Towards a Flexible, High-Performance Operating System for Mobile/Cloud Applications

Irene Y. Zhang

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

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Henry M. Levy, Chair

Arvind Krishnamurthy

Luis Ceze

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Towards a Flexible, High-Performance
Operating System for Mobile/Cloud Applications

Irene Y. Zhang

Chair of the Supervisory Committee:
Director Henry M. Levy
Computer Science & Engineering

The convergence of ubiquitous mobile devices, large-scale cloud platforms and pervasive network connectivity have changed the face of modern user applications. Unlike a traditional desktop application, which runs on a single machine and supports a single user, the typical user-facing application today spans numerous mobile devices and cloud servers while supporting large numbers of users. This shift significantly increased the difficulty of building new user applications. Programmers must now confront challenges introduced by distributed deployment (e.g., partial failures), new mobile/cloud application features (e.g., reactivity), and new mobile/cloud requirements (e.g., scalability).

This thesis proposes a new type of mobile/cloud operating system designed to meet the evolving needs of modern applications. Mobile/cloud applications are the standard applications of the future; thus, they deserve a first-class operating system that simplifies their development and run-time management. Our key contribution is the design, implementation and evaluation of three systems that together form the basis for a new mobile/cloud operating system: (1) Sapphire, a new distributed run-time and process management system, (2) Diamond, a new distributed memory management system, and (3) TAPIR, a new distributed storage system. Each system introduces new operating systems abstractions and mechanisms designed to eliminate the challenges and simplify the development of mobile/cloud applications. We demonstrate that, like operating systems of the past, these systems make it easier for programmers to build bigger and more complex applications.

Table of Contents

List of Figures

Glossary

FAULT-TOLERANCE: The ability to handle failures without losing data.

AVAILABILITY: The ability to operate normally when machines or network links are down. Also, frequently defined as the percentage of total operation time that a system is able to operate normally regardless of hardware downtime.

PERSISTENCE: The ability to never lose or corrupt data in the face of system failures or reboots.

REPLICATION: To use copies of data items, typically for fault-tolerance or availability.

CACHING: Placing copies of data objects in lower latency storage for better performance.

CODE OFFLOADING: To execute application code on a different machine with more processing power for better performance.

SYSTEM MODEL: The placement and use of computers in the system.

CONCURRENCY: The ability to shared data items from more than one thread of execution.

COHERENCE MODEL: The rules defining when a replicated system's behavior can diverge from that of an ideal, unreplicated system for accesses to a single data item.

CONSISTENCY MODEL: The rules defining when a replicated system's behavior can diverge from that of an ideal, unreplicated system for accesses to all of the system's data items.

ISOLATION MODEL: The rules defining when the behavior of a database executing multiple transactions concurrently can diverge from the behavior of an ideal database that executes all transactions serially.

REACTIVITY: The ability to automatically propagate changes across the system on any change to the system inputs.

Acknowledgments

This thesis would not have been possible without many, many people in my life. I would like to begin by thanking my batchmates: Jialin Li, Adriana Szekeres and Naveen Sharma. None of us could have anticipated this journey when we all started it together, but we all knew we were in it together and now the end is in sight!

None of this work would have been possible without my numerous collaborators: Adriana Szekeres, Naveen Sharma, Niel Lebeck, Brandon Holt, Ray Cheng, Simon Peter, Pedro Fonseca, James Bornholt, Doug Woos and Mothy Roscoe. I am honored to have worked with three amazing undergraduates: Dana Van Aken, Isaac Ackerman and Ariadna Norberg. The other students in our lab that made my day-to-day life enjoyable and kept me motivated and excited about research: Katelin Bailey, Pete Hornyack, Jialin Li, Antoine Kaufmann, Ellis Michael, Kaiyaun Zhang, Helgi Sigurbjanarson, Helga Gudmundsdottir, Qiao Zhang, Danyang Zhuo, Yuchen Jin, Vincent Liu, Haichen Shen and Seungyeop Han. I can easily say that I value these relationships much more than any other accomplishments during my PhD.

I would like to thank Mel Kadenko for making my life easier with her ability to reimburse (almost) anything and providing distraction when needed. Hal Perkins always provided a friendly face and candy bar every time I walked to Arvind's office and didn't find him. This thesis would not be the amazing document that it is without the help of Sandy Kaplan, who has read and touched every word.

I'll end by thanking the many advisors in my life, beginning with Hank and Arvind. Advising any student is not easy and I'm sure I was not just any student. I would not have started this journey

without support from my M. Eng. advisors, Frans Kaashoek and Jeremy Stribling. I would not have finished this journey without help from the other systems faculty at the University of Washington: Tom Anderson, Luis Ceze, Magda Balazinska, Ed Lazowska, Xi Wang, Franzi Roesner and Mark Oskin. Although Lindsay Michimoto was not an official academic advisor, I credit her with completely changing my PhD experience.

Finally, I have to thank my family. My husband, Dan Ports, is, as always, my closest mentor, collaborator and cheerleader. My parents have always pushed me do what would make me happy and never valued me based on my accomplishments. My in-laws, Catherine Covey and Tom Ports, have always taken equal pride in my achievements as their own boys.

In the end, I acknowledge my mom for my pursuit of elegance in both software and life.

Dedication

to my husband and best friend, Dan

1 Overview

Over the last decade, three hardware trends and a "killer app" have converged to revolutionize the face of everyday applications. First, the low-cost and widespread availability of small but powerful *mobile devices* (e.g., smartphones, tablets) offers users a portable computing platform with easy access to countless applications. Next, web companies, like Google [78] and Amazon [12], leveraged cheap commodity hardware to build *cloud computing* platforms that are always available and powerful enough to support millions of users. Finally, cellular networks developed data capabilities, and wireless networks became plentiful and cheap, resulting in almost *pervasive network connectivity* between mobile devices and the cloud.

Combined, these technologies gave rise to the killer app of the mobile/cloud revolution: social networks. Facebook [67], Twitter [208] and similar applications [211, 101, 152, 185] provide a platform for users to publish and share information. Mobile devices give users access to their apps on the go, so users can share information about their lives as they experience them. Cloud services make this information constantly available, and pervasive network connectivity keeps users connected. As a result, social apps became a way for users to not only share, but to interact, in real time.

Today, social interactivity is not limited to social networks; it has permeated everyday applications. For example, social games (e.g., Words With Friends [217]), where users play games with friends in real time, have become a multi-billion dollar industry [63]. Productivity and organizational applications (e.g., Trello [203]) are changing how users collaborate remotely by providing sophisticated communication channels in real time. Google Docs [81] adds social interaction to a traditional desktop application – a document editor – by leveraging the mobile/cloud environment. It and similar apps

(e.g., Microsoft Word Online [142], Keynote Live [102]) let users interactively edit the same document at the same time, access their documents from any device at any time, and avoid worrying about lost data or limited storage capacity.

1.1 The Mobile/Cloud Revolution

Today, mobile, web and desktop applications have blurred together so much that mobile/cloud have become the norm; so much so that they have affected the behavior of an entire generation [204]. As a result, the average user-facing application has acquired the following new characteristics:

1. **A new distributed, heterogenous and wide-area *deployment*.** Unlike past user applications that run on a single machine, mobile/cloud applications span mobile devices and cloud servers with vastly different computational and networking capabilities. As a result, they must cope with performance variability, partial failures and unavailability, and hundred-millisecond communication delays.

2. **New multi-user, interactive and real-time sharing *features*.** Unlike past user applications that support a single user, mobile/cloud applications support multiple users. These users interact through the application in complex ways (e.g., playing a word game, collaborating on a document). As a result, mobile/cloud applications must coordinate synchronized access to shared data between users in real time.

3. **New longevity, reliability and persistence *guarantees*.** Unlike past user applications that store persistent state in a file system, mobile/cloud applications do not present a strong separation between run-time and stored state to users; for example, Facebook users have no concept of files or a save button. Users expect the application to run forever, never crash and never lose their data.

These characteristics add up to more complexity for application programmers, as we explore in Section 1.3. They introduce a new set of application requirements (e.g., availability, scalability) that application programmers have never faced before. Worse, no general-purpose systems currently exist

to help application programmers because these characteristics were previously limited to specialized applications (e.g., massive multi-player online games) and distributed systems, which are both typically built by experts. As a result, today's mobile/cloud programmers either implement custom solutions (requiring distributed systems expertise) or cobble together ad hoc solutions using existing systems that meet a limited set of their requirements.

This thesis addresses the wide gap in the systems design space. We first make the case for a new type of operating system (OS) designed to meet the needs of mobile/cloud applications. We examine application requirements for such an OS and discuss why existing state-of-the-art mobile/cloud systems fail to meet them. Finally, we describe the design, implementation and evaluation of three systems that together comprise the basis for a new mobile/cloud OS.

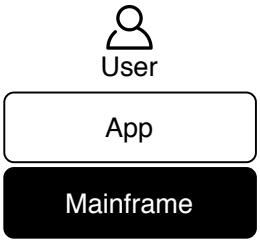
1.2 The Case for a Mobile/Cloud Operating System

Figure 1.1 shows the progression of system architectures from the first mainframe environment (Figure 1.1a) to today's mobile/cloud environment (Figure 1.1d). Early application programmers faced a world similar to today's mobile/cloud developers: they built applications directly on machine hardware and were responsible for tackling the challenges of their computing platform using ad-hoc collections of systems and libraries. For example, programmers built custom applications for specific hardware and coordinated computing resources with other applications and libraries. They also organized persistent storage to ensure that application state was not lost on failure.

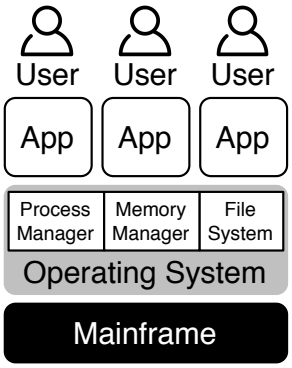
Operating systems were developed to address the growing variety and complexity of hardware and applications. As shown in Figure 1.1b, the operating system runs between the machine hardware and user-facing applications and has three important components[1]: (1) a process manager that handles the application's run-time execution, (2) a memory manager that handles in-memory application state and (3) a file system that manages persistent storage. Table 1.1 lists each component's functions and the abstractions for applications using those functions.

With these three components, operating systems significantly simplified application development

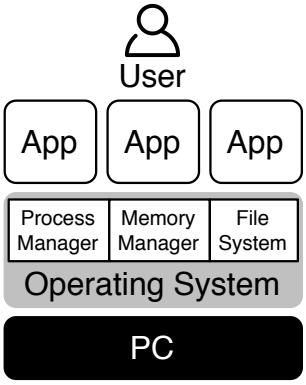
[1]The OS typically also manages I/O to external entities and provides protection and isolation; both are beyond the scope of this thesis.



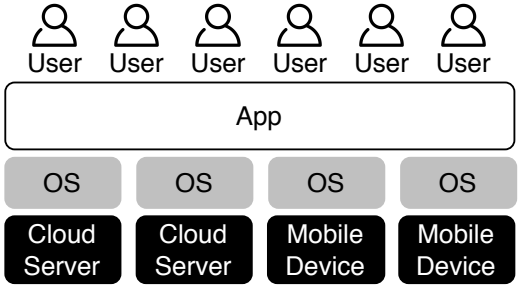
(a) The first mainframes ran applications directly on hardware. Each application supported a single user.



(b) Operating systems let mainframes run more than one application at a time. Each application still supported a single user.



