to provide a diskless mode of operation. It used a recovery protocol and an extension to the view change protocol to, in essence, replace each write to disk with communication with a quorum of nodes. We show below that this protocol is insufficient to ensure continued correctness of the system.

Recovery Protocol VR Revisited's recovery protocol is straightforward: the recovering replica sends a RECOVERY message to all other replicas.[6] If not recovering or in the middle of a view change, every other node replies with a RECOVERY-RESPONSE containing its view number; the leader also includes its log of operations. Once the recovering replica has received a quorum of responses with the same view number, including one from that view's leader, it updates its state with the information in the log.

VR Revisited adds another phase to the view change protocol. When nodes determine a view change is necessary, they increment their view number, stop processing requests in the old view, and send a START-VIEW-CHANGE message to all other replicas. Only when replicas receive START-VIEW-CHANGE messages from a quorum of replicas do they send their VIEW-CHANGE message to the new leader and proceed as in the original protocol. The additional phase is intended to serve as the equivalent of a disk write, ensuring that replicas do not commit to a new view until a majority of them become aware of the intended view change. Together with the recovery protocol, the added phase aims to prevent violation of the view change invariant by ensuring that a crashed replica recovers in a view at least as high as any VIEW-CHANGE message it has sent.

By itself, however, this approach is not sufficient. The problem of persistence has only shifted a layer: rather than a node "forgetting" that it sent a VIEW-CHANGE message on a crash, it can forget that it sent a START-VIEW-CHANGE.

Failure Trace In Figure 4.1, we show a trace with 3 replicas, in which a new leader (NL) mistakenly overwrites the decision of a previous leader (OL).

⇒ Initially, NL suspects OL of failing and sends a START-VIEW-CHANGE message to node 1 to switch to view 1.

[6] This message contains a unique nonce to distinguish responses from different recoveries if a node recovers more than once.

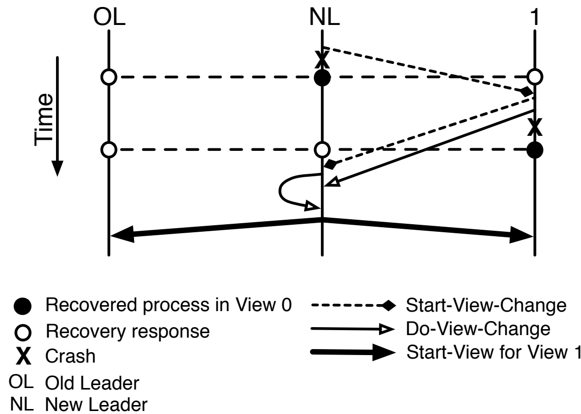


Figure 4.1: Trace showing safety violation in Viewstamped Replication Revisited [33]

⇒ NL crashes immediately after sending this message – before node 1 receives it – then immediately initiates recovery. It sends RECOVERY messages and receives RECOVERY-RESPONSE messages from OL and node 1, both of which are in view 0 – so NL recovers in view 0.

⇒ Node 1 receives the START-VIEW-CHANGE message sent by the previous incarnation of NL

⇒ Node 1 sends a START-VIEW-CHANGE message to NL for view 1. Because node 1 has a quorum of START-VIEW-CHANGE messages for view 1 (its own and the one from NL), it also sends a DO-VIEW-CHANGE message to NL. Both messages are delayed by the network.

⇒ Node 1 crashes and immediately recovers, sending RECOVERY messages to and receiving responses from OL and NL – both of which are in view 0.

⇒ NL receives START-VIEW-CHANGE and DO-VIEW-CHANGE messages from node 1. It has a quorum of START-VIEW-CHANGE messages, so it sends a DO-VIEW-CHANGE message for view 1. This is enough for it to complete the view change. It sends a START-VIEW message.

⇒ Until the START-VIEW message is received, nodes OL and 1 are still in view 0 and do not believe a view change is in progress. Thus, they can commit new operations, which the new leader NL will not know about, leaving the system in an inconsistent state..

In this trace, messages are reordered inside the network. In particular, messages are reordered across failures, i.e., messages from a prior incarnation of a node are sometimes delivered *after* messages from a later incarnation. A network with FIFO communication

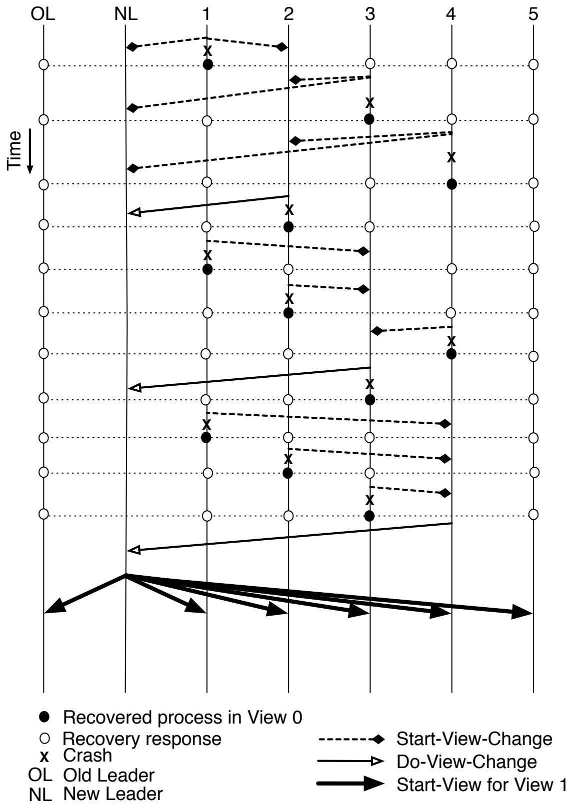


Figure 4.2: Trace showing safety violation in Viewstamped Replication Revisited [33], even when FIFO channels are assumed

channels may be able to avoid the particular violation described above.

However, message reordering is *not required* in general: there exists a trace with 7 nodes that leads to the same behavior, even with FIFO communication channels, as shown in Figure 4.2.

4.5.2. PAXOS MADE LIVE

Paxos Made Live is Google's production implementation of a Paxos [59]. It is based on the well-known Multi-Paxos optimization which chains together multiple instances of Paxos [89] and is effectively equivalent to VR. This system primarily uses stable storage to support crash recovery, but because disks can become corrupted or otherwise fail, the authors propose a version that allows recovery without disks.

Recovery Protocol On recovery, a process first uses a *catch-up* mechanism to bring itself up-to-date. The specific mechanism it uses is not described, but presumably it is an application-level state transfer from a quorum of correct processes as in VR. In order to ensure consistency, the recovering process is not allowed to participate in the protocol until it observes a completed instance of successful consensus after its recovery, i.e., until it learns that at least a quorum have agreed on a value for a new consensus instance. This mechanism suffers from a similar problem to the one in VR. Although it protects against losing ACKNOWLEDGMENTS (i.e., PREPARE-OK messages in VR), it does not protect against losing PROMISES made to potential new leaders (i.e., VR's DO-VIEW-CHANGE messages).

4

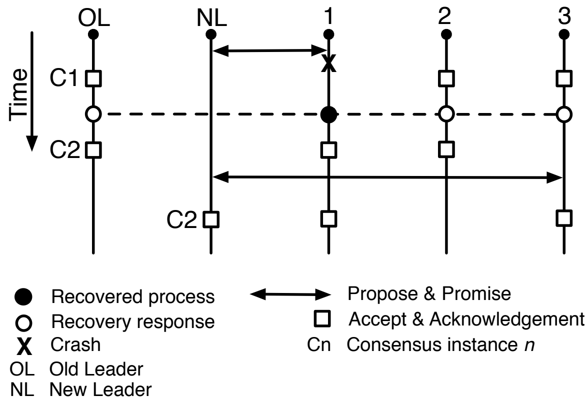


Figure 4.3: Trace showing safety violation in Paxos Made Live [59]

Failure Trace Figure 4.3 shows a trace with 5 processes that leads to a new leader mistakenly deciding a new value for a prior successful instance of consensus, overwriting the decision of the previous leader:

⇒ Initially, node OL is the leader.

⇒ NL suspects the leader of having failed and sends a PROPOSE message proposing itself as the next leader.

⇒ Node 1 receives NL's proposal and sends a PROMISE to NL, promising that it will not accept any further operations from OL.

⇒ Node 1 crashes and immediately recovers.

⇒ OL selects a new value for the next instance, C1, and sends ACCEPT and receives ACKNOWLEDGMENT messages from nodes 2 and 3. The operation is committed, and node 1 completes recovery because it has now observed an instance of consensus.

⇒ OL selects a value for instance C2, and sends ACCEPTS and receives ACKNOWLEDGMENTS

from nodes 1 and 3.

⇒ Node 3 then receives NL's PROPOSE and sends NL a PROMISE.

⇒ NL has now received a quorum of PROMISE messages: from itself, node 3, and the previous incarnation of node 1. None of these observed consensus instance C2, so NL can now start instance 2 of consensus and overwrite the previous value with its own ACCEPT messages.

4.5.3. JPAXOS

4

JPaxos [63] is a state machine replication protocol based on Paxos. It is a hybrid between Multi-Paxos and VR, replacing *promises* with *views*. The protocol is presented in several deployment options, including one that does not use stable storage, i.e., supports the Diskless Crash-Recovery model.[7]

Recovery Protocol JPaxos provides a protocol for the DCR model called *epoch-based recovery*. This protocol is similar in spirit to our use of crash vectors: it attempts to maintain a vector clock of how many times each node has crashed and recovered. Each node updates this vector when it sends a recovery response to another node, and includes a copy of its vector in the PROPOSE-OK messages that it uses to commit to a new view (i.e., the equivalent of the DO-VIEW-CHANGE message). It then uses these vectors to discard all PROPOSE-OK messages that do not come from a crash-consistent quorum.

JPaxos's epoch-based recovery protocol has one critical difference from ours. It treats recovery as complete once it has received RECOVERY-ANSWER messages from a quorum of replicas. In contrast, our protocol requires recovery responses from a *crash-consistent* quorum of replicas. That is, it discards recovery responses from the quorum if it learned that the replica that sent the response itself crashed and recovered.

This subtle change has important implications for correctness. Without this, JPaxos does not correctly handle recovery of nodes when *the nodes they are recovering from* can themselves crash and recover. When this happens, JPaxos can allow a node to recover in a way that simultaneously (1) causes the node to lose the promise that it made in a previous PREPARE-OK, yet (2) does not allow other nodes to detect that the promise might have been lost. We give an example below:

Failure Trace Figure 4.4 shows a trace in which a process (node 1 in the figure) commits to a new view via a PREPARE-OK message, then crashes and recovers. When it recovers,

[7]The protocol is actually presented as using stable storage on process initialization, but only to store a monotonic value for each recovery. The monotonic counter available in our DCR model fills this need.

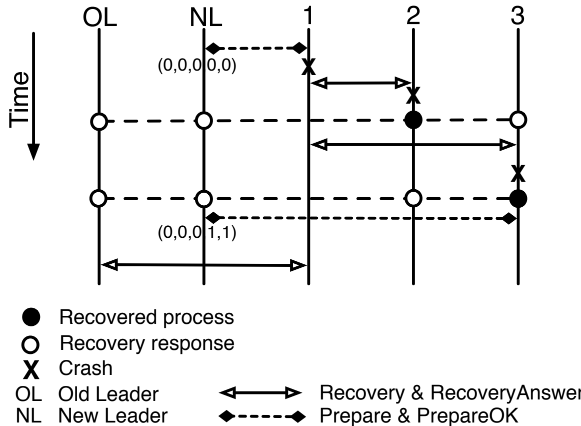


Figure 4.4: Trace showing safety violation in JPaxos [63]

it recovers in the previous view without having learned of its promise; however, the new leader (NL) does not learn about node 1's crash and recovery, and therefore does not discard its PREPARE-OK message.

⇒ The system starts in a view where OL is the leader, and all nodes have epoch vector $(0,0,0,0,0)$.

⇒ NL suspects OL of having failed and sends out a PREPARE message proposing itself as the leader of the next view.

⇒ Node 1 receives NL's PREPARE message and sends a PREPARE-OK.

⇒ Node 1 crashes and immediately recovers. It sends a RECOVERY message to node 2 and receives a RECOVERY-ANSWER. Node 2's epoch vector is now $(0,0,1,0,0)$

⇒ Node 2 crashes and immediately recovers. It sends a RECOVERY message and receives RECOVERY-ANSWERS from OL, NL, and node 3. All of these have epoch vector $(0,0,0,1,0)$, so node 2 now has this vector as well – in other words, it has lost its knowledge that node 1 crashed.

⇒ Node 1 sends a RECOVERY message to node 3 and receives a reply. Node 3's epoch vector is now $(0,0,1,1,0)$.

⇒ Node 3 crashes and immediately recovers, communicating with OL, NL, and node 2 during recovery. After recovery, its epoch vector is $(0,0,0,1,1)$.