(c) PCs were designed for a single user, but their operating systems let them run multiple applications at the same time.



(d) Mobile/cloud applications run across multiple mobile devices and cloud servers and support many users.

Figure 1.1: *The evolution of application environments and system architectures.*

on mainframes. As a result, programmers could more quickly and easily build complex applications, making operating systems increasingly important into the next era.

The 1980s and 1990s saw mainframes and minicomputers give way to smaller and cheaper personal computers (PCs). These compact machines supported a single user with many applications, as shown

Table 1.1: *Classic operating system components.* For each component, this table lists the set of functions and the abstractions provided by the component to simplify applications.

OS Component	Component Functions	OS Abstractions
Process Manager (Run-time)	• Deploy and schedule application code • Abstract low-level hardware • Mediate communication and coordination between applications	• processes, threads • sockets, file descriptors • inter-process communication (IPC) pipes, signals
Memory Manager	• Provide a uniform address space • Manage allocation and scheduling • Create and synchronize memory mappings	• virtual memory • malloc, swap-to-disk • shared memory, memory-mapped files
File System (Storage Manager)	• Abstract low-level disk interface • Mediate concurrent accesses • Handle crashes and recover from failures	• files, directories • atomicity, locking • fsync, journaling

in Figure 1.1c. While mainframes and minicomputers were largely constrained to labs in work environments, PCs became ubiquitous devices on office and home desktops, leading to an explosion in the types and complexity of applications available. While hardware and applications changed dramatically, the operating systems remained basically the same in terms of their functions and abstractions. They gained importance, however, because they let programmers who were not systems experts build complex, user-facing applications. For the first time, anyone owning a computer could build simple programs.

Figure 1.1d shows the computing environment and system architecture for a mobile/cloud application. The most striking difference is that the user-facing application is *distributed* for the first time.

Each mobile device and cloud server continues to run an OS, but no OS components span the entire user application.

In lieu of an OS, programmers have developed many new libraries and run-time systems for mobile/cloud applications, which we review in Section 1.4. However, these existing systems are still limited: they span only a portion of the mobile/cloud application, cover a subset of OS responsibilities, or work for only a class of applications. Because no existing systems provide general-purpose abstractions across the entire mobile/cloud application, programmers must create a patchwork of systems to meet their needs and mediate among them. More often than not, these systems work together poorly because there are so many existing systems.

Instead of this makeshift approach, we advocate for the design of a new operating system to meet the end-to-end requirements of mobile/cloud applications. Such applications have clearly established themselves as the preeminent class of future user applications, and a dedicated OS is necessary to simplify their programming.

1.3 Mobile/Cloud Challenges and Application Requirements

This thesis introduces three new systems that form the basis of a new mobile/cloud operating system. Section 1.1 noted the three characteristics that differentiate mobile/cloud applications from traditional applications: their new deployment, new features and new guarantees. In this section, we explore how those characteristics lead to new programming challenges, which will dictate the requirements for our new OS.

1.3.1 Mobile/Cloud OS Requirements

The three mobile/cloud characteristics lead to a set of new challenges for programmers when building mobile/cloud applications. Table 1.2 maps these characteristics to new programming challenges. These challenges, in turn, dictate the requirements for our mobile/cloud OS because the goal of our new OS is to help programmers cope with their programming challenges.

The first two programming challenges and OS requirements are a consequence of distributed, heterogenous and wide-area deployment. Partial failures and network partitions occur in any distributed

Table 1.2: *Mobile/cloud application characteristics, challenges and OS requirements.* Each mobile/cloud characteristic leads to a set of new application challenges, which dictate the requirements for a mobile/cloud OS.

Characteristic	New Programming Challenge	OS Requirement
Deployment	Remain usable with unreachable nodes and network partitions	Availability
	Respond with low latency despite variable and limited performance	Responsiveness
Features	Support many users accessing the app concurrently	Scalability
	Allow users to access shared data without conflicts	Consistency
Guarantees	Ensure data is not lost on crashes and failures	Fault-tolerance
	Automatically save and propagate updates	Reactivity

environment, especially one which spans wide-area networks. At the same time, mobile/cloud users expect to have continuous access to their applications, so programmers must ensure that their applications is *highly available*. Mobile devices have limited computational resources and communication across the wide-area is expensive. As a result, programmers must deliver a *responsive*, low-latency user experience under varying computational and communication latencies.

The next two challenges and requirements stem from new multi-user, interactive sharing features. Large numbers of users may access a mobile/cloud application at the same time from different devices. Therefore, the application must be designed to *scale* to many devices and servers. Interacting users will naturally access shared data, so they must receive a *consistent*, conflict-free view even if many users are concurrently reading and writing.

The final two challenges and requirements arise from new longevity, reliability and persistence guarantees. Users do not expect desktop applications to survive crashes without losing data; however, users do not understand failures of cloud servers that they cannot see. As a result, users expect mobile/cloud applications to run forever and never lose their data. Programmers must therefore build

applications to be *fault-tolerant*. Without a boundary between run-time and persistent application state, users expect applications to automatically propagate their updates throughout the application and to persistent storage, so programmers must also build their applications to be *reactive*.

1.3.2 The Challenge of Designing a Mobile/Cloud OS

Classic operating systems meet the requirements of most applications with a single design because the desktop environment is relatively resource-rich. For example, the page-based virtual memory mechanism works for all but a few of today's large memory applications because accessing memory is a low latency operation and coordinating virtual memory remains cheap. However, the mobile/cloud environment is considerably more constrained: communication latencies across the wide-area are high, computational resources are limited on mobile devices and throughput between mobile devices and cloud servers is low.

Due to these constraints, mobile/cloud applications often require different solutions to their application requirements, and the choice between solutions can be application-specific. For example, there several ways to achieve fault-tolerance, like replicating application processes or continuously synchronizing updates storage. Synchronization is a better solution if an application has very little run-time state, but replication is better in applications with more run-time state or rapidly changing run-time state.

Application programmer may also have to choose between meeting one requirement or another. For example, replicating data to the cloud increases the fault-tolerance and availability of the application at the cost of responsiveness. Applications with important data (e.g., banking and finance apps) would choose the former, while others (e.g., games) might choose the latter.

These trade-offs make it challenging to design a general-purpose, mobile/cloud operating system. The different solutions add needed functions to one or more management components in our new OS. For example, replicating processes would be a new run-time management function, while synchronization between memory and disk is a memory management function.

Table 1.3 presents examples of new management functions that a new mobile/cloud OS could potentially implement to help programmers build applications that meet each requirement. This

Table 1.3: Examples of functions a mobile/cloud OS might provide in each component to help programmers meet the six mobile/cloud requirements.

Requirement	Run-time Manager	Memory Manager	Storage Manager
Availability	Auto-restart on crash	Auto-sync w/ storage	Replication
Responsiveness	Automatic process migration	In-memory caching	Storage caching
Scalability	Automatic process spin-up	In-memory caching	Partitioning
Consistency	Distributed locks	Atomic memory operations	Transactions
Fault-tolerance	Periodic process checkpoint	Auto-sync w/ storage	Log to disk
Reactivity	Notifications	Sync across address spaces	Triggers

table is not meant to be comprehensive; more than one function can help programmers achieve a requirement even within one management component (e.g., many run-time management strategies can achieve availability). As a result, a general-purpose operating system must be flexible enough to support all of these application-specific functions. As we will see later, this flexibility constitutes the most important design goal in every part of our new OS.

1.4 Existing Mobile/Cloud Systems

Few existing systems meet the myriad requirements and needs of today's mobile/cloud applications. Nearly all lack flexibility; instead, they implement a single solution to a subset of the mobile/cloud requirements discussed previously. As a result, existing systems can support only one class of applications or part of an application. Programmers are must piece together systems to meet some requirements and implement custom solutions for the rest.

This section reviews existing systems and their uses. Table 1.4 summarizes the classes of systems that we cover, along with the requirements met and any application limitations. Figure 1.2 plots each

Table 1.4: *Evaluation of existing mobile/cloud application systems.* We list the classes of existing mobile/cloud systems. For each system, we evaluate how many of the mobile/cloud requirements the system meets and note any limitations on the application.

System Type	Requirements Met	Limitations
Wide-area Storage [62, 61]	Availability, Responsiveness	Limited app state and computation
	Scalability, Fault-tolerance	Limited app state and computation
	Consistency	Requires app-level locking
	Reactivity	Requires polling
Cloud Storage[22, 56]	Availability	Requires app restart cloud servers
	Scalability	Requires app scale cloud servers
	Responsiveness	Limited to server-side code
	Consistency	Requires app-level locking
Lock Service [35]	Consistency	Requires app acquire locks
Notification Service [4, 15]	Consistency	Requires app-level coordination
	Reactivity	Requires app publish and subscribe
Code-offloading [52, 177]	Responsiveness	Limited data for computation
Virtual Machines [9, 80]	none	none
Containers [19, 18]	Availability	Takes time to restart
	Scalability	Requires app spin-up containers
Lambda Functions [11, 82]	Availability, Scalability	Limited programming stack
App Engines [79, 17]	Availability, Scalability, Fault-tolerance	Limited programming stack and app state
Backend-as-a-Service [163, 69]	Availability, Responsiveness	Limited app state or limited app class
	Scalability, Fault-tolerance	Limited app state or limited app class
	Consistency	Weak or none
	Reactivity	Requires polling or pub/sub

system class based on the number of requirements met and the range of applications supported.

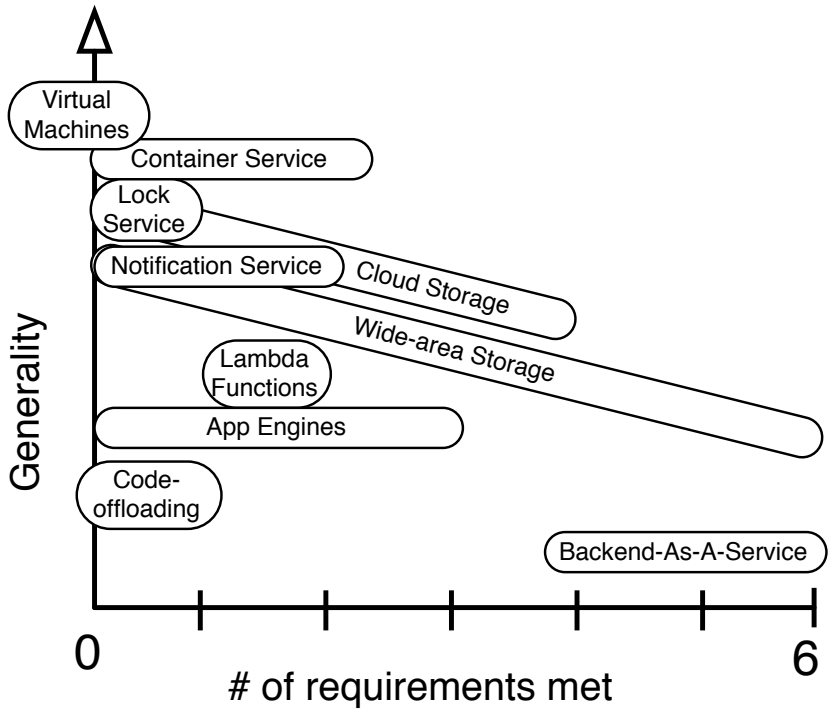


Figure 1.2: *Evaluation of existing systems.* We graph each class of systems based on how many mobile/cloud requirements it meets. If the system gives applications control over the requirement or requires applications help meet the requirement, then we draw the requirements as a range between the number of requirements that the application can choose to meet. Note that the more general-purpose systems meet fewer requirements, while the less general ones meet more. This trade-off leaves programmers with a difficult choice since no systems are both general-purpose and meet all of their requirements.

1.4.1 Distributed Storage Systems

Distributed storage systems have become crucial components for mobile/cloud applications because they meet many mobile/cloud requirements. For example, a wide-area storage system like Drop-

box [62] or Google Drive [61] eliminates or simplifies all mobile/cloud requirements; however, the programmer must implement the application as a completely stateless mobile client. Such an application would meet the six mobile/cloud requirements because it would: (1) be available when it can read and write to the storage system, (2) be responsive when it has low latency access to storage, (3) rely on the storage system to scale to a large number of clients, (4) use a simple technique for consistency and concurrency control (e.g, a lock file to avoid conflict), (5) continuously checkpoint to storage for fault-tolerance, and (6) rely on periodic polling to reactively propagate updates. In this way, as shown in Figure 1.2, a wide area storage system completely meets three requirements and simplifies the others.

The disadvantage of this application design is poor performance given either large amounts of application state to read and write on each request or an application that needs more computation than the mobile client can support. As a result, we mark wide-area storage systems as less general in Figure 1.2. Nevertheless, these systems provide a popular solution for many types of applications, like recipe managers [158, 24, 151], note taking [66, 184] and journaling apps [59, 54, 87], and to-do lists [200, 2, 218].

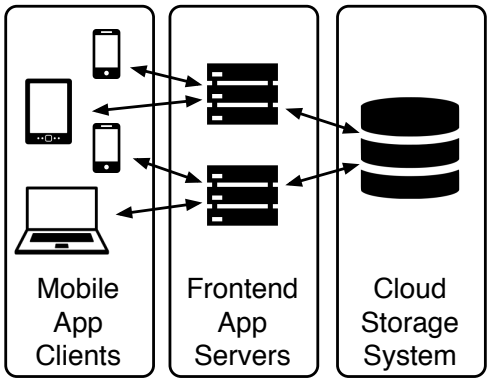


Figure 1.3: *A simple client-server architecture.* Both client and server code are typically stateless and idempotent to facilitate retries and recovery on failures.

Of course, not all mobile/cloud applications can be implemented with stateless mobile clients; for example, applications may have heavy computational or security needs that require some components

to run on the cloud. Figure 1.3 shows a simple architecture for a mobile/cloud application using a cloud-based distributed storage system (e.g., Dynamo [56], BigTable [40], Spanner [48]). The application consists of a client-side component, which runs on mobile devices, and a server-side component, which runs on cloud servers. Many of today's popular mobile/cloud applications use this architecture at their core, including Facebook [67], Twitter [207] and Amazon [12].

While cloud storage systems simplify the task of meeting mobile/cloud requirements for range of applications (marked as general purpose in Figure 1.2), they leave significant work for application programmers. Because both client and server components are typically stateless, the application must still re-start servers (for availability), cache locally on mobile clients (for responsiveness), spin up more servers (for scalability), coordinate between clients and servers (for consistency), checkpoint both clients and servers (for fault-tolerance), and poll servers (for reactivity). Simply stated, building a mobile/cloud application using a cloud storage system is the modern-day equivalent of building a desktop application with only a file system.

1.4.2 Service-Oriented Architectures

Since traditional distributed storage systems leave significant responsibilities unmanaged, application programmers and systems researchers have developed new types of distributed systems to more completely meet the needs of mobile/cloud applications. Examples include lock services (e.g., Chubby [35], ZooKeeper [98]), distributed graphs (TAO [33], FlockDB [165]), queue services (Starling [192], Kafka [114]) and notification/pub-sub services (e.g., Thialfi [4]). As Figure 1.4 shows, applications use these services to help implement specific functionality (e.g., concurrency control and push notifications). As a result, the services typically help with only one mobile/cloud requirement and do not eliminate all work needed to achieve that requirement. For example, lock services (Figure 1.2) help applications provide consistency by avoiding conflicts, but programmers must still call the lock service at appropriate times to achieve consistency. Likewise, notification services help applications efficiently achieve reactivity by eliminating the need for polling; however, programmers must still send and subscribe to notifications correctly to propagate updates to other users.

Applications using these systems with narrow functionality are said to have a *service-oriented*

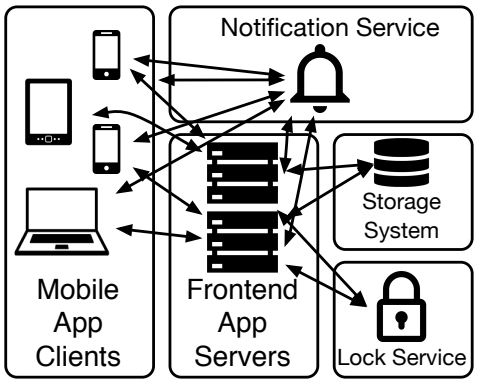


Figure 1.4: *A service-oriented architecture.* Unlike most services, notification services [4, 15] directly connect to mobile clients rather than interfacing through the application front-end. Note the large amount of coordination across services, which the application is left to manage.

architecture. Twitter and Amazon are popular examples [94, 95]. For example, Twitter implements separate services for storing tweets (T-store [93]), distributing tweets (Firehose [94]), issuing unique IDs (Snowflake [110]), and maintaining its social graph (Flock [165]). The benefits of a service-oriented architecture are its strong boundaries between application components and ability to make those service available to the public, which both Amazon and Twitter leverage. However, each application service/component must re-implement solutions to all mobile/cloud requirements, and it is often difficult for programmers to reason about cross-service coordination. Even with a correct implementation, achieving many mobile/cloud requirements may prove difficult; for example, it requires a complex and expensive two-phase commit protocol to achieve the correct fault-tolerant behavior.

1.4.3 Distributed Run-time Systems

While distributed storage systems and service-oriented architectures offer useful tools and functionality, programmers must still run and manage the execution of application components. For example, they must detect failures and re-start servers (for availability) and monitor load and spin up more servers (for scalability). Distributed run-time systems help programmers meet these challenges.

We divide existing run-time systems into mobile and cloud solutions. Most mobile run-time systems are code-offloading systems [85, 52, 42], which automatically move computationally intensive mobile code execution to more powerful platforms for better performance or battery usage. These systems remain in the research realm because the growing computational capabilities of mobile devices are eroding the benefits of moving computation to the cloud.

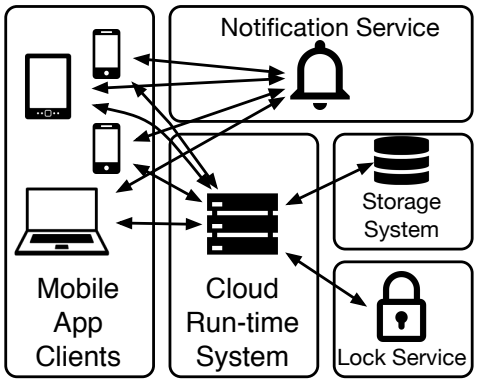


Figure 1.5: *A cloud run-time architecture.* Systems like Google App Engine [79] and Amazon Lambda [11] manage execution of the application's server-side computation, automatically scaling and restarting code as necessary. They require stateless application code but provide access to a distributed storage system and other services.

Figure 1.5 shows how cloud run-time systems enable application programmers to easily execute application logic run in the cloud. These systems vary based on flexibility in the programming stack (e.g., OS, run-time libraries, programming language, etc.) and the amount of run-time management they provide. For example, EC2 [9] and services like it (e.g., Google Compute Engine [80], Azure VM) run virtual machines that let programmers control the entire programming stack. However, they offer little to no run-time management, leaving programmers to detect failures and restart VMs (for availability) and monitor load and spin up more VMs (for scalability).

Amazon's EC2 container service [19], Google's Container Engine [80] and Azure Containers [18] support lightweight containers that let programmers customize the full programming stack with less overhead. Google's public container management service, Kubernetes [34], provides basic run-time

management; it detects and restarts crashed containers and provides load monitoring, but it leaves the application to spin up more containers. It provides no fault-tolerance and meets no other mobile/cloud requirements. Overall, Kubernetes offers a bare-bones run-time management solution because allocating and starting a container is a high-overhead operation.

Amazon Lambda [11] – along with Google and Azure Functions [82, 20] – offer stateless hooks that programmers can register and run. However, they restrict the programming language and offer little to no control over run-time libraries and the OS. These services dynamically scale based on function invocations (for scalability) and restart functions if they do not complete (for availability), but they provide no support for the other four requirements.

Google App Engine [76], Azure App Service [17], and Amazon's Elastic Beanstalk [10] combine function services with persistent storage for stateful server-side application code. While application programmers must still checkpoint to storage, these systems will detect failures and automatically recover (for availability and fault-tolerance) and scale to meet load. They provide a complete suite of run-time management services for application logic run in the cloud provided that applications do not have special needs that require a more customized deployment, for example, geo-distributed placement for performance or other reasons. Nevertheless, these systems power many popular apps, including Snapchat [185] and Spotify [189].

1.4.4 Backend-as-a-Service

Systems that provide *backend-as-a-service* (BaaS) have become a new trend in mobile/cloud development. As shown in Figure 1.6, BaaS systems encapsulate all cloud-side computation and storage into a general API. Some BaaS systems are general purpose, like Firebase [69], Parse [159] and Meteor [141]; they resemble a wide-area storage system, with stateless clients accessing a shared back-end. Unlike a storage system, they let programmers define a custom data model for their applications. BaaS systems meet some mobile/cloud requirements and help programmers meet the rest. They are highly available, responsive and scalable. They offer weak or no consistency, and some provide notifications for more efficient reactivity. However, the application must still be stateless, so programmers must checkpoint to the back-end (or log to a local disk) after every operation for fault-tolerance. Finally,

programmers typically cannot run application code on the more powerful and secure cloud servers.

More specialized BaaS systems have been developed for application-specific cloud functionality. For example, PlayFish [164, 92], Playfab [163] and GameSparks [72] cater to mobile game developers, with an API specific to that class of applications. Figure 1.6 shows a typical architecture for a mobile game using a BaaS system. Mobile game Baas systems typically have APIs based on game concepts like scoreboards, coins, bags, and swords. This specialized API lets the BaaS implement some application logic in the cloud. For example, mobile gaming BaaS systems often implement simple game logic (e.g., exchanging coins for goods) in the cloud for security and performance. However, BaaS designers must carefully choose the cloud-side features and system API to meet the specific needs of an application class.

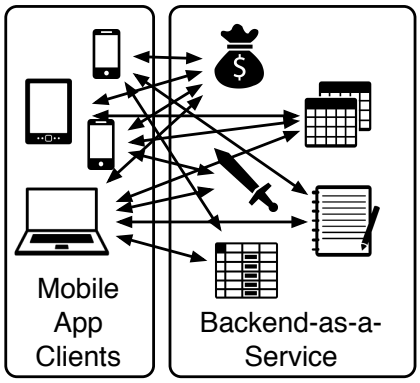


Figure 1.6: *Backend-as-a-Service architecture.* All cloud-side storage and computation is encapsulated by the back-end system, which presents either a general data store or a more app-specific API.