⇒ NL sends a PREPARE message to node 3 and receives a PREPARE-OK responses. It now has a quorum of PREPARE-OK responses from itself, node 1, and node 3, so it can start a new view.

⇒ Node 1 sends a RECOVERY message to OL, and receives a response. It is now fully recovered in the original view.

⇒ OL can now commit operations (via the quorum of itself, node 1, and node 2) which will not appear in NL's new view.

Our protocol avoids this problem by checking for a crash-consistent quorum on recovery. When node 1 receives a recovery response from OL, that response will have crash vector $(0,0,0,1,1)$ – and so node 1 will discard the earlier recovery responses it received from nodes 2 and 3. It does so because it has learned that those nodes have crashed and recovered, and therefore their updates to the crash vector may not be stable.

4.6. EVALUATION

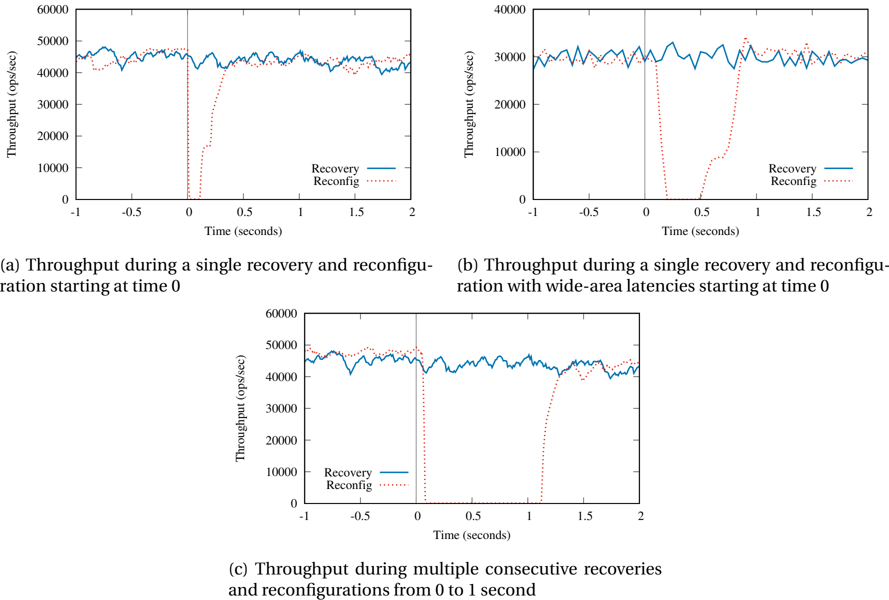
VIRTUAL stable storage (Section 4.4.2) is a practical solution for diskless recovery. To demonstrate this, we added an implementation of our recovery protocol to the reference Viewstamped Replication [33] codebase, which provides a general state machine interface. This VR implementation consists of approximately 3500 lines of Java code; less than 100 had to be modified to implement diskless recovery. All clients and replicas ran on servers with 2.5 GHz Intel Xeon E5-2680 processors and 64 GB of RAM. All experiments used three replicas (thereby tolerating one replica failure).

We conducted several experiments to show the effects of our recovery protocol on the throughput of the system. We compared it with the existing recovery protocol from VR, which we showed in Section 4.5 does not guarantee correctness. Not surprisingly, as our recovery protocol only adds a small amount of metadata to view changes and recovery, the performance is indistinguishable; we do not show these results here.

We also compared against the reconfiguration protocol already provided in the VR codebase. The reconfiguration protocol implements the R_1 method described in [74], in which reconfiguration is triggered with a special operation, $rcfg(C)$, and executed using the state machine itself. Method R_1 is more appropriate for recovery purposes than the delayed R_α [74], because processes should be allowed to recover as fast as possible. Our recovery protocol requires just one round trip delay to both persist the new crash vector and recover the lost state.

We ran each experiment multiple times and show the average throughput across those individual experiments to reduce noise from extraneous factors (e.g. Java garbage collection) and better show the impact of recovery and reconfiguration on system throughput.

Effect of a single recovery/reconfiguration. We ran our first experiment with cluster-level network latencies ($\approx 100\ \mu s$) with 24 closed-loop clients, enough to saturate the



(a) Throughput during a single recovery and reconfiguration starting at time 0

(b) Throughput during a single recovery and reconfiguration with wide-area latencies starting at time 0

(c) Throughput during multiple consecutive recoveries and reconfigurations from 0 to 1 second

Figure 4.5: Experimental results comparing a recovery protocol and a reconfiguration protocol under various scenarios

4

replicas' load. Figure 4.5a shows the effect of a single recovery/reconfiguration procedure on the throughput of the system. The blocking nature of the reconfiguration protocol is clearly visible as the throughput drops to 0 ops/sec for at least 100 ms.

Effect of network latency. The second experiment shows the effect of high latency (e.g., in a wide area network) on the throughput during a recovery/reconfiguration (Figure 4.5b). We induced 25 ms additional network latency on our testbed using the Linux traffic control tool, `tc`, emulating a geodistributed deployment. We used 900 closed-loop clients. Our recovery protocol does not affect the overall throughput. In contrast, the impact of reconfiguration is even more visible, as the throughput drops to 0 ops/sec for $\approx$ 300 ms and does not fully recover for nearly a second.

Effect of consecutive recoveries. Another consequence of a blocking reconfiguration protocol is that a single misconfigured node can prevent the system from making progress. We demonstrate this by simulating an unstable process which continuously crashes and restarts, beginning the recovery/reconfiguration procedure anew as soon as the previous recovery/reconfiguration finished. This sort of "flapping" might occur due to a configuration error, or when system load causes slow nodes to be inadvertently sus-

pected of being faulty and removed from the system[91]. As shown in Figure 4.5c, there is no visible effect on the throughput during consecutive recoveries when no data transfer is conducted (the leader must only process a single message per recovery period). In contrast, multiple consecutive reconfigurations reduced the throughput to 0 ops/sec for the whole period.

State transfer. Note that in the preceding experiments, we do not transfer any application data state or the operation log. We do this to simulate the best case scenario for a reconfiguration protocol, which may have to transfer state during the blocking period. Variations of reconfiguration protocols have been proposed that optimize state transfer using various techniques [76]; our experiments assume an ideal, free state transfer. As our recovery protocol is non-blocking, state transfer is done entirely in the background. We have measured system performance for varying amounts of state (transferring a log of up to 500K operations), and state transfer has no impact on system throughput for our recovery protocol.

4.7. SUMMARY

In this chapter we examined the Diskless Crash-Recovery model, where processes can crash and recover but lose their state. We show how to provide persistence guarantees in this model using new quorum primitives that write to and read from *crash-consistent quorums*. These general primitives allow us to construct shared objects in the DCR model. In particular, we show a MWMR atomic register protocol requiring fewer communication rounds and weaker liveness assumptions than the best prior work. We also build a SWSR atomic set that can be used to provide each process with *virtual stable storage*, which can be used to easily migrate any protocol from traditional Crash-Recovery models to DCR.

5

A PRACTICAL PRIVACY-PRESERVING SYSTEM

SHARING is the hallmark of applications in the mobile era. Mobile devices constantly collect information about their users (e.g., their location, photos, etc.) and supply it to applications, which then share this personal data with other users distributed over many mobile devices. This data ranges from the mundane to the highly sensitive, making protecting its privacy a critical challenge for modern applications.

Mobile operating systems let users restrict application access to the sensitive data on their devices (e.g., through the Android Manifest [92] or iOS privacy settings [93]). However, once an app has access, users must trust the app to ensure their privacy. Almost all apps offer their users a choice of privacy policies; unfortunately, they frequently violate these policies due to bugs [94–96] or other reasons [97–99].

While mobile OSes are effective at enforcing user privacy policies, that enforcement does not extend to application backends and other cloud services that sharing applications rely on to move data between device. Researchers have proposed distributed cloud platforms, but they only support some application features [100, 101], require application modification [102–104] or have complex user policies [105, 106]. As a result, users are left to blindly trust that applications will respect their privacy even as the apps move their data across a complex landscape of backend servers, storage systems and cloud services.

This chapter introduces a practical alternative for users. Unlike existing systems, we

aim to enforce *existing privacy policies on unmodified and untrusted sharing applications* across mobile devices and cloud services. We achieve this goal with a key insight: while mobile OSes cannot enforce policies on cloud services, the OS can vet cloud services on behalf of users and ensure that a cloud service will respect a user's policies before handing over a user's data.

We introduce a new *Secure Application Flow Enforcement* (SAFE) framework for vetting systems that handle user data. The framework defines a single guarantee: *Given a piece of user data and a* SAFE *flow policy for that data, the system must ensure that it will only release that data and any data derived from that data to: (1) another* SAFE *system or (2) an un-*SAFE *system allowed by the* SAFE *flow policy.* SAFE flow policies, detailed in Section 5.1, are access-control lists (ACLs) consisting of users and groups that reflect existing policies already set by users.

The SAFE guarantee can be applied to any software that handles user data, including systems, cloud services and user-facing applications. However, it is clearly not practical to modify all applications to meet the guarantee. Instead, we rely on SAFE *enforcement systems*, trusted systems that ensure untrusted and unmodified applications meet the SAFE guarantee. As a consequence, apps running atop a SAFE enforcement system can be deemed SAFE *apps*.

With this framework, users can begin by running a SAFE *enforcement operating system* on their mobile devices. Then, they can trust the SAFE OS – and any untrusted apps running on it – to give sensitive user data to either another SAFE system and application or another user allowed within the SAFE policy. SAFE OSes vet untrusted application backends and cloud services by using TPM-based attestation to verify that the cloud server is running a trusted systems stack, including a trusted SAFE enforcement system. Once verified, the user's mobile OS can safely send sensitive user data and be certain that the SAFE guarantee will be upheld by the SAFE enforcement system and untrusted applications running on top.

The SAFE guarantee is incredibly powerful: if a user gives a piece of data to a SAFE enforcement system along with a SAFE policy, the user can trust that the policy will be enforced no matter where the data flows until it is released to another user allowed by the policy. In other words, the SAFE framework lets users construct a chain of trust from their mobile OS to an arbitrary set of cloud services to other users' devices. Furthermore, users do not need to understand which cloud services their applications use, only to trust that the services are verified SAFE. In fact, users do not even need to understand the SAFE concept, provided that they trust their mobile OSes to be SAFE and to correctly verify that other systems that handle their data are SAFE as well.

The remainder of this chapter describes the SAFE framework (Section 5.1) and the design and implementation of three SAFE enforcement systems:

- *Agate*, a SAFE mobile OS that securely collects user policies, enforces them on untrusted mobile apps and translates them to SAFE policies for SAFE cloud services (Section 5.2).
- *Magma*, a SAFE distributed cloud runtime system that enforces SAFE policies on untrusted cloud backends using fine-grained information flow control (Section 5.3).
- *Geode*, a SAFE proxy that enforces SAFE policies on untrusted storage systems which do not manipulate user data. (Section 5.4).

Using these three systems, we demonstrate that it is possible to enforce user policies end-to-end on unmodified distributed mobile apps largely without changing the user experience. We do so for several existing applications, including a 70,000-line calendar application and a 250,000-line chat application. Furthermore, we show that the SAFE policy model supports a range of existing user policies and that systems can implement SAFE policy enforcement with little user-noticeable overhead (≈20% on a mobile device).

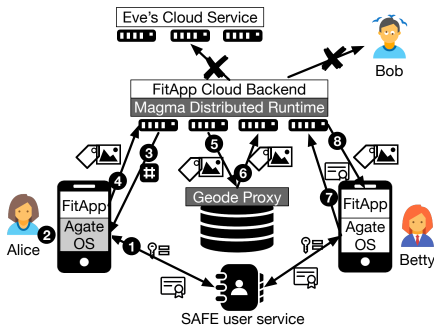
5

5.1. THE SAFE FRAMEWORK

THIS section summarizes the SAFE framework, including the system model and concepts, the policy model and the threat model. We had three goals when designing the framework: (1) minimize changes to user experience, (2) minimize changes to application code, and (3) minimize performance cost. This section reviews the ways in which the SAFE design meets these goals.

5.1.1. SAFE SYSTEM MODEL AND CONCEPTS

Figure 5.1 shows the SAFE system model. A SAFE *application* consists of processes distributed across mobile devices (i.e., mobile app clients) and cloud servers (i.e., cloud app backends) as well as cloud services used by the application (e.g., distributed storage [107, 108]). Some of the data that a SAFE application handles will have attached SAFE policies, while other data does not. We assume mobile devices are owned by users, while cloud servers are operated either by the application provider (e.g., Twitter, Facebook) or by a third-party cloud provider (e.g., Amazon). The application runs on one or more mobile OS platforms and one or more cloud platforms.



1. Alice logs in to Agate to access FitApp
2. Alice sets a policy in Agate to share her photos with Betty through FitApp
3. Agate verifies that the FitApp cloud backend is a SAFE service (it attests to running Magma)
4. Agate allows the FitApp mobile client to send Alice's photo to the FitApp cloud backend
5. Magma allows FitApp to store Alice's photo because the storage system attests to running the Geode Proxy
6. Geode releases Alice's photo back to FitApp backend
7. FitApp wants to send Alice's photo to Betty's phone, so Magma requests app and user certificates from Agate and checks the photo's SAFE policy
8. Magma allows FitApp to send Alice's photo to Betty's phone

Figure 5.1: **Example SAFE ecosystem.** We show the steps for Alice to share her photos with Betty securely through the SAFE framework. We color the systems based on Alice's trust in them; she trusts her SAFE Agate OS running on her phone (light gray), she trusts her Agate to verify (using attestation) that the (dark gray) cloud systems are SAFE, and she does not trust the white apps or systems. Magma will not let FitApp send Alice's photo to Eve's cloud service because it is not an attested SAFE cloud service. Likewise, Magma ensures that FitApp cannot share Alice's photo with Bob because Bob is not in Alice's SAFE policy.

5.1.1.1. SAFE User Service

Each SAFE deployment instance, called an *ecosystem*, is defined by a centralized, trusted user management service. The SAFE *user service* is shown at the bottom of Figure 5.1. The SAFE user service has two roles: (1) securely managing the principals used to express SAFE policies, including group membership management, and (2) verifying app and user identities to authorize the release of data to an untrusted device. The SAFE user service gives every application, user and group a unique identifier and stores a mapping between names and ids. It also authenticates users, issues certificates to untrusted devices to authorize them to access data on behalf of users and manages group membership.

Any trusted entity can launch a SAFE user service and ecosystem to support one or more applications. Apps within an ecosystem can easily exchange data and policies (if allowed by the SAFE policy) because they share the same SAFE user service and its principals. Apps outside an ecosystem must negotiate a way to translate SAFE policies when exchanging data.

The SAFE user service is not SAFE-specific; there are many existing single-sign-on (SSO) services that could be used to implement the same functionality (e.g., Google Accounts [109], OpenID [110]). The only requirement is that the service be able to authenticate users, issue certificates and manage group membership. For trust reasons,

we assume that the SAFE user service is implemented and deployed by an entity separate from the application provider; otherwise, we would have to trust the application to manage users, which many fail at [96, 111]. As an example, Google could create its own ecosystem for Google apps by deploying a SAFE user service or using Google Accounts.

5.1.1.2. SAFE MOBILE OPERATING SYSTEMS

Users run a SAFE *enforcement OS* on their mobile devices to ensure that user policies are securely captured, enforced on untrusted mobile apps and expressed to SAFE cloud services. The SAFE OS also verifies cloud services as being SAFE before allowing apps to send sensitive user data. This thesis describes the design of the Agate SAFE OS (shown on Alice and Betty's phones in Figure 5.1), but we imagine that other SAFE OSes would exist.

Before running SAFE apps, users must login to the SAFE mobile OS, which authenticates the user with the SAFE user service. The SAFE user service issues a user certificate to the SAFE OS, which authorizes it to collect data and policies on behalf of the logged-in user and hold data shared with that user. We assume that users trust any SAFE OS that they are willing to log in to. Thus, for a SAFE system to release data to an untrusted device (e.g., belonging to Alice's friend Betty), the SAFE OS on the device must present a certificate belonging to a user that is authorized to access the data (e.g., through a SAFE policy).

5.1.1.3. SAFE CLOUD ENFORCEMENT SYSTEMS

SAFE OSes can pass sensitive user data to trusted SAFE cloud services because the OS can rely on the cloud service to respect the user's policies. However, we do not expect programmers to modify all cloud services to meet the SAFE guarantee, so we rely on SAFE enforcement systems to ensure that untrusted cloud services meet the guarantee. This paper describes two SAFE cloud enforcement systems, *Magma* and *Geode*.

Magma is a distributed cloud runtime for application backends; Figure 5.1 shows it running the FitApp backend. Magma enforces the SAFE requirement using fine-grained, dynamic information flow control to track and control the flow of SAFE data through unmodified application code. Geode, shown as the storage layer in Figure 5.1, is a storage proxy for storage systems that do not modify application data. It enforces the SAFE requirement on untrusted key-value stores (e.g., memcached [108], Redis [107]) by interposing on storage accesses and encrypting and checksumming data. While we believe that these systems meet the needs of many applications, we envision the potential for other SAFE enforcement systems, including ones using existing IFC systems [102, 112], sandboxing systems [100, 113] or computation over encrypted data [101, 114].

5.1.1.4. SAFE VERIFICATION AND ATTESTATION

SAFE enforcement systems make it easier for SAFE OSes to verify that cloud services are SAFE. Rather than simply keeping a list of SAFE cloud services, cloud services *demonstrate* that they are SAFE by attesting, using trusted platform modules (TPMs), that they are running a trusted systems stack including a SAFE enforcement system. We do not innovate here; this could be achieved using a secure bootloader [115, 116], a trusted hypervisor [117], or a secure enclave mechanism [118].

The trusted hardware component measures all of the software that makes up the platform up to and including the SAFE enforcement system and produces a signed hash summarizing this software stack. The SAFE OS validates this hash against a list of SAFE software platforms. We imagine that these hashes could be provided by the mobile OS vendor (much as OS and browser vendors maintain lists of trusted SSL CAs today) or a trusted third party. Note that it is practical for OS vendors to distribute these hashes because we assume a limited number of trusted system stacks and SAFE enforcement systems; however, a trusted cloud service could also do so.

As alternative (e.g., if trusted attestation hardware is not available or a cloud vendor prefers not to reveal its deployment), the SAFE architecture can be used by having the OS vendors (or a trusted third party) validate cloud platforms. Then, these platforms (e.g., Amazon Lambda [119] or S3 [120]) would be trusted axiomatically to enforce the SAFE property. The OS vendor would simply issue them a signed certificate that the cloud provider stores and presents to mobile OS clients. Obviously, this is a less secure option than attestation because it requires trusting the cloud providers to correctly run a trusted SAFE enforcement system and would not work for application providers that provide their own infrastructure. However, we offer the alternative because many applications today run on a third-party cloud provider platform which has other strong incentives to enforce user privacy guarantees.

5.1.2. SAFE POLICIES

SAFE policies are flow policies expressed as access control lists including *users*, *groups*, and *applications*. A SAFE policy encodes: (1) the app that can access the data, and (2) the list of apps, users and groups with which the app can share that data and any data derived from it. For example, Alice can set a policy allowing her fitness app (FitApp) to share her GPS location only with Betty. We use the following notation to denote this SAFE policy: GPS = ⟨FITAPP, {BETTY}⟩. Note that this policy only allows FitApp to share Alice's GPS-derived data with Betty; Alice may have different policies for other applications.

Apps create SAFE groups that map to application-specific concepts and register them with the SAFE user service. For example, FitApp could define ALICE.RUNNING-GROUP, which lets Alice share her runs with her running partners. The SAFE user service securely manages these groups by querying the user through a SAFE OS when an app wants to add members to the group. For example, if FitApp tries to add Betty to Alice's running group, the SAFE user services will request that Alice's SAFE OS accept or deny this request.

5.1.3. TRUST AND THREAT MODEL

The SAFE framework makes it possible for users to create a chain of trust from their mobile devices and OSes to the cloud. Thus, we begin with the assumption that users trust their own mobile devices and their SAFE OS. We establish user trust by requiring users log in to their SAFE OS. This login establishes trust in several ways: (1) it authenticates the user to the SAFE OS, allowing the OS to give her access to the data and sharing policies on the mobile device and (2) it indicates to the SAFE OS that the user trusts the device to collect and hold her sensitive data.

A user login also indicates that the device and SAFE OS is trusted to hold data shared with that user. For example, if Alice gives Betty access to her GPS location, then she must trust any device and SAFE OS that Betty is willing to log in to. This extension of trust makes sense; even if Betty's phone was not running malicious software, Betty herself could extract Alice's GPS location from the phone once it is shared because Betty has physical ownership of the phone.

Users also trust attested SAFE cloud services. In particular, users trust SAFE services running on a trusted systems stack including a SAFE enforcement system. Similar to other security systems, the degree of protection offered by each SAFE enforcement system depends on its mechanism. In general, SAFE systems can suffer from timing attacks, probabilistic channels, or physical attacks on trusted components. For example, Magma uses IFC to enforce SAFE policies on untrusted applications, so it suffers from many of the same limitations as previous IFC systems [103, 104, 121–123], including termination. Despite these limitations, the SAFE framework significantly improves the security of user data handled by distributed mobile apps. Since, today, users must trust their applications, SAFE enforcement systems significantly raise the bar on attacks by malicious applications on user privacy.

5.2. AGATE, A SAFE MOBILE OS

AGATE is a SAFE enforcement operating system built atop Android, the most popular mobile OS today [124]; however, Agate's design could layer atop other mobile OSes as well (e.g., iOS). A SAFE mobile OS performs three important functions: (1) providing users with a secure user interface for logging in and setting and managing policies, (2) labeling data with SAFE policies and (3) enforcing SAFE policies.

5.2.1. AGATE ARCHITECTURE

Figure 5.2 shows the Agate mobile OS architecture. To minimize the impact on the user experience, we limit Agate to enforcing SAFE policies on data that mobile OSes already protect (essentially anything in the Android Manifest). We define these data sources as *OS-protected resources*, including hardware resources like the camera and GPS (shown below Agate in Figure 5.2), as well as software resources provided by built-in apps, like the user's calendar (shown in the middle of Figure 5.2), contacts, etc.

Agate provides two interfaces (shown in gray in Figure 5.2): (1) the *user interface* lets users securely log in, set policies and manage group membership and (2) the *syscall interface* lets applications access OS-protected resources, suggest policies and create groups.

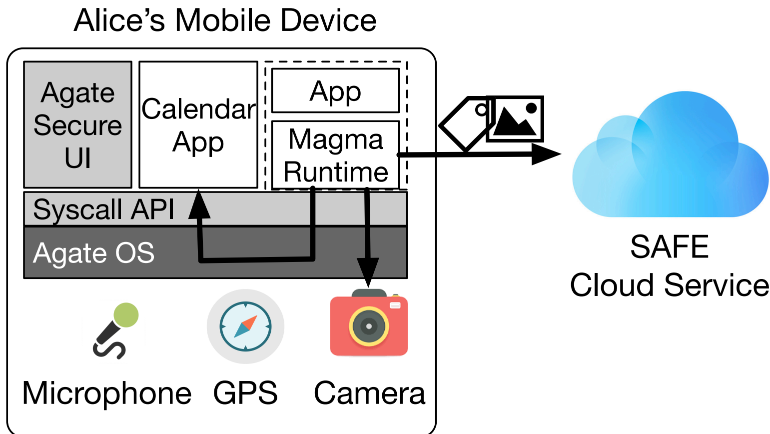


Figure 5.2: **Agate Mobile OS Architecture.** Agate mediates application access to hardware and software resources (similar to Android). It provides a secure user interface for users to log in and set Agate policies, and a syscall interface for apps to access resources and propose policies. Agate runs every app in a Magma runtime for SAFE enforcement. Magma ensures that apps only send data from OS-protected resources to other SAFE systems or users within the policy.

Agate is a SAFE enforcement system. Once an app has accessed an OS-protected resource, Agate must ensure that untrusted apps do not send data from those resources to untrusted cloud services or mobile OSes. For SAFE enforcement, Agate embeds the Magma runtime and runs every app with it. Unlike the Magma distributed cloud runtime, Agate does not not require attestation because it is sufficient that the logged-in user trusts Agate and its apps to be SAFE.

5.2.2. AGATE POLICIES

Agate policies match existing user privacy policies as closely as possible. They combine today's access control policies with SAFE flow policies and let users express: (1) which OS-protected resources an application can access, and (2) how the application can export data derived from that resource. For example, Alice can set a policy allowing her FitApp to access her GPS and share with Betty. Agate will give every piece of data that FitApp derives from her GPS the SAFE policy, GPS = ⟨FITAPP, {BETTY}⟩.

Agate lets apps create SAFE groups and suggest policies through the syscall API. For example, once FitApp has created the group, ALICE.RUNNING-GROUP, then it can offer Alice the choices: GPS = ⟨FITAPP, {ALICE}⟩ GPS = ⟨FITAPP, {ALICE, BETTY}⟩ and GPS = ⟨FITAPP, {ALICE, ALICE.RUNNING-GROUP}⟩ to share data from her GPS only with her devices, with her and Betty's device, or with her entire running group.

5.2.3. AGATE SYSCALL INTERFACE

Mobile apps interact with Agate through the syscall interface. Syscalls fall into three categories: (1) access to OS-protected resources, (2) Agate policy proposals, and (3) SAFE group management. Most of these syscalls are directly handled by Agate or coordinated with the SAFE user service. Whenever possible, we maintain the existing OS interface; for example, application access to OS-protected resources is unchanged.