Using BaaS, programmers implement client-side code and rely on the back-end for almost all mobile/cloud requirements. Like wide-area storage systems, this benefit eliminates many application responsibilities. BaaS can often support applications with more run-time state because their higher level API naturally reduces accesses to the back-end and moves some computation to cloud servers. However, these systems work best when tailored to a narrow application class, like mobile games, which makes them less useful for other applications. Still, BaaS represents the first significant move towards a mobile/cloud operating system, albeit for a narrow set of applications.

1.5 Contributions

This thesis provides the basis for a new operating system for mobile/cloud applications. It consists of three new systems: (1) Sapphire, a distributed run-time manager, (2) Diamond, a distributed memory manager, and (3) TAPIR, a distributed storage system. Figure 1.7 shows the architecture of a new mobile/cloud OS and how it integrates into the mobile/cloud environment. Table 1.5 lists the OS components and their functions and new abstractions.

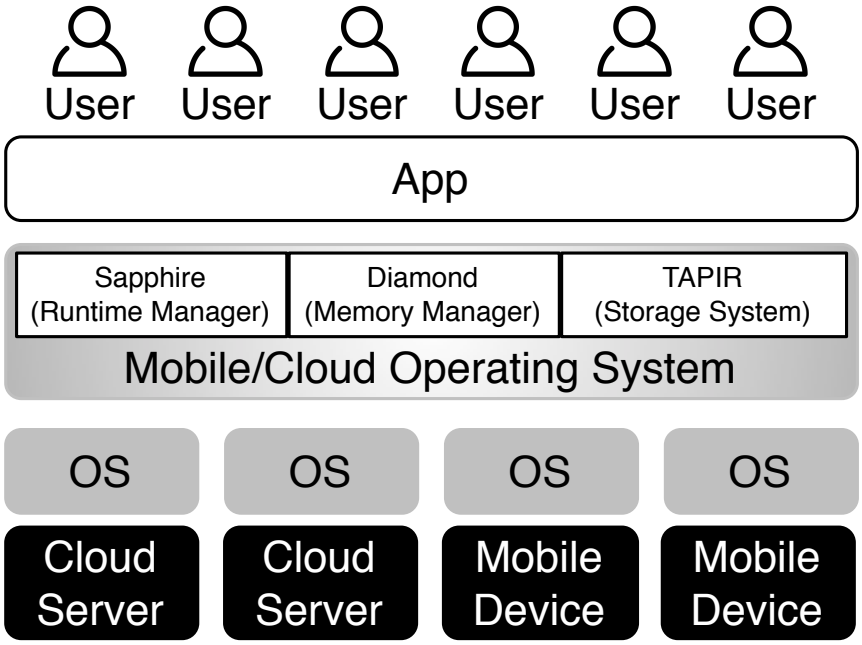


Figure 1.7: *The new mobile/cloud operating system.* The mobile/cloud OS spans mobile devices and cloud servers to provide end-to-end runtime, memory and storage management across the entire mobile/cloud application.

1.5.1 Run-time Management: Sapphire

Unlike desktop applications, mobile/cloud applications consist of many components partitioned into distributed processes that communicate and coordinate to implement application features.

Table 1.5: *Components of the new mobile/cloud operating system.* For each component, this table lists its functions and new abstractions.

OS Component	Component Functions	OS Abstractions
Run-time Manager (Sapphire)	• Partition and run application code	Sapphire Objects
	• Provide low-level deployment mechanisms	Deployment Kernel
	• Provide high-level run-time management	Deployment managers
Memory Manager (Diamond)	• Create copies of shared memory	Reactive Data Types
	• Link and synchronize shared memory	Reactive Data Map (`rmap`)
	• Propagate updates to derived data	Reactive Transactions
Storage Manager (TAPIR)	• Abstract low-level disk interface	Key-value interface
	• Mediate concurrent accesses	Distributed transactions
	• Handle crashes and recover from failures	Inconsistent Replication

Thus, Sapphire's first function is *deployment*: it automatically partitions and runs application code in distributed processes, provides transparent RPC between processes, and moves and tracks the location of processes. Sapphire introduces a new abstraction, a *Sapphire Object* (SO), which forms the unit of deployment and run-time management for application code. The *Sapphire kernel* provides best-effort, run-time management for Sapphire Objects, including object creation, migration, and tracking, best-effort RPC routing and delivery, and performance monitoring and failure detection.

However, to meet the six mobile/cloud requirements presented in Table 1.2, applications require more complex run-time management. For example, the application may need to spin-up copies of an SO (for scalability), replicate an SO (for availability), checkpoint an SO (for fault-tolerance), or automatically migrate an SO (for responsiveness). As noted previously, different applications may require different run-time management features depending on the application's needs and the

programmers' design decisions.

To accommodate the wide range of design choices, Sapphire introduces a new *deployment manager* abstraction to implement these more complex run-time management features. Deployment managers are run-time libraries that extend the Sapphire kernel with extra functionality. For example, they implement replication, RPC logging, and checkpointing to storage as different options to achieve fault-tolerance.

Application programmers choose one deployment manager per Sapphire object to customize the run-time management for each part of their application. As a result, programmers can construct a completely customized run-time manager to meet their application requirements. We designed and implemented a prototype of the Sapphire system. We discovered its flexibility greatly simplifies the run-time management needs for a wide range of mobile/cloud applications.

1.5.2 Memory Management: Diamond

Unlike desktop applications, mobile/cloud applications keep many copies of application state on mobile devices, cloud servers, caches and distributed storage. Further, they increasingly desire *reactivity*; that is, they want to automatically propagate updates across these copies, making updates from one user visible to other users without the need for save or refresh buttons.

Diamond's core function is to provide shared memory across the distributed processes of a mobile/cloud application while meeting the mobile/cloud requirements: any memory shared through Diamond is highly available, responsive, scalable, consistent, fault-tolerant and reactive. Diamond avoids the pitfalls of page-sized memory (e.g., false sharing) by introducing *reactive data types* (RDTs) as the unit for sharing. RDTs are simple primitives (e.g., string), simple collections (e.g., list), or conflict-free replicated data types [182] (e.g., set, counter). Diamond integrates a cloud-base distributed storage system with a key-to-RDT interface to help meet the mobile/cloud requirements.

Mobile/cloud applications place copies of application state in different places for different reasons. For example, an application might place copies on mobile devices for responsiveness, on cloud servers for scalability, and on storage servers for fault-tolerance. Diamond lets programmers place copies in any place for any reason with a new *reactive data map* (`rmap`) abstraction. Programmers use `rmap` to

link any in-memory variable to a key in the storage system. Diamond automatically synchronizes all memory linked to the same key; thus, programmers can create reliable, distributed shared memory across processes using `rmap`.

Finally, to automatically propagate updates from RDTs to memory derived from RDTs, Diamond introduces *reactive transactions*. Reactive transactions are general-purpose, strongly consistent transactions that automatically re-execute when a `rmap`ped RDT changes. Programmers can use reactive transactions to synchronize any application component from user interfaces to application-level caches (e.g., memcached [140]).

With these abstractions, Diamond can synchronize application memory without making assumptions about where it lives or how it is organized. As a result, mobile/cloud applications can create a shared memory space customized to their needs. We built a prototype of the Diamond system and found that it eliminated complex data management code and strengthened the consistency guarantees of a wide variety of mobile/cloud applications.

1.5.3 Storage Management: TAPIR

Existing mobile/cloud storage systems force programmers to make a trade-off between responsiveness and consistency. Thus, applications that can cope with consistency errors use weak consistency storage systems without transactions (e.g., Redis [173], MongoDB [146]); others that require correctness use strong transactional systems with limited performance (e.g., MegaStore [22]). Because no existing systems work for a wide range of applications, programmers must understand the system trade-offs and change storage systems if their application's needs change.

TAPIR alleviates this issue by providing a general-purpose storage system with strong guarantees and good performance. Our key observation is that existing transactional storage systems (e.g., Spanner [48], MegaStore [22]) waste performance by using both strongly consistent transaction protocols and replication protocols. For example, Spanner combines two-phase commit with strict two-phase locking and Paxos. TAPIR eliminates this wasted work by using a new transaction protocol that provides consistent transactions atop a new *inconsistent* replication protocol.

We evaluated TAPIR and found that it halves the commit latency and triples the throughput of

existing transactional storage systems. Even better, we found that TAPIR performs comparably to weak consistency systems, like MongoDB [146]. We verify that TAPIR provides strong linearizable distributed transactions with a TLA+ model-checked specification. TAPIR finally eliminates the trade-off between responsiveness and consistency, allowing it to meet the needs of a wide range of mobile/cloud applications.

1.5.4 Summary

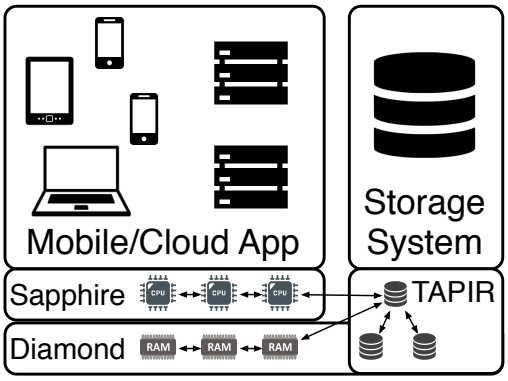


Figure 1.8: *Mobile/cloud application running on Sapphire, Diamond and TAPIR.* Note that all coordination has moved into the OS components, eliminating difficult distributed systems problems from the mobile/cloud application.

Together, Sapphire, Diamond and TAPIR provide a new mobile/cloud operating system that manages distributed computation, memory, and storage for mobile/cloud applications. Figure 1.8 shows the architecture of a mobile/cloud application running on the three systems integrated into an OS. Sapphire and Diamond work together to provide a single platform that flexibly and transparently meets any and all mobile/cloud requirements. Sapphire provides the abstractions and mechanisms for managing the application's run-time execution, while Diamond provides shared persistent memory for storing run-time state. TAPIR coordinates across the nodes of the distributed storage system to provide strong transactional guarantees to support the needs of Sapphire and Diamond. Together,

these three systems offer mobile/cloud applications a new set of abstractions to manage today's programming challenges.

2 Sapphire

As discussed in Section 1.1, run-time management is a key operating system function. Lacking an end-to-end OS, today's mobile/cloud programmers perform most run-time management themselves. As a result, mobile/cloud applications commonly implement distributed run-time tasks: communicating across multiple devices and servers, offloading execution from devices to the cloud, and integrating heterogeneous components with vastly different software stacks and hardware resources.

Compared to the desktop environment, the mobile/cloud environment is more resource-limited: mobile devices have limited computational abilities and the wide-area network has high latency and low throughput. These limitations require programmers to make *deployment decisions* when implementing run-time tasks, such as:

- Where application processes should be located
- What processes should be replicated or scaled
- What communication is needed to coordinate access to run-time state

These decisions depend on the requirements of each application component – such as scalability and fault tolerance – which force difficult performance vs. function trade-offs. The dependency between application requirements and deployment decisions leads programmers to mix deployment decisions with complex application logic in the code, which makes mobile/cloud applications difficult to implement, debug, maintain, and evolve. Even worse, the rapid evolution of devices, networks, systems, and applications means that the trade-offs that impact these deployment decisions are

constantly in flux. For all of these reasons, programmers need *a flexible* system that allows them to easily *create and modify distributed application deployments without needing to rewrite major parts of their application*.