Existing apps access hardware resources (e.g., the camera) through syscalls handled by the OS and software OS-protected resources through inter-process calls to built-in OS apps (similar to a user-level file system on a traditional OS). For example, on Android, an app can access a user's contacts by sending an intent to the Contacts app. When an app accesses an OS-protected resource, Agate continues to enforce an access control policy similar to today's mobile OSes (e.g., Alice gives FITAPP access to her contacts through the Android manifest) but labels any data from the resource (e.g., Betty's address) with a SAFE policy. It then gives that SAFE policy to Magma to enforce on the untrusted mobile

app and pass on to SAFE cloud services.

5.2.4. AGATE USER INTERFACE

Before the user can access SAFE apps on Agate, they must log in. Agate presents a log-in interface similar to existing mobile OSes or apps. It authenticates the user's identity with the SAFE user service to obtain a user certificate. Agate can support apps from more than one SAFE ecosystem; however, users have to log in to each ecosystem separately. Apps provide the location of their ecosystem's user service, so that Agate can retrieve the app id and certificate. Agate will ask the user to log in again only if it does not already have a certificate from that user service.

Agate requires a secure way for users to specify policies for their OS-protected resources and manage groups. To avoid application interference, Agate cannot trust the application to display or draw the policy-creation UI. Instead, it displays the UI in a *secure user interface*, similar to the UI used today when mobile apps request additional app permissions [93]. Any policy-creation UI should be secure from: (1) *visual manipulation* by the application (e.g., changing what the user sees); (2) *input forgery* by the application (e.g., entering a policy on Alice's behalf); and (3) *clickjacking* or similar attacks [125]. Prior work has considered these and other secure UI requirements in depth [126, 127]; we refer to that work for implementation details for these properties.

To extend Agate's support for text-based applications, we added a new secure text box. For example, a chat application can open a secure Agate text box, which reads text from the user and then labels it with a policy before handing it back to the application. We assume the mobile OS allows users to verify that they are operating in the context of a trusted built-in application or Agate UI (e.g., an indicator in the system navigation bar [128]).

5.2.5. AGATE SAFE ENFORCEMENT

Agate performs SAFE enforcement because we want to let untrusted mobile apps manipulate user data from OS-protected resources but not let apps leak that data to un-SAFE cloud services or unattested OSes. It leverages Magma for this enforcement by running every mobile app in a Magma runtime environment and giving Magma SAFE policies for any data from OS-protected resources. Magma is a modified IFC JVM, making it suitable for mobile apps as well as cloud backends. We leave the discussion of how Magma implements SAFE enforcement to the next section.

5.3. MAGMA, A SAFE RUNTIME SYSTEM

MAGMA is a runtime system that provides SAFE enforcement for unmodified application processes. Because Magma is a SAFE enforcement system, it can always pass data to another process running the Magma runtime. Thus, it is easy to construct a distributed runtime platform from processes running Magma across distributed nodes.

Magma's responsibilities as a SAFE enforcement system is to: (1) take SAFE policies and turn them into IFC tags for its tracking mechanism, (2) propagate those tags as the application manipulates user data, and (3) check tags to ensure that SAFE policies are enforced when the application backend sends data to mobile devices.

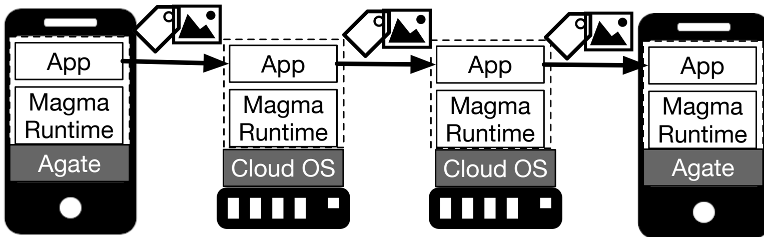


Figure 5.3: **Magma Architecture.** Every Magma process runs application code atop the Magma runtime and can always pass data to another (verified) Magma process. Magma processes can run embedded in Agate OSes on mobile devices or as stand-alone components on cloud servers.

5

5.3.1. MAGMA ARCHITECTURE

Figure 5.3 shows the architecture of a distributed Magma runtime system. Every Magma process (shown as dotted rectangles) runs the Magma application runtime under the application. Magma processes can run on mobile devices as part of the Agate OS or as part of a cloud service.

We prototyped the Magma runtime by extending the Dalvik JVM to support Android apps on Agate. However, other application runtimes (e.g., Python, Scala, Go) could be used as well. Using a language runtime lets Magma support fine-grained IFC with low overhead. In contrast, supporting low-level languages would cause Magma either to overtaint or impose too much performance overhead. We observe that most sharing applications today run in a managed runtime, so this trade-off is a reasonable way to achieve our goal of supporting unmodified applications.

Magma uses fine-grained information tracking, rather than tainting processes, because backend sharing app code may handle the data of many users over time. Eventu-

ally, processes running the cloud service would become so tainted that they would not be able to release data to any users. Another option would be to start a new process for each request (e.g., using something like Amazon Lambda [119]); however, that could add significant latency, so we leave that option to future work.

5.3.2. MAGMA IFC MODEL

Magma's IFC model is similar to other IFC systems with one key difference: SAFE *policies directly map to IFC labels.* As a result, Magma is able to work with unmodified applications without the help of programmer or users. Magma automatically tags data from an Agate OS or another SAFE cloud service with the SAFE policy before handing that data to the application. Magma propagates and enforces those policies represented as tags across the entire backend cloud application.

There are three types of IFC principals in Magma – users, groups and applications – which map to the principals in SAFE flow policies. We directly use SAFE ids for Magma labels. There are two types of labels in Magma: the mutable *data labels*, which carry the SAFE policy, and the immutable *process labels*, which encode authorization and help enforce the SAFE policies.

Data Labels. Magma data labels are tags of the form $l = \{\text{O}_1, \text{O}_2 \rightarrow \text{A}_1, \text{U}_1, \text{U}_2, \text{G}_1\}$, where O_1, O_2 are the user principal ids of the owners of the labeled data, A_1 is the principal id of an application allowed to access the labeled data and $\text{U}_1, \text{U}_2, \text{G}_1$ are users and groups allowed to view the labeled data. We will refer to the set $\{\text{A}_1, \text{U}_1, \text{U}_2, \forall$ user principal id $u \in \text{G}_1\}$ as $readers(l)$.

Process Labels. Each Magma process is labeled with an application principal (e.g., $\{\text{A}_1\}$) and, if the process runs on Agate, a user principal (e.g., $\{\text{A}_1, \text{U}_1\}$). Magma requires these two to enforce the SAFE flow policies, which dictate which application can move a user's data, as well as, whom the app can give that data to.

Information Flow Rules. To enforce SAFE policies, Magma must guarantee the following security property:

> *Data from an OS-protected resource labeled with a non-empty initial data label l_1 may reach a process labeled with label l_2 only if $l_2 \subseteq readers(l_1)$.*

It does so by applying the following two data and policy propagation rules:

Intra-process propagation. Inside a process, data is allowed to flow freely (i.e., no flow control checks are performed) but data labels may change. Any data *derived* from one or more labeled data sources is labeled with a label which reflects all the SAFE policies involved. For example, if the application combines two pieces of labeled data, their labels are *merged* into the resulting label l, where the owners are the union of the two sets of owners and where $readers(l)$ is the intersection of the two sets of readers. That is, given two pieces of data with labels $l_1 = \{o_1 \rightarrow readers(l_1)\}$ and $l_2 = \{o_2 \rightarrow readers(l_2)\}$, the resulting label of any derived data is $\{o_1, o_2 \rightarrow readers(l_1) \cap readers(l_2)\}$.

Inter-process propagation. Data labeled with the current data label l_1 is allowed to flow to a process labeled with label l_2 only if $l_2 \subseteq readers(l_1)$. If the data is allowed to flow to the new process, it maintains its label, l_1, until an intra-process propagation rule is applied.

These rules ensure that data protected by SAFE policies, and data derived from that data, flows only to other processes running the same app (or another allowed app), and, if the process is running on an untrusted Agate OS, a process with a logged in user that is in the SAFE policy.

5

5.3.3. MAGMA FLOW TRACKING

Magma implements both explicit and implicit flow tracking. Explicit flows are caused by direct assignment (e.g., `x = gps-loc`), whereas implicit flows are caused by control flow (e.g., `if (gps-loc == home) {x = true}`. Magma's explicit flow tracking mechanism is relatively straightforward; as apps propagate data, Magma propagates the corresponding tags, joining the tags by following the IFC rules. Handling implicit flow through unmodified applications is more complicated, so much so that many systems [102, 129, 130] ignore the problem altogether. However, they are an important way that user privacy can be violated; thus, Magma must consider them.

Using the previous example, `if (gps-loc == home) {x = true}`, the value assigned to `x` is a literal value containing no sensitive labels. However, the execution of the assignment operation reveals information regarding Alice's location. It is also worth noting that information is leaked even if the conditional branch is *not* executed since the absence of an update to `x` also reveals information regarding Alice's location (i.e., she is not home).

For implicit flow tracking, Magma uses a combination of static analysis and runtime taint propagation. For every conditional block, Magma's static analyzer identifies both the set of variables that are updated in either branch (e.g., `x`) and the control flow vari-

ables that determine the conditional execution of the updates (e.g., `gps-loc`). The static analysis pass then inserts code that causes the runtime taint propagation system to update the labels of the modified variables to include the labels associated with the control flow variables, regardless of whether the conditional is executed.

Our Magma prototype runs this static analysis on Java bytecode as it loads apps. It uses a control flow graph representation of the program and resembles the techniques outlined in [131–133]. Instrumented code for runtime taint tracking needs to be added to every control flow block that might contain sensitive data, as well as any function invoked from either branch of these control flow blocks.

Instrumenting control flow has the potential to increase code size and execution time. To mitigate this problem, Magma's static analyzer makes a conservative pass to determine which control flow blocks will *never* result in implicit flows, because they never access tainted data. Similarly, it identifies functions that are never called from tainted contexts. We found this pruning to be useful in practice because many tagged data objects are unlikely to be used to make control flow decisions: for example, most apps do not branch on the bytes of a JPEG image.

5.3.4. MAGMA SAFE ENFORCEMENT

Magma uses flow control to enforce SAFE policies on untrusted applications. Because Magma directly expresses SAFE policies as IFC labels, it can use labels tagged on the data to check policies. Thus, simply by preventing flows that violate Magma's IFC rules, Magma can ensure that an application meets the SAFE requirement.

First, Magma always permits applications to send data to another trusted SAFE cloud service. Magma verifies that cloud services are safe using one of the methods detailed in Section 5.1.1.4 (e.g., comparing a TPM hash). It then translates the IFC label into a SAFE policy and securely send it along with the data.

When a Magma application sends labeled data to an unattested mobile device (i.e., belonging to another user), Magma performs a policy enforcement check. It first checks that the mobile device is running an authorized SAFE OS and retrieves the application id and logged-in user id of the process that will receive the data. The user id and app id are both provided as signed certificates from the SAFE user service when the user logs in and starts the app.

If the app id matches, then Magma intersects $readers(l)$, where l is the label on the data being sent, with the logged-in user principal of the destination process. If the intersection is not empty, then the Magma tags are translated into a SAFE policy and

sent together to the destination. If the intersection is empty, then the destination is not permitted to receive the data and Magma returns an error to the application. This check is sufficient to ensure SAFE policies are respected.

5.4. GEODE, A SAFE STORAGE PROXY

APPLICATION cloud back-ends often need to persistently store data for fault-tolerance or archival storage. To support this requirement, we provide a cryptographic proxy, Geode, to make existing, untrusted storage systems SAFE. Geode takes its approach to building secure storage from untrusted infrastructure similar to prior work on TPM-based filesystems and databases [134–136].

5.4.1. GEODE INTERFACE AND GUARANTEES

Geode provides a key-value object storage interface, like Amazon S3 [120]. It provides three guarantees: (1) *confidentiality* - the storage service cannot read user data, and it will be released only according to the SAFE policy on the data; (2) *integrity* - each object and its policy can only be modified by the application that created it and cannot be tampered with by the storage service, and (3) *single-object linearizability* of updates.

Geode provides linearizability per object both because it is a desirable property for reasoning about concurrency, and to prevent a malicious storage service from returning incorrect values under the guise of weak consistency. Geode can be used with any storage service that provides the same interface, guarantees linearizability, and does not manipulate user data itself. Many storage systems meet these requirements, including most weak consistency distributed storage systems (e.g., S3, Redis [107]), which typically provide per-key linearizability.

5.4.2. GEODE ARCHITECTURE

Figure 5.4 shows Geode's architecture. Geode operates multiple proxy nodes, each responsible for a different portion of the keyspace. Due to space constraints, we do not discuss fault-tolerance of proxy nodes, other than to note that it can be handled by replication and logging techniques as in previous systems [134, 137]. Each Geode node is equipped with a TPM or other trusted hardware component; in addition to attesting to SAFE clients that the server is running the Geode proxy, it also provides the Geode proxy

with access to a sealed encryption key and a tamper-proof monotonic counter.