This chapter presents Sapphire, a general-purpose distributed programming platform that greatly simplifies the design and implementation of applications spanning mobile devices and clouds. Sapphire removes much of the complexity of run-time management in a wide-area, multi-platform environment, yet still provides developers with the fine-grained control needed to meet critical application needs. A key concept of Sapphire's design is the *separation of application logic from deployment logic*. That is, deployment code is factored out of application code, allowing the programmer to focus on the application logic. At the same time, the programmer has full control over deployment decisions and the flexibility to customize them.

Sapphire's architecture facilitates this separation with a highly extensible distributed kernel and run-time system. At the bottom layer, Sapphire's *Deployment Kernel* (DK) integrates heterogeneous mobile devices and cloud servers through a set of common low-level mechanisms, including best-efforts RPC communication, failure detection, and location finding. Between the kernel and the application is a *deployment layer* – a collection of pluggable *Deployment Manager* (DM) modules that extend the kernel to support application-specific deployment needs, such as replication and caching. DMs are written in a generic, application-transparent way, using interposition to intercept important application events, such as RPC calls. The DK provides a simple yet powerful distributed execution environment and API for DMs that makes them extremely easy to write and extend. Conceptually, Sapphire's DK/DM architecture creates a seamless distributed run-time system that is customized specifically for each application's requirements.

We implemented a Sapphire prototype on Linux servers and Android mobile phones and tablets. The prototype includes a library of 26 Deployment Managers supporting a wide range of distributed management tasks, such as consistent client-side caching, durable transactions, Paxos replication, and dynamic code offloading between mobile devices and the cloud. Using the kernel's extension API, we were able to implement all of the deployment managers in only 10 to 177 lines of code. We also built 10 Sapphire applications, including a fully featured Twitter clone, a multi-player game, and

a shared text editor.

Our experience and evaluation show that Sapphire's extensible three-layer architecture greatly simplifies the construction of both mobile/cloud applications and distributed deployment functions. For example, a single-line application code change – switching from one DM to another – is sufficient to transform a cloud-based multi-player game into a P2P (device-to-device) version that significantly improves the game's performance. The division of function between the DK and DM layers makes deployments extremely easy to code; e.g., the DM to support Paxos state machine replication is only 129 lines of code, an order of magnitude smaller than a C++ implementation built atop an RPC library. We also demonstrate that Sapphire's structure provides fine-grained control over performance trade-offs, delivering performance commensurate with today's popular communication mechanisms like REST.

2.1 Background

Section 1.4 described a some general architectures for mobile/cloud applications. Currently, programmers must deploy their applications across this patchwork of user devices, cloud servers, and backend services, while satisfying demanding requirements such as responsiveness and availability. For example, the programmer might need to apply caching techniques, perform application-specific splitting of code across clients and servers, and develop solutions for fast and convenient data sharing, scalability, and fault tolerance.

Programmers use tools and systems when they match the needs of their application. In some cases an existing system might support an application entirely; for example, a simple application that only requires data synchronization could use a backend storage service like Dropbox [62], Parse [159] or S3 [176]. More complex applications, though, must integrate multiple tools and systems into a custom platform that meets their needs. These systems include server-side storage like Redis [173] or MySQL [153] for fault-tolerance, protocols such as REST [68] and SOAP [186] or libraries like Java RMI and Thrift [13] for distributed communication, load-balanced servers for scalability, client-side caching for lower wide-area latency, and systems for notification [4], coordination [35, 98], and monitoring [51].

Sapphire provides a flexible environment whose extension mechanism can subsume the functions of many of these systems, or can integrate them into the platform in a transparent way. Programmers can easily customize the run-time system to meet the needs of their applications. In addition, programmers can quickly switch deployment solutions to respond to environment or requirement changes, or simply to test and compare alternatives during development. Finally, Sapphire's Deployment Manager framework simplifies the development or extension of distributed deployment code.

2.2 Architecture

Sapphire is a distributed programming platform designed for flexibility and extensibility. In this section, we cover our goals in designing Sapphire, the deployment model that we assume, and Sapphire's system architecture.

2.2.1 Design Goals

We designed Sapphire with three primary goals:

1. *Create a distributed run-time platform spanning devices and the cloud.* A common platform integrates the heterogeneous distributed environment and simplifies communications, code/data mobility, and replication.

2. *Separate application logic from deployment logic.* Application code is focuses on servicing client requests rather than distribution, simplifying programming, evolution, and optimization.

3. *Facilitate system extension and customization.* Delegating run-time management to an extensible deployment layer gives programmers the flexibility to easily make or change deployment options.

Sapphire is designed to deploy applications across mobile devices and cloud servers. This environment causes significant complexity, as the programmer must stitch together a distributed collection of highly heterogeneous software and hardware components with a broad spectrum of capabilities, while still meeting application goals.

Sapphire is *not* designed for deploying backend services like Spanner [48] or ZooKeeper [98]; its applications interact with such backend services using direct calls, similar to current apps. A Sapphire Deployment Manager can easily integrate a backend service transparently to the application, e.g., using ZooKeeper for coordination or Spanner for fault-tolerance. Sapphire is also not designed for building user interfaces; we expect applications to customize their user interfaces for the devices they employ.

2.2.2 *System Architecture*

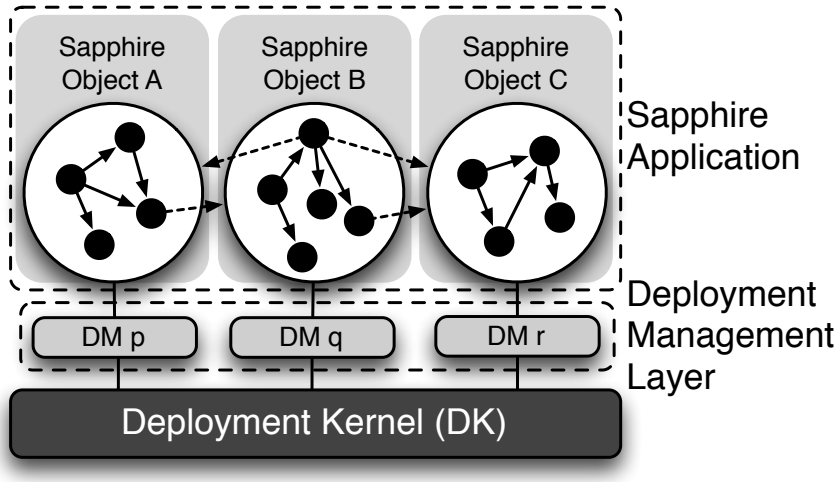


Figure 2.1: *Sapphire run-time architecture.* A Sapphire application consists of a distributed collection of Sapphire Objects executing on a distributed Deployment Kernel (DK). A DK instance runs on every device or cloud node. The Deployment Management (DM) layer handles distribution management/deployment tasks, such as replication, scalability, and performance.

Figure 2.1 shows an application-level view of Sapphire's architecture. A Sapphire application, which encompasses all of the client-side and server-side application logic, consists of a collection of *Sapphire Objects* (SOs). Each Sapphire Object functions as a single unit of distribution, like a virtual node. Sapphire Objects in an application share a logical address space that spans all cloud servers and client-side devices. That is, a Sapphire application is written so that all SOs can invoke each other

directly through simple location-independent procedure calls.

The bottom layer of Figure 2.1 is the *Deployment Kernel* (DK), which is a flexible and extensible distributed run-time system. It provides only the most basic distribution functions, including SO addressing and location tracking, best effort RPC-based communication, SO migration, and basic resource management. It does *not* support more complex tasks, such as fault tolerance, failure management, reliability, and consistency. In this way, the DK resembles IP-level network messaging – it is a basic service that relies on higher levels of software to meet more demanding program goals. The kernel is thus deployment agnostic and does not favor (or limit the application to) any specific approaches to deployment issues.

More complex management tasks are supported in the deployment layer by extensions to the DK, called *Deployment Managers* (DMs). Each Sapphire Object can optionally have an attached DM – shown in the middle of Figure 2.1 – which provides run-time distribution support in addition to the minimal features of the DK. The programmer selects a DM to manage each SO; e.g., he may choose a DM that handles failures to improve fault-tolerance, or one to cache data locally on a mobile device for performance. We have built a library of DMs supporting common distribution tasks used by applications today.

The separation between the DK and DMs provides significant flexibility and extensibility within the Sapphire distributed programming platform. As extensions to the DK, Deployment Managers provide additional distribution management features or guarantees for individual SOs. Often, these features involve performance trade-offs; thus, not every application or every SO will want or need a DM. Finally, by separating application logic (in the application program) from deployment logic (provided by DMs), we greatly reduce application complexity and allow programmers to easily change application deployment or performance behaviors.

2.3 Programming Model

The Sapphire application programming model is object based and could be integrated with any object-oriented language. Our implementation (Section 2.6) uses Java.

Sapphire Objects are the key programming abstraction for managing application code and data

locality at run time. To develop a Sapphire application, the programmer first builds the application logic as a single object-oriented program. He then breaks the application into distributed components by declaring a set of application objects to be Sapphire Objects. Sapphire Objects can still call each other via normal method invocation, however, these calls may now be remote invocations. Finally, the programmer applies Deployment Managers (DMs) to SOs as desired for additional distributed management features. In this section, we will show that the Sapphire programming model provides: (1) ease of programming in a distributed environment, (2) flexibility in deployment, and (3) programmer control over performance.

Defining Sapphire Objects. Programmers define Sapphire Objects as classes using a `sapphireclass` declaration, instead of the standard `class` declaration. As an example, Figure 2.2 shows a code snippet from our Twitter-clone, BlueBird. All instances of the User class defined here are independent SOs. In this case, the programmer has also specified a DM for the class, called ConsistentCaching, to enhance the object's performance.

SOs can encapsulate internal language-defined objects (Java objects in our system), such as the User string and arrays. These are shown as small solid circles in Figure 2.1; the solid arrows in the figure are references between internal objects within an SO. SO-internal objects cannot move independently or be accessed directly from outside the SO. The SO is therefore the granularity of distribution and decomposition in Sapphire. Moving an SO always moves all of its internal objects along with it; therefore, the programmer knows that all SO-internal objects will always be co-located with the SO.

A Sapphire Object encapsulates data and computation into a "virtual node" that: (1) ensures that each data/computation unit (a Sapphire Object) will always have its code and data on the same node, (2) lets the system transparently relocate or offload that unit, (3) supports easy replication of units, and (4) provides an easy-to-understand unit of failure and recovery. These benefits make Sapphire Objects a powerful abstraction; using fine-grained programmer-defined Sapphire Objects, instead of a coarse-grained client/server architecture, increases *both* flexibility in distributed deployment and programmer control over performance.

```
1  public sapphireclass User uses ConsistentCaching {
2      // user handle
3      String username;
4      // people who follow me
5      User[] followers;
6      // people who I follow
7      User[] friends;
8        ....
9      public String getUsername() {
10         return username;
11     }
12     public User[] getMyFollowers() {
13         return followers;
14     }
15     public User[] getPeopleIFollow() {
16         return friends;
17     }
18     public Tweet[] getMyTweets() {
19         return myTweets.getTweets();
20     }
21 }
```

Figure 2.2: *Example Sapphire object code from BlueBird.*

Calling Sapphire Objects. Sapphire Objects communicate using method invocation. The dashed lines in Figure 2.1 show cross-SO references, which are used to invoke the target SO's public methods. Invocation is location-independent and *symmetric*; it can occur transparently from mobile device to server, from server to device, from device to device, or between servers in the cloud. An SO can be moved by its DM or by the DK as a result of resource constraints on the executing node. Therefore, between two consecutive invocations from SO A to SO B, either or both objects can change location; the DK hides this change from the communicating parties. Invocations can fail, e.g., due to network or node failure; DMs help to handle failure on behalf of SOs.

SOs are passed by reference. All other arguments and return values from SO invocations are passed by value. For example, the return value of `getUsername()` in Figure 2.2 is a copy of the username object stored inside the SO, while `getMyFollowers()` returns a copy of the array containing references to User SOs. This preserves the encapsulation and isolation properties of Sapphire

Objects, since it is impossible to export the address of internal objects within them.

Our goal was to create a uniform programming model integrating mobile devices and the cloud *without* hiding performance costs and trade-offs from the programmer. Therefore, the programmer makes explicit choices in the decomposition of the application into SOs; once that choice is made, the system provides location-independent communication, which simplifies programming in the distributed environment.

Choosing Deployment Managers. Programmers employ the `uses` keyword to specify a DM when defining a Sapphire Object. For example, in Figure 2.2, the `sapphireclass` declaration (line 1) binds the ConsistentCaching DM to the User class. In this case, every instance of User created by the program will have the ConsistentCaching DM attached to it. It is easy to change the DM binding with a simple change to the sapphireclass definition.

Supporting DMs on a class basis lets programmers specify different features or properties for different application components. While the binding between an SO and its DM could be specified outside of the language (e.g., through a configuration file), we felt that this choice should be visible in the code because deployment decisions about the SO are closely tied to the requirements of an SO.

Sapphire provides a library of standard DMs, and most programmers will be able to choose the behavior they want from the standard library. Additionally, DMs are extensible; we discuss the API for building them in the next section. As programmers can build their own DMs and DMs are designed to be reusable, we expect the library to grow naturally over time.

An SO can have at most one DM, and each instance of the SO must use the same DM. We chose these restrictions for simplicity and predictability, both in the design of applications and DMs. In particular, the behavior of multiple DMs attached to an SO depends on the order in which the functions of the multiple DMs are invoked, and DMs could potentially interfere with each other. For this reason, programmers achieve the same result by explicitly composing DMs using inheritance. This allows the programmer to precisely control the actions of the composed DM. Since instances of the same SO should have the same deployment requirements, we chose not to allow different DMs for different instances of the same SO.

We chose not to allow multiple DMs per SO because it becomes difficult to predict what the behavior of combined DMs would be. Instead, we allow explicit composition of DMs through the DM extension API. Every instance of an SO is managed by an instance of the chosen DM.

DMs separate management code into generic, reusable modules that: (1) automatically deploy the application in complex ways, (2) give programmers per-application-component control over deployment trade-offs, and (3) allow programmers to easily change deployment decisions. These advantages make DMs a powerful mechanism for deploying distributed applications.

2.4 Deployment Kernel

Sapphire's *Deployment Kernel* is a distributed run-time system for Sapphire applications. At a high level, the goal of the DK is to create an integrated execution platform across mobile devices and servers. The key functions provided by the DK include: (1) management and location tracking of Sapphire Objects, (2) location-transparent inter-object communications (RPC), (3) low-level replica support, and (4) services to simplify the writing and execution of Deployment Managers.

A DK instance provides best-effort deployment of a single Sapphire application. It consists of a set of servers that run on every mobile and back-end computing device used by the application, and a centralized Object Tracking System (OTS) for tracking Sapphire Objects.

The Sapphire OTS is a distributed, fault-tolerant coordination service, similar to Chubby [35], ZooKeeper [98] and Tango [23]. The OTS is responsible for tracking Sapphire Objects across DK servers. DK servers only communicate occasionally with the OTS when creating or moving SOs. DK servers do not have to contact the OTS on every RPC because SO references contain a cached copy of the SO's last location,

Each DK server hosts a number of SOs by acting as an event server for the SOs, receiving and dispatching RPCs. The DK server also hosts and manages the DMs for those SOs. DK servers instantiate SOs locally by initializing the SO's memory, creating its DM (which potentially has components on multiple nodes), and registering the SO with the OTS. Once created, the server can move the SO at any time because SO location and movement are invisible to the application.

The DK provides primitive SO scheduling and placement. If a DK server becomes overloaded,

it will contact the OTS to find a new server to host the SO, move the SO to the new server, and update the OTS with the SO's new location. The DK API, described in Section 2.5, provides primitives that allow DMs to express more complex placement and scheduling policies, such as geo-replicated fault-tolerance, load balancing, etc.

To route an RPC to an SO, the calling DK server sends the RPC request to the destination server cached in the SO reference. If the destination no longer hosts the SO, the caller contacts the OTS to obtain the new address. If the destination DK server is unavailable, the calling server simply returns an error, because RPC in the DK is always best effort; DMs implement more advanced RPC handling, like retrying RPCs, routing RPCs between replicas, etc.

DK servers are not fault-tolerant: when they fail, they simply reboot. That is, on recovery, DK servers do not recover the SOs that they hosted on failure; they simply register with the OTS and begin hosting new SOs. Failures are entirely handled by DMs. We assume there is a failure detection system, such as FALCON [123], to notify the OTS when servers fail, which will then notify the DMs of the SOs that were hosted on the failed server.

We expect devices to be Internet connected most of the time, since applications today frequently depend on online access to cloud servers. When a device becomes disconnected, its DK server continues to run, however the application will be unable to make or receive remote RPCs. Therefore, any SOs hosted on a disconnected device will be inaccessible to outside devices and servers. The OTS keeps a list of mobile device IP addresses in order to quickly re-register SOs hosted on those devices when they reconnect. DMs can provide more advanced offline access.

2.5 Deployment Managers

A key feature of the Sapphire kernel is its support for the programming and execution of Deployment Managers, which customize and control the behavior of individual SOs in the distributed mobile/cloud environment. The DK provides direct API support for DMs. That API is available to DM developers, who we expect to be more technically sophisticated than application developers, although the DM framework can be used by anyone to customize or build new DMs. As this section will show, DMs can accomplish complex distributed deployment tasks with surprisingly little code. This is due to the

careful factoring of function between the DMs and the DK: the DK does the heavy lifting, while the DMs simply tell the DK what to lift through the DK's API.

2.5.1 DM Library

Table 2.1: *Library of Deployment Managers.*

Category	Extension	Description	LoC
Primitives	Immutable	Efficient distribution and access for immutable SOs	19
	AtLeastOnceRPC	Automatically retry RPCs for bounded amount of time	27
	KeepInPlace	Keep SO where it was created (e.g., to access device-specific APIs)	15
	KeepInCloud	Keep SO on cloud server (e.g., for availability)	15
	KeepOnDevice	Keep SO on accessing client device and dynamically move	45
Caching	ExplicitCaching	Caching w/ explicit push and pull calls from the application	41
	LeaseCaching	Caching w/ server granting leases, local reads and writes for lease-holder	133
	WriteThroughCaching	Caching w/ writes serialized on the server and stale, local reads	43
	ConsistentCaching	Caching w/ updates sent to every replica for strict consistency	98
Serializability	SerializableRPC	Serialize all RPCs to SO with server-side locking	10
	LockingTransactions	Multi-RPC transactions w/ locking, no concurrent transactions	81
	OptimisticTransactions	Transactions with optimistic concurrency control, abort on conflict	92
Checkpointing	ExplicitCheckpoint	App-controlled checkpointing to disk, revert last checkpoint on failure	51
	PeriodicCheckpoint	Checkpoint to disk every N RPCs, revert to last checkpoint on failure	65
	DurableSerializableRPC	Durable serializable RPCs, revert to last successful RPC on failure	29
	DurableTransactions	Durably committed transactions, revert to last commit on failure	112
Replication	ConsensusRSM-Cluster	Single cluster replicated SO w/ atomic RPCs across at least $f + 1$ replicas	129
	ConsensusRSM-Geo	Geo-replicated SO w/ atomic RPCs across at least $f + 1$ replicas	132
	ConsensusRSM-P2P	SO replicated across client devices w/ atomic RPCs over $f + 1$ replicas	138
Mobility	ExplicitMigration	Dynamic placement of SO with explicit move call from application	20
	DynamicMigration	Adaptive, dynamic placement to minimize latency based on accesses	57
	ExplicitCodeOffloading	Dynamic code offloading with offload call from application	49
	CodeOffloading	Adaptive, dynamic code offloading based on measured latencies	95
Scalability	LoadBalancedFrontEnd	Simple load balancing w/ static number of replicas and no consistency	53
	ScaleUpFrontEnd	Load-balancing w/ dynamic allocation of replicas and no consistency	88
	LoadBalancedMasterSlave	Dynamic allocation of load-balanced M-S replicas w/ eventual consistency	177

Sapphire provides programmers with a library of DMs that encompass many management features, including controls over placement and RPC semantics, fault-tolerance, load balancing and scaling, code-offloading, and peer-to-peer deployment. Table 2.1 lists the DMs that we have built along with a description and the LoC count[1] for each one. We built these DMs both to provide programmers with useful DMs for their applications and to illustrate the flexibility and programming ease of the DM programming framework.

2.5.2 DM Structure and API

We designed the DM API to provide as minimal an interface as possible while still supporting a wide range of extensions. A DM extends the functionality of the DK to meet the deployment requirements of a specific SO by interposing on DK events for the SO. For example, on an RPC to the SO, the DK will make an upcall into the DM for that SO. DMs are implemented as objects, therefore each DM can execute code on each upcall and store state between upcalls.

A DM consists of three component types: the *Proxy*, the *Instance Manager*, and the *Coordinator*. A programmer builds a DM by defining three object classes, one for each type. Since DMs are intended to manage distribution, the DK creates a distributed execution environment in which they operate; i.e., a DM is *itself* distributed and its components can operate on different nodes. When the DK instantiates a Sapphire Object with an attached DM, it also instantiates and distributes the DM's components. The DK provides transparent RPC between the DM components of an SO instance for coordination between components.

Figure 2.3 shows an example deployment of the DM components for a *single* Sapphire Object A. The DK may instantiate many Proxies and Instance Managers but at most one Coordinator, as shown in this figure. The center box (marked "Instance A") indicates that A has two replicas, marked replica 1 and replica 2. Each replica has its own copy of the Instance Manager. Were the DM to request a third replica of A, the DK would also create a new Instance Manager for that replica. A replica and its Instance Manager are always located on the same node.

---

[1]Generated using SLOCCount [216].

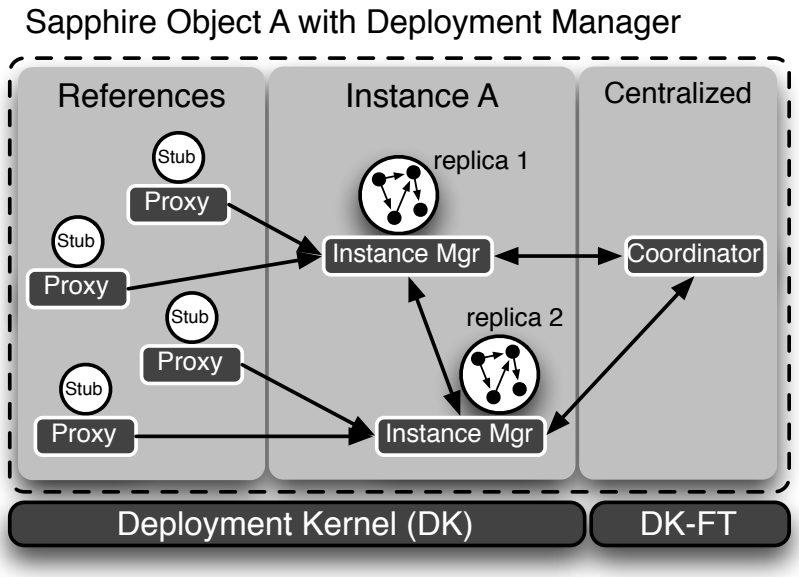


Figure 2.3: *Deployment Manager (DM) organization.* The components named *Proxy*, *Instance Mgr*, and *Coordinator* are all part of the DM for one Sapphire Object instance (shown here with two replicas). DK-FT is a set of fault-tolerant DK nodes, which also host the OTS, that support reliable centralized tasks for DMs and the DK.