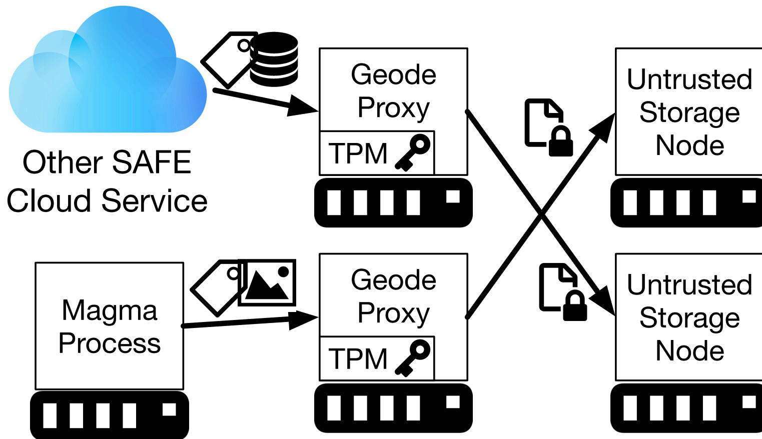


Figure 5.4: **Geode Architecture.** Geode interposes on access to an untrusted storage system from SAFE systems. It securely checksums and encrypts the data and SAFE policies before handing them to the storage system. Geode ensures that only SAFE systems and users within the SAFE policy can retrieve the data.

Note that Geode could be the only cloud service that an app needs. Many applications today (e.g., to-do lists or recipe apps) use only a cloud storage system (e.g., Dropbox [138]) and do not require a separate cloud backend app.

Geode could be deployed in a number of ways. It could be run by the application provider, the storage provider or a third party entity. For the best performance, Geode should be co-located with the storage system.

5.4.3. Geode SAFE Enforcement

Geode interposes on every access to the storage system. For each write operation, Geode prepends a header to every stored object with its SAFE policy. It then generates a random initialization vector and uses it to encrypt the block (using AES-128 in CBC mode with the secure key). Encryption ensures that the storage system cannot read the data, and the initialization vector prevents known-plaintext attacks. Subsequently, it records the initialization vector and a SHA-256 HMAC of the block contents. These are added to a per-object integrity table, itself encrypted with the same key, which also contains the latest monotonic counter value; this table is stored in a special object in the underlying storage system. (A more efficient implementation might use a Merkle tree [139] to prevent having to rewrite the table on each update.)

On each read operation, the Geode proxy fetches and decrypts the object. It verifies

that the hash of the object matches the one stored in the integrity table, and the version number of the table is up to date. The hash ensures that the storage service did not tamper with the object or its policy, and the version number prevents it from rolling back to an earlier state. If the data object has a SAFE policy, Geode first checks if the requesting application is either a SAFE service or a SAFE OS. If it is a SAFE service, Geode securely sends the decrypted data and SAFE policies to the service. If it is a SAFE OS, Geode checks the SAFE policy for the user certificate presented by the SAFE OS (similar to Agate and Magma).

5.5. EVALUATION

IN addition to ensuring that user privacy policies are respected, we stated three goals in the SAFE framework design: (1) minimize changes to the user experience, (2) minimize changes to application code and (3) minimize performance cost. In this section, we evaluate the effectiveness of the SAFE design in meeting these goals.

5

5.5.1. IMPLEMENTATION

To support Android sharing apps, we prototyped Agate and Magma using the Android OS and Dalvik JVM, respectively. Agate runs on ARM-based mobile devices while Magma runs on both ARM and x86 architectures. This section describes the implementation of these prototypes.

5.5.1.1. AGATE MOBILE OS PROTOTYPE

Agate extends Android into a SAFE OS by adding support for secure user log-in, policy collection and SAFE enforcement. Our prototype UI differs slightly from the one imagined for Agate; because Android already provides users with a secure way to set access policies for their OS-protected resources, the Agate UI only provides settings for SAFE policies. Our policy management interface consists of dialog boxes drawn in the context of the current application rather than in a separate, trusted application. A more secure prototype would show the policy interface in a separate application, as demonstrated in prior work [126].

Our current prototype leaves access control to existing Android mechanisms but interposes on accesses to collect policies and attach labels for Magma to use. After attaching labels, Agate uses its embedded Magma runtime to enforce the SAFE guarantee. Our prototype interposes only on calls to the built-in camera and GPS resources (i.e., the

`takePhoto()` and `getLastKnownLocation()` system calls); in a full implementation, similar modifications would be required for other OS-protected resources.

5.5.1.2. MAGMA PROTOTYPE

Magma is a SAFE enforcement runtime that: (1) translates SAFE policies into IFC tags, (2) tracks both explicit and implicit flows and (3) enforces SAFE policies with IFC checks. Magma's explicit flow tracking mechanism is based on TaintDroid [129] for Android 4.3_r1. However, TaintDroid is a limited taint-tracking – not flow-enforcement – system, so Magma requires extensive modifications to TaintDroid's mechanisms. For example, while TaintDroid tracks binary taints for only 32 sources, Magma must use more complex IFC labels and rules to represent SAFE policies.

TaintDroid has no implicit flow tracking, so Magma implements its own mechanism. Magma's hybrid mechanism uses a custom analysis tool to insert annotations into Android dex files, and then dynamically propagates labels at runtime via those annotations. Magma's static analysis tool uses the Soot framework for Java/Android apps [140], and consists of 5,400 lines of Java code to perform class hierarchy analysis, global method call flow analysis, control flow analysis within methods, side effect analysis inside conditionals, and insertion of taint tracking code for implicit flows.

Magma inherits some performance optimizations from TaintDroid that could lead to overtainting. Neither TaintDroid nor Magma performs fine-grained flow tracking through native code due to the performance overhead. Instead, Magma uses a conservative heuristic that assigns the result of a native code function to a combination of the taints of the input arguments. Similarly, TaintDroid keeps a single taint label for an entire array, which could cause overtainting due to false sharing. Phosphor [141], a newer JVM-based taint tracking mechanism, eliminates the potential for overtainting at a reasonable performance cost.

5.5.2. SECURITY ANALYSIS

We first evaluate the effectiveness of SAFE's security guarantees. We examine the top 10 web security risks and the top 10 mobile security risks identified by the Open Web Application Security Project (OWASP) [142, 143], summarized in Table 5.1. As the checkmarks in Table 5.1 indicate, the SAFE framework can successfully prevent applications from leaking user data for nearly all of the most common web and mobile vulnerabilities compared to Android, which handles only three.

In many cases, simply using a SAFE enforcement system suffices to avoid the vulnerability. For example, applications frequently inadvertently leak user data and violate user

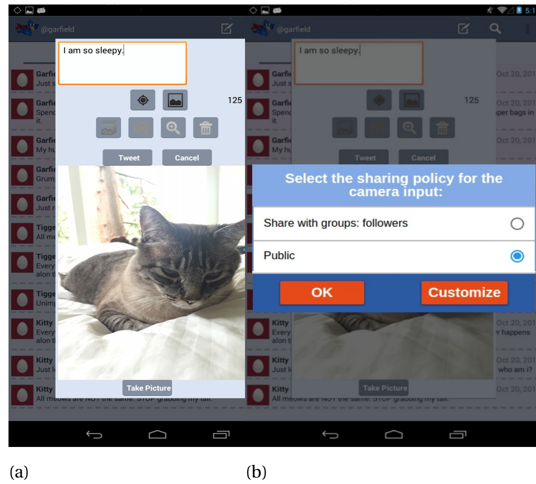
Table 5.1: **Protection offered by Android and the SAFE framework against top web and mobile vulnerabilities [142, 143].** Related items from the lists are merged. Android handles only a small subset of these issues, while SAFE covers nearly all.

Vulnerability	Android	SAFE
Broken access control	–	✓
Broken authentication	–	✓
Broken cryptography	–	✓
Buffer overflow	✓	✓
Client or server side code injection	–	✓
Cross site scripting	✓	✓
Insecure data storage	–	–
Insecure direct object references	✓	✓
Insufficient transport layer protection	–	✓
Improper error handling	–	✓
Improper session handling	–	✓
Lack of binary protections	–	✓
Missing function-level access control	–	✓
Path traversal & command injection (server)	–	✓
Security decisions via untrusted inputs	–	✓
Sensitive data exposure	–	✓
Unintended data leakage	–	✓

policies through *improper error handling*. However, an IFC-based enforcement system, like Magma, would ensure that applications cannot release user data except to another SAFE system or a user within the SAFE policy running a SAFE OS. In fact, while porting applications to Agate, we often found Magma barring the release of error message for debugging to us.

Finally, the entire SAFE framework is designed to prevent *broken access control*. Rather than trusting applications to correctly implement access control checks, a SAFE OS requires that a cloud service run a SAFE enforcement system, which is trusted to perform these checks on behalf of the application. For example, Magma would not allow Facebook to release Mark Zuckerberg's photos to users that are not in his friends list [94].

While the SAFE framework protects against *insecure data storage* in the cloud (i.e., by using the Geode proxy), we explicitly do not handle risks that can be exploited by improper storage on untrusted user devices. For example, once Alice releases her photo to Betty, she must trust that Betty does not copy the photo to untrusted cloud storage or lose her phone. Overall, the analysis shown in Table 5.1 demonstrates that, through the use of the SAFE framework, taking trust away from applications and putting it into the hands of attested SAFE enforcement systems can help users avoid many of the vulnerabilities that plague mobile sharing applications today.



(a) (b)

Figure 5.5: **Agate UI for a Twitter-like application.** When the user takes a photo in SAFETweet, shown in 5(a), Agate interposes on the system call and lets the user set a policy via a secure, system-controlled user interface, shown in 5(b). Agate labels the photo with the specified SAFE policy, which Magma enforces, once the photo is given to SAFETweet.

5.5.3. USER EXPERIENCE

A key goal of the SAFE framework is to minimize changes to the user experience and use existing user policies. We evaluate whether we achieved this goal by exploring the Agate user interface.

To demonstrate Agate's UI, we use an open-source Twitter clone [144] that we ported to our SAFE framework, which we call SAFETweet. While interacting with SAFETweet, Alice takes a photo of her cat, Max, to post to her feed, which requires SAFETweet to access her camera. At this point, Agate interposes on the system call to: (1) ask permission for SAFETweet to access the camera and (2) let Alice to set a SAFE policy for all data that SAFETweet receives from the camera (i.e., photos). Once Alice has given permission and set a policy, Agate will give the photo and SAFE policy to the embedded Magma runtime to return to the app.

We note two key aspects of Agate's UI that are enabled by the SAFE framework. First, it looks and feels much like a user's experience with Twitter because SAFETweet can propose SAFE policies that match those that it would offer today. Second, although Agate creates a familiar user experience, it can enforce SAFE policies on untrusted apps using its Magma runtime *and* verify that cloud services will also enforce those policies. So while the user experience remains unchanged, the security properties are completely different with the SAFE framework.

Table 5.2: For each distributed application, we list the unmodified Android app ported to Agate for the client side and the unmodified Java app ported to Magma for the server side, along with their size in lines of code.

App	Ported Client	LoC	Ported Server	LoC
SAFEChat	Xabber [145]	78K	Openfire [146]	190K
SAFETweet	Twimight [144]	13K	MinnieTwitter [147]	1.2K
SAFECal	aCal [148]	30K	Cosmo [149]	40K

5.5.4. PROGRAMMABILITY AND PORTING EXPERIENCE

Another goal of the SAFE framework is to minimize application code changes. To gain experience with SAFE applications, we created three SAFE sharing applications. We ported three unmodified Java server applications as cloud app backends and three Android apps as mobile app clients, as listed in Table 5.2. Using the Agate UI, we placed SAFE policies on different sources of data for each application; for example, in SAFEChat (70K LoC), we used Agate's secure text input facility to create a private chat between Alice and Betty.

5

To port mobile app clients to Agate and Magma, the only changes made to the application code were those needed to incorporate Agate APIs (e.g., to use the system call to propose policies and treat invalid flow exceptions). In SAFECal (250K LoC), we found both explicit and implicit flows that might violate Alice's policy; e.g., Bob, who is not Alice's co-worker, cannot view Alice's meetings this week (an explicit flow), or check whether Alice is free at 3 on Tuesday (an implicit flow).

With Magma, we were particularly interested in issues with overtainting, policy accumulation on data, and unexpected flow restrictions. We experienced no problems with the application code itself: indeed, the cloud app backends (Openfire, MinnieTwitter, Calendar) required *no modification* to run on Magma. However, we did encounter overtainting in libraries used by cloud backend apps, particularly for communications and parsing, such as the core Java libraries (BufferedReader, OutputStreamReader), Java RMI, and dom4j. For example, the message serialization libraries reuse memory buffers, leading to unnecessary data overtainting and policy accumulation and blocking valid flows. After manually fixing the problematic libraries, the applications worked as expected. Note that these fixes could be reused for other applications by releasing Magma-compatible Java libraries.

While we believe Agate and Magma are two representative SAFE systems, the programming experience will vary between different SAFE OSes and SAFE enforcement systems. However, with a variety of options, programmers can choose the best one for their

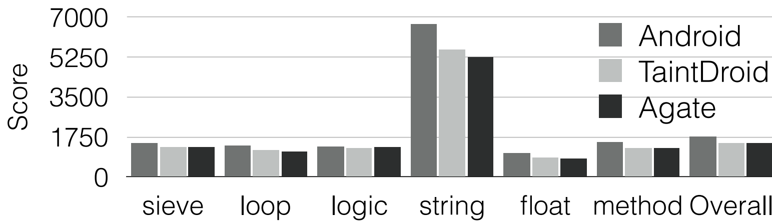


Figure 5.6: **CaffeineMark microbenchmark for Android Dalvik, TaintDroid and Agate (higher is better).** Both Agate and TaintDroid impose an overhead, but Agate's overhead is similar to TaintDroid's.

cloud backend or mobile app.

5.5.5. PERFORMANCE

5

Finally, we measure the performance cost of meeting the SAFE requirement. In our experiments, mobile devices run Agate with its embedded Magma runtime and cloud servers run the distributed Magma runtime. Each server contained 2 quad-core Intel Xeon E5335 CPUs with 8GB of DRAM running Ubuntu 12.04. The mobile devices were first-gen Nexus 7 tablets (1.3 GHz quad-core Cortex A9, 1 GB DRAM). Our servers shared one top-of-rack switch and connected to tablets via a local-area wireless network.

5.5.5.1. MICROBENCHMARKS

As a baseline, we compare the performance of Magma running on Agate to unmodified Dalvik and TaintDroid running on Android. We execute mobile apps without tainted data. We use CaffeineMark 3.0, a computationally intensive program commonly used as a microbenchmark for Java. CaffeineMark does not access data with SAFE policies, so there are no tags to be tracked; however, TaintDroid and Magma must still propagate and merge empty labels, so this measurement gives us a lower bound on the performance cost for each system.

CaffeineMark scores roughly correlated with the number of Java instructions that the JVM interpreter executed per second. The overall CaffeineMark score is the geometric mean of the individual scores. Figure 5.6 shows the results for an Android tablet. On this baseline (no SAFE policies and no tags), Agate and Magma impose little overhead relative to TaintDroid (between .3% and 5.8%, with an average of 1.8%). The CaffeineMark overall score for Agate is within 16% of baseline Android, while TaintDroid's overall score is within 14.5% of Android.

TaintDroid's overhead stays constant as data accumulates more taint because it uses a single-bit representation and tracks only 15 OS-protected resources. Magma also prop-

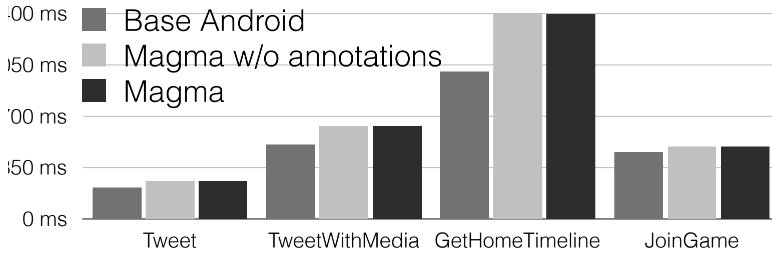


Figure 5.7: **Latency of SAFETweet and WordsWithFriends functions.** The figure shows that Magma imposes roughly 20% overhead compared to unsecured Android. joinGame is the only function that requires static annotation, but those costs are extremely low.

agates a single 32-bit tag per primitive or primitive array, but these tags are references to a list of all policies with which the data is tainted. Thus, while Magma's cost for propagating tags remains constant as taint accumulates, the cost of merging when combining two tainted pieces of data increases with the number of policies tainting the data.

To evaluate the impact of accumulating taint in Magma, we measure the cost of merging up to 20 SAFE labels, each with 20 principals/tags (10 users and 10 groups). We consider this measurement an upper bound because it would require the application to have at least 20 policy options, *each* with 20 principals. In practice, policies would become unwieldy for both applications and users at a much smaller number (i.e., 5-10 policies each with a small number of principals). We found that merging two labels with 20 principals each took 6 µs. Overhead increases with the number of labels on the data up to 20 µs to merge 20 labels with 20 tags each.

5.5.6. APPLICATION PERFORMANCE

To validate our expectations about how apps and users use SAFE policies, and to measure Agate's performance overheads for a full application, we use two distributed applications: (1) the SAFETweet app mentioned above (MinnieTwitter + Twimight), and (2) a multi-player game (WordsWithFriends) that we implemented from scratch. We run mobile client apps with Magma on Agate and the app backends on the Magma distributed cloud runtime.

Figure 5.7 shows latency for tweet, tweetWithMedia and getHomeTimeLine from SAFETweet, and joinGame from WordsWithFriends. For each function, we show latency for the unsecured app on Android, Agate+Magma without static analysis annotations, and Agate+Magma with annotations.

Tweet does not access sensitive data with SAFE policies and thus shows the ba-

sic cost of making Android SAFE, which is 17%. TweetWithMedia includes a labeled photo; we used the default policy suggested by SAFETweet, which is CAMERA = ⟨SAFETWEET, {USER.FOLLOWERS}⟩. Overhead increases to 20% because Magma must call the SAFE user service to translate group/user names into principal ids. Some optimizations could reduce this cost, including caching id mappings and optimizing TaintDroid's disk writes. The overhead of GetHomeTimeline is 22% because it accesses more tainted data (2 photos), requiring Agate to check the policy on each photo. Most of the extra time is due to remote calls to resolve group membership, which again could be reduced with caching [104]. Currently, each operation required three RPCs to perform the checks before sending the photo.

For SAFETweet, merging and static analysis did not add to the runtime overhead. Magma propagates but never merges labels because SAFETweet never derives new data from photos. Static analysis found no implicit flows as SAFETweet never uses photos as branch conditions. We expect this behavior to be typical for most applications that access photos.

WordsWithFriends' JoinGame operation accesses the GPS to find nearby friends for game-play, so its default policy for the GPS is GPS = ⟨WORDSWITHFRIENDS, {USER.FRIENDS}⟩. Static analysis added only five annotations for WordsWithFriends because the control flow based on the GPS is limited to checking for nearby friends.

JoinGame branches on tagged GPS locations every time it compares two locations, so it must track more labeled data than SAFETweet operations and accumulates taint as it runs due to the comparisons. However, JoinGame does not release the user's GPS location (because it performs the comparisons on the user's device), so it does not have to resolve user names or group membership for enforcement checks, incurring a lower overhead than SAFETweet operations.

While these applications are prototypes, we believe they use the SAFE framework in a representative way. Data derived from OS-protected resources with SAFE policies were typically copied or transmitted but rarely merged with other sensitive data. Furthermore, static analysis is effective at determining which conditionals operated on sensitive data. As a result, Magma is able to reduce the number of taint merge operations and implicit flow annotations, and thus keep the overheads low.

Overall, our results show that Agate and Magma's performance is very close to TaintDroid's and within approximately 20% of the performance of the base Android system, both of which have no policy enforcement. From a user's qualitative point of view, the difference is not detectable in using either prototype application. Although we have not yet tried to optimize our prototype, we feel that this difference is well worth the addi-

tional privacy guarantees that the SAFE framework provides.

5.6. RELATED WORK

IN designing the SAFE framework we drew inspiration from many existing privacy-preserving and trust management systems. Although this thesis presents the design of only three SAFE enforcement systems, we envision many others being built atop existing systems.

Modern OSes for smartphones incorporate access control mechanisms that let users control which resources applications can access, e.g., through an Android manifest file. Significant research [150] builds on this idea to give users better security and more effective access control. For example, Access Control Gadgets [151] provide a more intuitive user interface for permission granting, and Preservers [152] lets users choose the code/policies that mediate data access. Our SAFE framework and its SAFE systems provide stronger guarantees by enforcing user policies beyond the phone.

5

Distributed platforms like πBox [100], Cleanroom [113] and Radiatus [153] protect user privacy by isolating each user in a sandbox environment. While isolating users makes it easy to enforce privacy for applications where users do not interact, social applications where users *want* to selectively share their data do not work well. All communication between users must go through a restrictive interface and be vetted by the system. However, sandboxing has the advantage that all data (not just OS-protected resources) is protected and cannot be exposed even to the application developer. πBox uses differential privacy [154] to control *how much* data can be released to the developer. A sandboxing-based SAFE enforcement system would be useful for apps in which part of or all of a user's data does not need to be shared with other users; for example, a cloud service that categorizes a users photos using machine learning.

Unlike other IFC runtimes [102, 104, 155, 156], Magma is explicitly designed to support SAFE flow policies. As a result, Magma can directly translate SAFE flow policies into IFC labels, rather than requiring users or programmers to set IFC policies. This design minimizes changes to both the user interface and application code.

However, the SAFE framework makes it possible to incorporate other IFC-based systems, provided that there is a way automatically translate the SAFE policies into their IFC policy model. This requirement may limit the untrusted applications they can support. For example, IFC-based systems that use coarse-grained tracking [122, 123, 157] could offer better performance than Magma but require more information about the application's architecture to deal with overtainting. Language-based IFC systems [103, 158–160]

seem even less suitable because they require information about application variables and functions, making it difficult to translate SAFE policies into their policy model.

Geode is a simple proxy for storage systems that do not manipulate user data. It is inspired by previous systems that leverage a trusted hardware platform to build secure storage out of untrusted components [134–137]. More complex options that enable computation on stored data include IFDB [112] and CryptDB [114].

Magma builds on TaintDroid, a binary instrumentation tool for programmers to find leaks of tainted data on Android applications. TaintDroid has a different goal: it aims only to *detect* flows, not to stop them. As a result, it does not have a notion of policies – it tracks only a single bit of taint – nor any enforcement mechanism. TaintDroid is also a single-node system; its flow tracking ends at the boundary of a single mobile device.

Our SAFE framework is inspired by public key infrastructure (PKI). While PKI has its issues (e.g., too many certificate authorities, authorities issuing certificates that they do not own), there have been efforts to address them (e.g., http public key pinning [161]). We hope that SAFE will avoid many of these pitfalls by relying on attestation instead of authorities. In particular, three factors differentiate SAFE from PKI: (1) it is more difficult for a system to become a trusted SAFE enforcement system than a certificate authority, (2) users can inspect the code of open-source SAFE enforcement systems and (3) mobile OSes will not verify a cloud service as SAFE unless it can attest that it is running a trusted SAFE enforcement system.

5.7. SUMMARY

This chapter introduced the SAFE framework for distributed mobile apps. SAFE provides a system guarantee that user policies will be enforced on sensitive user data no matter where it flows. The framework relies on SAFE enforcement systems to provide this guarantee without requiring application modifications.

The SAFE framework leverages mobile OSes to vet cloud services. Using attestation, a user's SAFE mobile OS can verify that a cloud service is running a trusted systems stack and a SAFE enforcement system. Once verified, the mobile OS can trust the cloud service to enforce SAFE policies even if it is running untrusted applications.

We presented three SAFE enforcement systems: Agate, a mobile OS; Magma, a distributed runtime system; and Geode, a distributed storage proxy. Using these three systems, we were able to run several unmodified applications across mobile devices and cloud server and enforce SAFE policies across the entire application. Our results demonstrate that the SAFE framework is a practical way for users to create a chain of trust from

their mobile devices to the cloud.

6

Conclusions and Future Research Directions

This thesis introduced new systems to address two of the major challenges of developing large-scale cloud applications: managing the large amount of application state, and preserving the privacy of users' data.

First, motivated by the extreme requirements of these applications and the latest trends in hardware and network infrastructure, this thesis introduced Meerkat, an in-memory, distributed, replicated, multi-threaded, key-value storage system that provides fault-tolerance, one-copy serializable transactions, and is multicore scalable for transactions that access disjoint sets of data items. Meerkat commits transactions in a single round trip (which makes it attractive in a geo-distributed and geo-replicated scenario as well) and avoids cross-core and cross-replica coordination for transactions that are disjoint access, i.e., transactions that do not access – read or write – the same data item. To recover from replica failures, Meerkat comprises a replica recovery protocol that needs to rarely write to a persistent storage devices a very small amount of application state. However, in cases where the use of persistent storage devices is not desirable at all, Meerkat can adopt the replica recovery mechanism for purely in-memory storage systems that is also introduced in this thesis. Our experiments found that Meerkat is able to scale up to 80 hyper-threads and execute 8.3 million transactions per second. Meerkat represents an improvement of 12× on state-of-the-art, fault-tolerant, in-memory, transactional storage systems built using leader-based replication and a shared transaction log. Pushing

forward into this space, we think two future research directions are worth exploring:

Stretching the Zero-Coordination Principle further. *Can we avoid cross-core and cross-replica coordination for conflicting, or even commutative, (as opposed to disjoint access) transactions without sacrificing the other properties that Meerkat offers (i.e., the same degree of fault-tolerance, one round trip commit)?*
Transactions should be allowed to read the same data item (which is not considered a conflict) without affecting the multicore scalability of the system. This property, known in the STM (software transactional memory) community as "invisible reads", is offered by several previous single-node databases, like Silo and can significantly improve the performance and multicore scalability of the system for highly contended read-heavy workloads. However, offering this property in addition to fault-tolerance and one round trip commit seems hard, if not impossible.