Each component of the DM is responsible for a particular set of distributed tasks. Proxies are responsible for *caller-side tasks*, like routing method calls. Instance Managers are responsible for *callee-side tasks*, like keeping replicas of the SO synchronized. Note that, due to the symmetric nature of SOs, the caller of the method may be on a cloud server and the SO itself may be on a client device. Lastly, the Coordinator is responsible for *centralized tasks* such as failure handling. All three components are optional; a DM can define one or more of the components, and the DK will instantiate only those components that are defined.

The DK completely manages DM components; they run only when invoked, they reside only where the DK places them and are limited to communicating with other components in the same DM instance, which are attached to a single SO. The DK invokes DM components using *upcalls*, which are shown in Table 2.2. Each component receives a different set of upcalls according to the

Table 2.2: *Deployment Managers Upcall API.*

Event	Description
`onCreate`	Creation of SO instance
`onRPC`	Method invocation on SO
`onFailure`	Replica failed
`onDestroy`	Coordinator eliminated SO
`onHighLatency`	Avg. RPC latency > limit
`onLowMemory`	Node running out of memory
`onMemberChange`	New replica added to group
`onRefRequest`	Request for an SO reference

component's responsibilities. By *interposing* on Sapphire Object events such as method invocations, DMs can implement a variety of distributed management features transparently and generically.

In each upcall, the DM component can perform various management tasks on the SO using a set of primitives supported by the DK. Table 2.3 lists these primitives. The DM components of an SO instance can communicate directly with each other through a transparent RPC mechanism provided by the DK. Note that the DK supports only the most basic replication functions, namely, creating a new replica for an SO and reporting on replica locations. All decisions about the number of replicas, when to create or delete them, how to synchronize them, and how to handle failures occur at the DM level.

The left-most box in Figure 2.3 shows four other SOs. Each contains a reference to A, shown as an RPC stub in the figure, to which the DK has attached an instance of A's DM Proxy component. Making an RPC to A through the DK and its DM proceeds as follows. The DK reflects the call via an `onRPC()` upcall to the attached Proxy. The upcall to the Proxy lets A's DM intercept an RPC *on the caller's node* where, for example, it can implement client-local caching. If the Proxy wants to forward the call to replica 1 of A, it simply invokes replica 1's Instance Manager which runs in the same DK

server as replica 1. The Instance Manager will pass the RPC through to replica 1 of A.

Because the Proxies and Instance Managers for A are all part of the same Deployment Manager, they all understand whether or not the SO (A, in this case) is replicated, and, if so, *how* that replication is implemented. The choice of which replica to call is made *inside* the DM components, which are aware of each other and can communicate with each other directly through RPCs.

Finally, the DK instantiates one Coordinator for each DM instance, shown in the right-most box of Figure 2.3. The OTS manages Coordinators, keeping them fault-tolerant and centrally accessible. It is well known that a centralized coordinator can simplify many distributed algorithms (e.g., eliminating the need for leader election). Since the DK needs the OTS to tracking Sapphire Objects, it was easy to provide fault-tolerance for some DMs as well. We do not expect every DM to have a Coordinator, and even if there is a Coordinator, it is used sparingly for management tasks that are easiest handled centrally, such as instantiating new replicas in the event of failures. In this sense, Coordinators are similar to other centralized management systems, like Chubby [35] or ZooKeeper [98].

Programmers can easily extend or compose existing DMs using inheritance. The new DM inherits all of the behavior of the super-DM's Component object classes. The programmer can then override or combine upcalls in each component. While we considered automatic composition, we believe that the DM programmer should be involved to ensure that the composed DM implements exactly the behavior that the programmer expects. Our experience with composing DMs has shown that the use of inheritance for DM composition is straightforward and intuitive.

2.5.3 DM Code Example

Figure 2.4 shows a simplified definition of the LeasedCaching DM that we provide in the Sapphire Library. We include code for the Proxy component and the function declarations from the Instance Manager. This DM does not have a Coordinator because it does not need centralized management.

The LeasedCaching DM is not replicated, so DK will only create one Instance Manager. The Instance Manager hands out mutually exclusive leases to Proxies (which reside with the remote reference to the SO) and uses timeouts to deal with failed Proxies. The Proxy with a valid lease can read or write to a local copy of the SO. Read-only operations do not incur communication costs,

Table 2.3: *DK API for Deployment Managers.*

Operation	**Description**
`invoke(RPC)`	Invoke RPC on the local SO
`invoke(SO,RPC)`	Invoke RPC on a specific SO
`getNode()`	Get ID for local node
`getNodes()`	Get list of all nodes
`pin(node)`	Move SO to a node.
`setHighLatency(ms)`	Set limit for RPC latency
`durable_put(SO)`	Save copy of the SO
`durable_get(key)`	Retrieve SO
`replicate()`	Create a replica
`destroyReplica(IM)`	Eliminate a replica
`getReplicas()`	Get list of replicas for SO
`getReplica()`	Get ref to SO instance
`setReplica(SO)`	Set ref to SO instance
`copy(SO)`	Create a copy of the SO instance
`diff(SO,SO)`	Diff two SO instances
`sync(SO)`	Synchronize two SO instances
`getIM()`	Get ref to DM Instance Mgr
`setIM(IM)`	Set reference to DM Instance Mgr
`getCoordinator()`	Get ref to DM Coordinator
`getReference(IM)`	Create DM Proxy for IM
`registerMethod(m)`	Register a custom method for DM
`getRegion()`	Get ID for local region
`getNode()`	Get ID for local node
`pin(region)`	Move SO to region
`pin(node)`	Move SO to node
`getRegions()`	Get list of server regions
`getNodes()`	Get list of nodes in local region

```
1  public class LeasedCaching extends DManager {
2    public class LCProxy extends Proxy {
3      Lease lease;
4      SapphireObject so;
5
6      public Object onRPC(SapphireRPC rpc) {
7        if (!lease.isValid() || lease.isExpired()) {
8          lease = Sapphire.getReplica().getLease();
9          if (!lease.isValid()) {
10           throw new SONotAvailableException(
11             ``Could not get lease.'');
12         } else {
13           so = lease.getSO();
14         }
15       }
16
17       SapphireObject oldSO = Sapphire.copy(so);
18       Sapphire.invoke(so, rpc);
19       SOStream diff = Sapphire.diff(oldSO, so);
20       if (diff) Sapphire.getReplica().update(diff);
21     }
22   }
23
24   public class LCReplica extends InstanceManager {
25     public synchronized Lease getLease();
26     public synchronized void update(SOStream);
27     // Code for Instance Manager methods
28   }
29 }
```

Figure 2.4: *Example Deployment Manager with arguments.*

which saves latency over a slow network, but updates are synchronously propogated to the Instance Manager in case of Proxy failure.

When the application invokes a method on an SO with this DM attached, the caller's Proxy: (1) verifies that it holds a lease, (2) performs the method call on its local copy, (3) checks whether the object has been modified (using `diff()`), and (4) synchronizes the remote object with its cached copy if the object changed, using an `update()` call to the Instance Manager.

Each Proxy stores the lease in the Lease object (line 3) and a local copy of the Sapphire Object (line 4). If the Proxy does not hold a valid lease, it must get one from the Instance Manager (line

8) before invoking its local SO copy. If the Proxy is not able to get the lease, the DM throws a SONotAvailableException (line 10). The application is prepared for any RPC to an SO to fail, so it will catch the exception and deal with it. The application also knows that the SO uses the LeasedCaching SOM, so it understands the error string (line 11).

If the Proxy is able to get a lease from the Instance Manager, the lease will contain an up-to-date copy of the SO (line 13). The Proxy will make a clean copy of the SO (line 17), invoke the method on its local copy (line 18) and then diff the local copy with the clean copy to check for updates (line 19). If the SO changed, the Proxy will update the Instance Manager's copy of the SO (line 20). The copy and diff is necessary because the Proxy does not know which SO methods might write to the SO, thus requiring an update to the Instance Manager. If the DM had more insight into the SO (i.e., the SO lets the DM know which methods are read-only), we could skip this step.

The example illustrates a few interesting properties of DMs. First, DM code is application agnostic and can perform only a limited set of operations on the SO that it manages. In particular, it can interpose only on method calls to its SO, and it manipulates the managed SO as a black box. For example, there are DMs that automatically cache an SO, but no DMs that cache a part of an SO. This ensures a clean separation of object management code from application logic and allows the DM to be reused across different applications and objects.

Second, a DM cannot span more than one Sapphire Object: it performs operations only on the object that it manages. We chose not to support cross-SO management because it would require the DM to better understand the application; as well, it might cause conflicts between the DMs of different SOs. As a result, there are DMs that provide multi-RPC transactions on a single SO, but we do not support cross-SO transactions. However, the programmer could combine multiple Sapphire Objects into one SO or implement concurrency support at the application level to achieve the same effect.

2.5.4 DM Design Examples

This section discusses the design and implementation of several classes of DMs from the Sapphire Library, listed in Table 2.1. Our goal is to show how the DM API can be used to extend the DK for a

wide range of distributed management features.

Code-offloading. The code-offloading DMs are useful for compute-intensive applications. The CodeOffloading DM supports transparent object migration based on the performance trade-off between locating an object on a device or in the cloud, while the ExplicitCodeOffloading DM allows the application to decide when to move computation. The ExplicitCodeOffloading DM gives the application more control than the automated CodeOffloading DM, but is less transparent because the SO must interact with the DM.

Once the DK creates the Sapphire Object on a mobile device, the automated CodeOffloading DM replicates the object in the cloud. The device-side DM Instance Manager then runs several RPCs locally and asks the cloud-side Instance Manager to do the same, calculating the cost of running on each side. An adaptive algorithm, based on Q-learning [214], gradually chooses the lowest-cost option for each RPC. Periodically, the DM retests the alternatives to dynamically adapt to changing behavior since the cost of offloading computation depends on the type of computation and the network connection, which can change over time.

Peer-to-peer. We built peer-to-peer DMs to support the direct sharing of SOs across client mobile devices *without* needing to go through the cloud. These DMs dynamically place replicas on nodes that contain references to the SO. We implemented the DM using a centralized Coordinator that attempts to place replicas as close to the callers as possible, without exceeding an application-specified maximum number of replicas. We show the performance impact of this P2P scheme in Section 2.7.

Replication. The Sapphire Library contains three replication DMs that replicate a Sapphire Object across several servers for fault tolerance. They offer guarantees of serializability and exactly-once semantics, along with fault-tolerance. They require that the SO is deterministic and only makes idempotent calls to other SOs.

The Library's replication DMs model the SO as a replicated state machine (RSM) that executes operations on a *master replica*. These DMs all inherit from a common DM that implements the

RSM, then extend the common DM to implement different policies for replica placement (e.g., Geo-replicated, P2P).