Reducing quorum sizes. *Can we commit transactions in one round trip using smaller quorum sizes? If so, at what cost?*
Meerkat uses a fault-tolerant consensus protocol that, like several other consensus protocols and their variants, uses $n = 2f + 1$ number of replicas to be able to tolerate f failures sometimes seamlessly, without the need to agressively replace failed replicas (the system makes progress despite up to f concurrent and potentially permanent failures). Moreover, similar to other fast consensus protocols, Meerkat requires fast quorums of at least $\frac{3}{4}n$ replicas in order to commit transactions in a single round trip. Reducing these quorum sizes can result in better tail latency and better throughput. Recent work in the consensus realm, FlexiblePaxos, demonstrates that the slow quorum sizes used in classic consensus protocols can be reduced (i.e., they can be smaller than $\frac{n}{2}$) if the system is willing to trade off fault-tolerance (i.e., f is smaller, even for the same number of total replicas, n). Can we adopt some ideas from this recent work to trade-off fault-tolerance to reduce Meerkat's quorum sizes?

Second, this thesis introduced SAFE, a privacy-preserving system that enforces users' privacy policies on largely unmodified and untrusted sharing applications across mobile devices and cloud services. SAFE currently comprises 3 components, Agate, Magma, and Geode, although we envision that other cloud services can be made SAFE in the future. In this thesis we demonstrated that SAFE can enforce users' privacy policies on three different applications, with a small performance penalty. However, we believe more research is necessary to make SAFE more practical and able to support a wider range of cloud applications. For every SAFE component, we propose a research direction we think is worth

exploring:

Blending the SAFE OS protected resources into the application for unchanged user experience. *Can we design a SAFE OS able to safely collect users' data and privacy policies while giving the users the impression they are interacting directly with the untrusted application?*
Agate is a SAFE mobile OS responsible for securely collecting users' data and privacy policies. How well the SAFE OS protected resources blend into the application dictates the impact on users' experience. Agate leverages existing trusted OS mobile apps, which come pre-installed with every modern mobile OS today, to safely collect the information and privacy policies from the users and then shares it with the untrusted mobile-cloud application. Unfortunately, although not too disruptive, this process is still not identical to the the one the users are familiar when they share their data privacy policies directly with the application.

Providing a secure general-purpose library of declassification modules. *Can we allow applications to safely declassify certain information that became overtainted?*
Magma implements an information flow control mechanism in order to automatically enforce the privacy policies on unmodified applications. The mechanism taints the information as it moves through the application's variables. Therefore, the result of a computation would inherit the union of the privacy policies of all information used in the computation. In some cases, this union of policies is more restrictive than necessary and can interfere with the functionality of the application. However, allowing the application to freely declassify the result (i.e., lift the restrictions of the policies) is not safe. The applications we experimented with did not need to declassify overtainted infromation to function properly. However, applications that collect statistics, for example, might need a secure way in which to declassify such information. Can we design a general-purpose library suitable for a wide range of applications, trusted to securely declassify over-tainted information?

Providing a more secure SAFE storage service. *Can we provide a more secure SAFE storage service whith a reasonable performance penalty?*
For performance reasons, we designed Geode as a proxy to an existing unmodified storage service that stores the policies alongside the users' data. However, the storage service is a complex piece of software and thus there is a high risk it might be exploited. To ensure that the SAFE storage service will not violate the privacy policies, it could be interesting to compare the following two approaches: 1) run the

SAFE storage service on top of Magma as well, and 2) leverage recent advancements in verification tools to prove that it is impossible for the verified implementation of Goede (the proxy and the storage service) to violate the privacy policies.

REFERENCES

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Addison Wesley, 1987).

[2] G. Graefe, *Revisiting optimistic and pessimistic concurrency control*, Tech. Rep. HPE-2016-47 (Hewlett Packard Labs, 2016).

[3] G. Schlageter, *Problems of optimistic concurrency control in distributed database systems,* ACM SIGMOD Rec. **12**, 6266 (1982).

[4] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, *MDCC: multi-data center consistency,* in *Proc. of EuroSys* (2013).

[5] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurhty, and D. R. K. Ports, *Building consistent transactions with inconsistent replication,* in *Proc. of SOSP* (2015).

[6] L. Lamport, D. Malkhi, and L. Zhou, *Vertical Paxos and primary-backup replication,* in *Proc. of PODC* (2009).

[7] R. van Renesse and F. B. Schneider, *Chain replication for supporting high throughput and availability.* in *Proc. of PLDI* (2004).

[8] R. H. Thomas, *A majority consensus approach to concurrency control for multiple copy databases,* ACM Transactions on Database Systems **4**, 180 (1979).

[9] S. Mu, L. Nelson, W. Lloyd, and J. Li, *Consolidating Concurrency Control and Consensus for Commits under Conflicts,* in *Proc. of PLDI* (2016).

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, *Spanner: Google's Globally-Distributed Database,* in *Proc. of PLDI* (2012).

[11] T. David, R. Guerraoui, and V. Trigonakis, *Everything you always wanted to know about synchronization but were afraid to ask,* in *Proc. of SOSP* (2013).

[12] D. Hedin and A. Sabelfeld, *A perspective on information-flow control,* in *Software Safety and Security* (2012).

[13] F. Li, *Cloud-native database systems at Alibaba: Opportunities and challenges,* in *Proc. of VLDB* (2019).

[14] D. Tahara, *Cross shard transactions at 10 million requests per second,* https://blogs.dropbox.com/tech/2018/11/cross-shard-transactions-at-10-million-requests-per-second/ (2018).

[15] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports, *Just say NO to Paxos overhead: Replacing consensus with network ordering,* in *Proc. of PLDI* (2016).

[16] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, *The End of a Myth: Distributed Transactions Can Scale,* in *Proc. of VLDB* (2017).

[17] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, *The end of an Architectural Era: (It's Time for a Complete Rewrite),* in *Proc. of VLDB* (2007).

[18] J. Cowling and B. Liskov, *Granola: Low-Overhead Distributed Transaction Coordination,* in *Proc. of USENIX ATC* (2012).

[19] A. Kalia, M. Kaminsky, and D. Andersen, *Datacenter RPCs can be General and Fast,* in *Proc. of NSDI* (2019).

[20] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, *Speedy Transactions in Multicore In-memory Databases,* in *Proc. of SOSP* (2013).

[21] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, *TicToc: Time Traveling Optimistic Concurrency Control,* in *Proc. of SIGMOD* (2016).

[22] H. Lim, M. Kaminsky, and D. G. Andersen, *Cicada: Dependably Fast Multi-Core In-Memory Transactions,* in *Proc. of SIGMOD* (2017).

[23] K. Kim, T. Wang, R. Johnson, and I. Pandis, *ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads,* in *Proc. of SIGMOD* (2016).

[24] T. Wang and H. Kimura, *Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores,* in *Proc. of VLDB* (2016).

[25] H. Kimura, *FOEDUS: OLTP Engine for a Thousand Cores and NVRAM,* in *Proc. of SIGMOD* (2015).

[26] F. B. Schneider, *Implementing fault-tolerant services using the state machine approach: A tutorial,* ACM Computing Surveys (1990).

[27] A. Dey, A. Fekete, R. Nambiar, and U. Rohm, *YCSB+T: Benchmarking web-scale transactional databases,* in *Proc. of ICDEW* (2014).

[28] A. Israeli and L. Rappoport, *Disjoint-access-parallel Implementations of Strong Shared Memory Primitives,* in *Proc. of PODC* (1994).

[29] H. Attiya, E. Hillel, and A. Milani, *Inherent Limitations on Disjoint-access Parallel Implementations of Transactional Memory,* in *Proc of SPAA* (2009).

[30] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia, *Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations,* in *Proc. of PODC* (2015).

[31] L. Lamport, *Paxos made simple,* ACM SIGACT News (2001).

[32] B. M. Oki and B. H. Liskov, *Viewstamped Replication: A new primary copy method to support highly-available distributed systems,* in *Proc. of PODC* (1988).

[33] B. Liskov and J. Cowling, *Viewstamped Replication Revisited,* Tech. Rep. (MIT, 2012).

[34] D. Ongaro and J. Ousterhout, *In search of an understandable consensus algorithm,* in *Proc. of USENIX ATC* (2014).

[35] J. Rao, E. J. Shekita, and S. Tata, *Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore,* in *Proc. of VLDB* (2011).

[36] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, *Paxos Replicated State Machines as the Basis of a High-Performance Data Store,* in *Proc. of NSDI* (2011).

[37] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing Scalable, Highly Available Storage for Interactive Services,* in *Proc. of CIDR* (2011).

[38] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, *The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors,* in *Proc. of SOSP* (2013).

[39] L. Lamport, *Generalized Consensus and Paxos*, Tech. Rep. 2005-33 (Microsoft Research, 2005).

[40] I. Moraru, D. G. Andersen, and M. Kaminsky, *There is more consensus in Egalitarian parliaments,* in *Proc. of SOSP* (2013).

[41] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, *Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks,* Proc. of SIGMOD (1995).

[42] Y. Mao, E. Kohler, and R. Morris, *Cache Craftiness for Fast Multicore Key-value Storage,* in *Proc. of EuroSys* (2012).

[43] H.-T. Kung and J. T. Robinson, *On optimistic methods for concurrency control,* ACM Transactions on Database Systems (1981).

[44] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, *Building Consistent Transactions with Inconsistent Replication (extended version)*, Tech. Rep. 2014-12-01 v2 (University of Washington CSE, 2015) available at http://syslab.cs.washington.edu/papers/tapir-tr-v2.pdf.

[45] E. Michael, D. R. K. Ports, N. K. Sharma, and A. Szekeres, *Recovering Shared Objects Without Stable Storage,* in *Proc. of DISC* (2017).

[46] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou, *KuaFu: Closing the Parallelism Gap in Database Replication,* in *Proc. of ICDE* (2013).

[47] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, *Calvin: Fast Distributed Transactions for Partitioned Database Systems,* in *Proc. of SIGMOD* (2012).

[48] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, *All About Eve: Execute-verify Replication for Multi-core Servers,* in *Proc. of PLDI* (2012).

[49] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking cloud serving systems with YCSB,* in *Proc. of SOCC* (2010).

[50] C. Leau, *Spring Data Redis – Retwis-J,* (2013), http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[51] MySQL, *MySQL,* https://www.mysql.com/.

[52] postgresql, *PostgreSQL,* http://www.postgresql.org/.

[53] Microsoft SQLServer, *Microsoft SQLServer,* https://www.microsoft.com/en-us/sql-server/default.aspx.

[54] D. Qin, A. Goel, and A. D. Brown, *Scalable Replay-Based Replication For Fast Databases,* in *Proc. of VLDB* (2017).

[55] J. Li, E. Michael, and D. R. K. Ports, *Eris: Coordination-free consistent transactions using in-network concurrency control,* in *Proc. of SOSP* (2017).

[56] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, *An empirical evaluation of in-memory multi-version concurrency control,* in *Proc. of VLDB* (2017).

[57] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, *Paxos Made Transparent,* in *Proc. of SOSP* (2015).

[58] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, *Rex: Replication at the Speed of Multi-core,* in *Proc. of EuroSys* (2014).

[59] T. D. Chandra, R. Griesemer, and J. Redstone, *Paxos made live: An engineering perspective,* in *Proc. of PODC* (2007).

[60] E. Pinheiro, W.-D. Weber, and L. A. Barroso, *Failure trends in a large disk drive population,* in *Proc. of FAST* (2007).

[61] H. Attiya, A. Bar-Noy, and D. Dolev, *Sharing memory robustly in message-passing systems,* J. of the ACM **42**, 124 (1995).

[62] K. M. Konwar, N. Prakash, N. A. Lynch, and M. Médard, *RADON: Repairable atomic data object in networks,* in *Proc. of OPODIS* (2016).

[63] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, *JPaxos: State machine replication based on the Paxos protocol,* Tech. Rep. EPFL-REPORT-167765 (2011).

[64] R. Oliveira, R. Guerraoui, and A. Schiper, *Consensus in the Crash Recover Model,* Tech. Rep. TR-97/239 (EPFL, Lausanne, Switzerland, 1997).

[65] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, *Failure detectors in omission failure environments,* in *Proc. of PODC* (1997).

[66] M. Hurfin, A. Mostéfaoui, and M. Raynal, *Consensus in asynchronous systems where processes can crash and recover,* in *Proc. of SRDS* (1998).

[67] J. Y. Halpern and Y. Moses, *Knowledge and common knowledge in a distributed environment,* in *Proc. of PODC* (1984).

[68] M. K. Aguilera, W. Chen, and S. Toueg, *Failure detection and consensus in the crash-recovery model,* in *Proc. of DISC* (1998).

[69] R. Guerraoui, R. R. Levy, B. Pochon, and J. Pugh, *The collective memory of amnesic processes,* ACM Trans. Algorithms **4**, 12:1 (2008).

[70] L. Jehl, T. E. Lea, and H. Meling, *Replacement: Decentralized failure handling for replicated state machines,* in *Proc. of SRDS* (2015).

[71] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline, *Detection of mutual inconsistency in distributed systems,* in *IEEE Trans. on Software Engineering* (1983).

[72] C. J. Fidge, *Timestamps in message-passing systems that preserve the partial ordering,* in *Proc. of ACSC* (1988).

[73] M. Merritt and G. Taubenfeld, *Computing with infinitely many processes under assumptions on concurrency and participation,* in *Proc. of DISC* (2000).

[74] L. Lamport, D. Malkhi, and L. Zhou, *Reconfiguring a state machine,* SIGACT News **41**, 63 (2010).

[75] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira, *Dynamic reconfiguration of primary/backup clusters,* in *Proc. of USENIX ATC* (2012).

[76] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, *The SMART way to migrate replicated stateful services,* in *Proc. of EuroSys* (2006).

[77] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, *Dynamic atomic storage without consensus,* J. ACM **58**, 7:1 (2011).

[78] N. A. Lynch and A. A. Shvartsman, *RAMBO: A reconfigurable atomic memory service for dynamic networks,* in *Proc. of DISC* (2002).

[79] E. Gafni and D. Malkhi, *Elastic configuration maintenance via a parsimonious speculating snapshot solution,* in *Proc. of DISC* (2015).

[80] P. Musial, N. Nicolaou, and A. A. Shvartsman, *Implementing distributed shared memory for dynamic networks,* Communications of the ACM **57** (2014).

[81] L. Jehl, R. Vitenberg, and H. Meling, *SmartMerge: A new approach to reconfiguration for atomic storage,* in *Proc. of DISC* (2015).

[82] R. Baldoni, S. Bonomi, A.-M. Kermarrec, and M. Raynal, *Implementing a register in a dynamic distributed system,* in *Proc. of ICDCS* (2009).

[83] R. Baldoni, S. Bonomi, and M. Raynal, *Implementing a regular register in an eventually synchronous distributed system prone to continuous churn,* IEEE Trans. Parallel Distrib. Syst. **23**, 102 (2012).

[84] H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch, *Simulating a shared register in an asynchronous system that never stops changing,* in *Proc. of DISC* (2015).

[85] A. Klappenecker, H. Lee, and J. L. Welch, *Dynamic regular registers in systems with churn,* Theor. Comput. Sci. **512**, 84 (2013).

[86] N. A. Lynch and M. R. Tuttle, *An introduction to input/output automata,* CWI Quarterly **2**, 219 (1989).

[87] L. Lamport, *On interprocess communication,* Distributed Computing. Parts I and II (1986).

[88] L. Lamport, *Time, clocks, and the ordering of events in a distributed system,* Communications of the ACM (1978).

[89] L. Lamport, *Paxos made simple,* ACM SIGACT News 32 (2001).

[90] M. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects,* ACM Trans. Program. Lang. Syst. (1990).

[91] H. S. Gunawi, T. Do, A. Laksono, T. L. Mingzhe Hao, J. F. Lukman, and R. O. Suminto, *What bugs live in the cloud? A study of issues in scalable distributed systems,* USENIX ;login: (2015).

[92] *Android App Manifest Developer's Guide,* Google, `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.

[93] Apple, *Human interface guidelines: Requesting permission,* `https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/`.

[94] M. Soze, *Mark Zuckerbergs private photos leaked in Facebook security flaw,* `http://www.inquisitr.com/166086/mark-zuckerbergs-private-photos-facebook-security-flaw/` (2011).

[95] V. Blue, *Researcher: Snapchat names, aliases, phone numbers vulnerable,* http://www.cnet.com/news/researcher-snapchat-names-aliases-phone-numbers-vulnerable/ (2013).

[96] Linkedin, *2012 linkedin hack,* Wikipedia, https://en.wikipedia.org/wiki/2012_LinkedIn_hack.

[97] K. Notopoulos, *The Snapchat Feature That Will Ruin Your Life,* http://www.buzzfeed.com/katienotopoulos/the-snapchat-feature-that-will-ruin-your-life (2012).

[98] Federal Trade Commission, *Android flashlight app developer settles FTC charges it deceived consumers,* https://www.ftc.gov/news-events/press-releases/2013/12/android-flashlight-app-developer-settles-ftc-charges-it-deceived (2013).

[99] A. Henry, *Twitter is tracking you on the web; here's what you can do to stop it,* http://lifehacker.com/5911389/twitter-is-tracking-you-on-the-web-heres-what-you-can-do-to-stop-it (2012).

[100] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, *πbox: A platform for privacy-preserving apps,* in *Proc. of NSDI* (2013).

[101] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, F. Kaashoek, and H. Balakrishnan, *Building web applications on top of encrypted data using Mylar,* in *Proc. of NSDI* (2014).

[102] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, *Hails: Protecting Data Privacy in Untrusted Web Applications,* in *Proc. of PLDI* (2012).

[103] A. C. Myers and B. Liskov, *Protecting Privacy Using the Decentralized Label Model,* ACM Trans. Softw. Eng. Methodol. (2000).

[104] W. Cheng, D. R. K. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov, *Abstractions for Usable Information Flow Control in Aeolus,* in *Proc. of USENIX ATC* (2012).

[105] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, *Securing Distributed Systems with Information Flow Control,* in *Proc. of NSDI* (2008).

[106] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, *Protecting users by confining JavaScript with COWL,* in *Proc. of PLDI* (2014).

[107] Redis, *Redis: Open source data structure server,* (2013), http://redis.io/.

[108] B. Fitzpatrick, *Distributed caching with memcached,* Linux Journal (2004).

[109] Google Accounts, *Google Accounts,* https://accounts.google.com.

[110] OpenID, *OpenID,* https://openid.net/.

[111] M. Kumar, *Collection of 1.4 billion plain-text leaked passwords found circulating online,* Hacker News (2017), https://thehackernews.com/2017/12/data-breach-password-list.html.

[112] D. Schultz and B. Liskov, *IFDB: Decentralized information flow control for databases,* in *Proc. of EuroSys* (2013).

[113] S. Lee, D. Goel, E. L. Wong, and M. Dahlin, *A CleanRoom Approach to BYOA: Bring Your Own Apps*, Tech. Rep. (University of Texas at Austin, 2014).

[114] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, *CryptDB: protecting confidentiality with encrypted query processing,* in *Proc. of SOSP* (2011).

[115] W. A. Arbaugh, D. J. Farber, and J. M. Smith, *A secure and reliable bootstrap architecture,* in *Proc. of IEEE S&P (Oakland)* (Oakland, CA, USA, 1997).

[116] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, *Design and implementation of a TCG-based integrity measurement architecture,* in *Proceedings of the 13th USENIX Security Symposium (Security '04)* (San Diego, CA, USA, 2004).

[117] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, *Terra: A virtual machine-based platform for trusted computing,* in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (ACM, Bolton Landing, NY, USA, 2003).

[118] A. Baumann, M. Peinado, and G. Hunt, *Shielding applications from an untrusted cloud with Haven,* in *Proc. of PLDI* (2014).

[119] Amazon Lambda, webpage, https://aws.amazon.com/lambda/.

[120] amazon, *Amazon s3,* https://aws.amazon.com/s3/.

[121] D. E. Denning and P. J. Denning, *Certification of programs for secure information flow,* Commun. ACM **20**, 504 (1977).

[122] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, *Labels and Event Processes in the Asbestos Operating System,* in *Proc. of SOSP* (2005).

[123] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, *Information flow control for standard os abstractions,* in *Proc. of SOSP* (2007).

[124] Mobile OS Statistics, *Global mobile os market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018,* Statista (2018), https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[125] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, *Clickjacking: Attacks and defenses,* in *Proc. of the USENIX Security Symposium* (2012).

[126] F. Roesner and T. Kohno, *Securing embedded user interfaces: Android and beyond,* in *Proceedings of the USENIX Security Symposium* (2013).

[127] F. Roesner, J. Fogarty, and T. Kohno, *User interface toolkit mechanisms for securing interface elements,* in *Proc. of the ACM Symp. on User Interface Software and Technology* (2012).

[128] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, *What the App is That? Deception and Countermeasures in the Android User Interface,* in *Solar Phys.* (2015).

[129] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, *TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,* in *Proc. of PLDI*, OSDI'10 (2010).

[130] S. Rosen, Z. Qian, and Z. M. Mao, *AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users,* in *Proc. of CODASPY* (2013).

[131] J. Clause, W. Li, and A. Orso, *Dytan: A Generic Dynamic Taint Analysis Framework,* in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)* (London, United Kingdom, 2007).

[132] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, *DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation,* in *Proceedings of NDSS* (2011).

[133] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati, *Spandex: Secure password tracking for android,* in *Proc. of USENIX Security Symposium* (2014).

[134] U. Maheshwari, R. Vingralek, and W. Shapiro, *How to build a trusted database system on untrusted storage,* in *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI '00)* (USENIX, San Diego, CA, USA, 2000).

[135] C. Weinhold and H. Härtig, *VPFS: Building a virtual private file system with a small trusted computing base,* in *Proceedings of the 3rd ACM SIGOPS EuroSys (EuroSys '08)* (ACM, Glasgow, Scotland, United Kingdom, 2008).

[136] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, *Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems,* in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)* (ACM, Seattle, WA, USA, 2008).

[137] C. Weinhold and H. Härtig, *jVPFS: Adding robustness to a secure stacked file system with untrusted local storage components,* in *Proceedings of the 2011 USENIX Annual Technical Conference* (USENIX, Portland, OR, USA, 2011).

[138] Dropbox, *Dropbox,* (2015), http://www.dropbox.com.

[139] R. C. Merkle, *Secrecy, authentication, and public key systems,* Ph.D. thesis, Stanford University, Stanford, CA, USA (1979).

[140] *Soot: A framework for analyzing and transforming Java and Android Applications,* Sable Research Group, https://sable.github.io/soot/.

[141] J. Bell and G. Kaiser, *Phosphor: Illuminating dynamic data flow in commodity jvms,* in *Proc. of OOPSLA* (2014).

[142] OWASP Top 10 Application Security Risks 2013, https://www.owasp.org/index.php/Top10.

[143] OWASP Top 10 Mobile Risks 2014, https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[144] Twimight, *Twimight open-source Twitter client for Android,* (2015), http://code.google.com/p/twimight/.

[145] Xabber, *Xmpp chat client for android,* http://www.xabber.com (2015).

[146] Openfire, *Xmpp chat server,* http://www.igniterealtime.org/projects/openfire/ (2015).

[147] MinnieTwitter, *Minnietwitter open-source Twitter server,* (2015), https://github.com/UWSysLab/Sapphire/tree/master/example_apps/MinnieTwitter.

[148] acal, *aCal Android CalDav Client,* http://acal.me/wiki/Main_Page.

[149] 1and1, *Cosmo calendar server,* https://github.com/1and1/cosmo.

[150] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, *Sok: Lessons learned from android security research for appified software platforms,* in *Proc. of IEEE S&P (Oakland)* (2016).

[151] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, *User-driven access control: Rethinking permission granting in modern operating systems,* in *Proceedings of the IEEE Symposium on Security and Privacy* (2012).

[152] J. Kannan, P. Maniatis, and B.-G. Chun, *Secure data preservers for web services,* in *Proceedings of the 2nd USENIX Conference on Web Application Development* (2011).

[153] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, F. Roesner, A. Krishnamurthy, and T. Anderson, *Radiatus: a shared-nothing server-side web architecture,* in *Proc. of SOCC* (ACM, 2016).

[154] C. Dwork, *Differential privacy,* in *Proceedings of the 33rd International Conference on Very Large Data Bases (ICALP '06)* (Venice, Italy, 2006).

[155] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, *RIFLE: An architectural framework for user-centric information-flow security,* in *Proc. of the 37th Annual IEEE/ACM Int. Symposium on Microarchitecture* (2004).

[156] F. Wang, R. Ko, and J. Mickens, *Riverbed: Enforcing user-defined privacy constraints in distributed web services,* in *Proc. of NSDI* (2019).

[157] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, *Making Information Flow Explicit in HiStar,* in *Proc. of PLDI* (2006).

[158] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, *Fabric: A Platform for Secure Distributed Computation and Storage,* in *Proc. of SOSP* (2009).

[159] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, *Laminar: Practical fine-grained decentralized information flow control,* in *Proc. of PLDI* (2009).

[160] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, *Improving Application Security with Data Flow Assertions,* in *Proc. of SOSP* (2009).

[161] M. Kranch and J. Bonneau, *Upgrading https in mid-air: An empirical study of strict transport security and key pinning.* in *Proc. of NDSS* (2015).