The RSM DM uses a Coordinator to instantiate the desired number of replicas, designate a leader, and maintain information regarding the membership of the replica group. The Coordinator associates an epoch number with this information, which it updates whenever membership changes.

For each RPC, Instance Managers forward the request to the Instance Manager of the master replica, which logs the RPC and assigns it an ID. The master then sends the ID and epoch number to the other Instance Managers, which accept it if they do not have another RPC with the same ID. If the master receives a response from at least f other Instance Managers, it executes the RPC and synchronizes the state of the SO on the other replicas. If one of the replicas fails, the DK notifies the Coordinator, which allocates a new replica, designates a leader, starts a new epoch, and informs other replicas of the change.

Scalability. To scale Sapphire Objects that handle a large number of requests, the Sapphire Library includes both stateless and stateful scalability DMs. The LoadBalancedFrontEnd DM provides simple load balancing among a set number of replicas. This DM only supports Sapphire Objects that are stateless (i.e., do not require consistency between replicas); however, the SO is free to access state in other Sapphire Objects or on disk. The ScaleUpFrontEnd DM extends the LoadBalancedFrontEnd DM with automatic scale-up, The DM monitors the latency of requests and creates new replicas when the load on the SO and the latency increases. Finally, the LoadBalancedMasterSlave provides scalability for read-heavy workloads by dynamically allocating a number of read-only replicas that receive updates from the master replica. This DM uses the Coordinator to organize replicas and select the master. We show the utility of our scalability DMs in Section 2.7.

Discussion. The DM's upcall API and its associated DK API are relatively small (only 8 upcalls and 27 DK calls), yet powerful enough to cover a wide range of sophisticated deployment tasks. Most of our DMs are under a hundred lines of code. There are three reasons for this efficiency of expression. First is the division of labor between the DMs and the DK. The DK supports fundamental

mechanisms such as RPC, object creation and mobility, and replica management. Therefore, the DK performs the majority of the work in deployment operations, while the DMs simply tell the DK what work to perform.

Second is the availability of a centralized, fault-tolerant Coordinator in the DM environment. This reduces the complexity of many distributed protocols; e.g., in the ConsensusRSM DMs, the Coordinator simplifies consensus by determining the leader and group membership. Our three replication DMs share this code but make different replica placement decisions, meeting different goals and properties with the same mechanism. Inheritance facilitates the composition of new DMs from existing ones; e.g., the DurableTransactions DM builds upon the OptimisticTransactions DM, adding fault-tolerance with only 20 more lines of code.

Finally, the decomposition of applications into Sapphire Objects greatly simplifies DM implementation. We implemented the code-offloading DM in only 95 LoC because we do not have to determine the unit of code to offload dynamically, and because the application provides a hint that the SO is compute-intensive by choosing the DM. In contrast, current code-offloading systems [52, 85, 42] are much more complex because they lack information on application behavior and because the applications are not easily composed into locality units, such as objects.

2.6 Implementation

Our DK prototype was built using Java to accommodate Android mobile devices. Altogether, the DK consists of 12,735 lines of Java code, including 10,912 lines of Apache Harmony RMI code, which we had to port to Dalvik. Dalvik was developed based on Apache Harmony, but does not include an implementation for Java RMI.

Figure 2.5 shows the prototype's architecture. We used Sun's Java 1.6.0_38 JVM to run Sapphire in the cluster, while the tablets and phones ran Sapphire on the Android 4.2 Dalvik VM. We used Java RMI for low-level RPCs between DK nodes. We used Voldemort [212] as the storage back-end for our checkpointing DMs.

Java RMI provides only point-to-point communication and only supports calls to Java objects that have a special Java RMI-provided interface. Thus, we could only use Java RMI for low-level

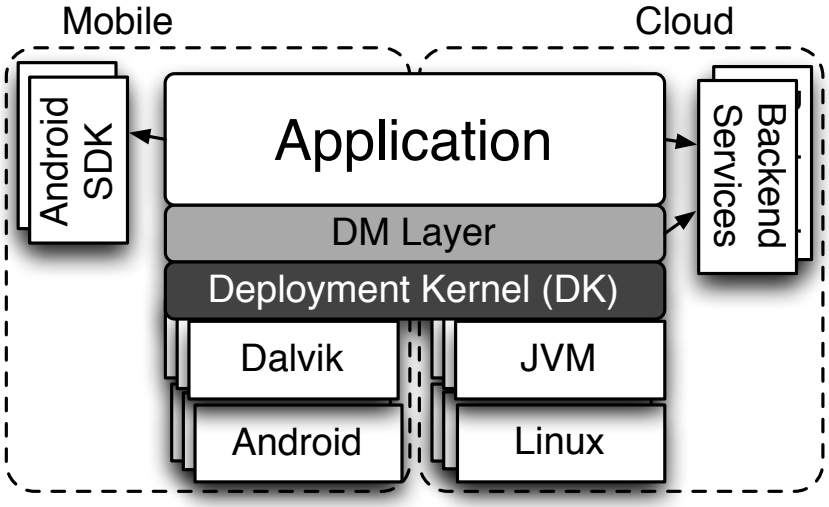


Figure 2.5: *Sapphire application and run-time system implementation.*

communication between DK servers and the OTS. To achieve transparent communication between SOs and between DM components, we built a compiler (862 LoC) that creates stubs for SOs and for DM Instance Managers and Coordinators. Having a stub for each SO allows the DK server to route RPCs and invoke DM components on the callee and caller side. DM Instance Managers and Coordinators also require stubs because the DK needs to be able to support transparent RPC from Instance Managers and Proxies. The compiler generates stubs as Java classes that extend the class of the target object, replacing all method contents with forwarding functions into the DK. A stub is therefore a reference that can be used for transparent communication with the remote object through the DK.

We also rely on Apache Harmony's implementation of RMI serialization – with Java reflection to marshall and unmarshall objects – for sending, diffing and copying objects. We did no optimization of Java RMI at all in this prototype. We could have applied well-known techniques [133, 162, 154] to improve RPC performance and expect to do so in the future; however, as we show in our evaluation, our performance is competitive with widely used client-server mechanisms, such as REST. In order to achieve this performance on mobile devices, we had to fix several bugs that caused performance problems in the Apache Harmony RMI code that we ported to Android.

Our prototype does not currently include secure communication between DK servers. Java RMI supports SSL/TLS, so our prototype could easily support encrypted communication between DK servers. We would also require an authentication mechanism for registering DK servers on mobile devices, like Google SSO [83].

In today's applications, mechanisms such as access control checks are typically provided by the application. With a unified programming platform like Sapphire, it becomes possible to move security mechanisms into the platform itself. While this discussion is outside the scope of the paper, we are currently exploring the use of information flow control-based protection for mobile/cloud applications in the context of Sapphire's object and DK/DM structure.

2.7 Experience & Evaluation

This section presents qualitative and quantitative evaluations of Sapphire. We first describe our experience building new applications and porting applications to Sapphire. Second, we provide low-level DK performance measurements, and an evaluation of several DMs and their performance characteristics. Our experience demonstrates that: (1) Sapphire applications are easy to build, (2) the separation of application code and deployment code, along with the use of symmetric (i.e., non-client-server) communication, maximizes flexibility and choice of deployment for programmers, and (3) Deployment Managers can be used effectively to improve performance and scalability in a dynamic distributed environment.

2.7.1 Applications

We consider the design and implementation of several Sapphire applications with respect to three objectives:

- **Development Ease:** It should be easy to develop mobile/cloud applications either from scratch or by porting non-distributed mobile device applications to Sapphire. Furthermore, it should be possible to write application code without explicitly addressing distribution management.

- **Deployment Flexibility:** The programmer should be able to choose from alternative distribution management schemes and change deployment decisions without rewriting application code.

- **Management Code Generality:** It should be possible to develop generic distribution management/deployment components that can be used widely both within an application and across different applications.

Table 2.4: *Sapphire applications.* We divide each application into front-end code (the UI) and back-end code (application logic). The source column indicates whether we developed new native Sapphire code or ported open-source code to Sapphire.

	Back-end		**Front-end**	
Application	**Source**	**LoC**	**Source**	**LoC**
To Do List	Native	48	Native	132
Text/Table Editor	Native	409	Native	533
Multi-player Game	Native	588	Native	1,186
BlueBird	Native	783	Ported	13,009
Sudoku Solver	Ported	76	-	-
Regression	Ported	348	-	-
Image Recognition	Ported	102	-	-
Physics Engine	Ported	108	-	-
Calculus	Ported	818	-	-
Chess AI	Ported	427	-	-

Table 2.4 lists several applications that we built or ported, along with their LoC. We built three applications from scratch: an online to-do list, a collaborative text and table editor, and a multi-player

game. We also built a fully-featured Twitter clone, called BlueBird, and paired it with the front-end UI from Twimight [205], an open-source Android Twitter client. The table also lists six non-distributed, compute-intensive applications that we ported to Sapphire.

Development Ease. It took relatively little time and programming experience to develop Sapphire applications. In particular, the existence of a DM library lets programmers write application logic without needing to manage distribution explicitly. Two applications – the multi-player game and collaborative editor – were written by undergraduates who had never built mobile device or web applications and had little distributed systems experience. In under a week, each student wrote a working mobile/cloud application of between 1000 and 1500 lines of code consisting of five or six Sapphire Objects spanning the UI and Sapphire back-end.

Porting existing applications to Sapphire was easy as well. For the compute-intensive applications, a single line change was sufficient to turn a Java object into a distributed SO that could adaptively execute either on the cloud or the mobile device. We did not have to handle failures because the CodeOffload DM hides them by transparently re-executing the computation locally when the remote site is not available. An undergraduate ported all six applications – and implemented the CodeOffload DM as well – in less than a week.

Our largest application was BlueBird, a Twitter clone that was organized as ten Sapphire Objects: Tweet, Tag, TagManager, Timeline, UserTimeline, HomeTimeline, MentionsTimeline, FavoritesTimeline, User and UserManager. We implemented all Twitter functions except for messaging and search in under 800 lines. In comparison, BigBird [60], an open-source Twitter clone, is 2563 lines of code, and Retwis-J [122], which relies heavily on Redis search functionality, is 932 lines of code.

Distributed mobile/cloud applications must cope with the challenges of running on resource-constrained mobile devices, unreliable cloud servers, and high-latency, wide-area links. Using Sapphire, these challenges are handled by selecting DMs from the DM library, which greatly simplifies the programmer's task and makes it easy to develop and test alternative deployments.

Deployment Flexibility. Changing an SO's DM, which changes its distribution properties, requires only a one-line code change. We made use of this property throughout the development of our applications as we experimented with our initial distribution decisions and tried to optimize them.

In BlueBird, for example, we initially chose not to make Tweet and Tag into SOs; since these objects are small and immutable, we thought they did not need to be independent, globally shared objects. Later, we realized that it would be useful to refer directly to Tweets and Tags from Timeline objects rather than accessing them through another SO. We therefore changed them to SOs – a trivial change – and then employed ExplicitCaching for both of them to reduce the network delay for reads of the tweet or tag strings.

As another example, we encountered a deployment decision in the development of our multi-player game. The Game object lasts only for the duration of a game and can be accessed only from two devices used to play. Since the object does not need high reliability or availability, it can be deployed in any number of ways: on a server, on one of the devices, or on both devices. We first deployed the Game object on a cloud server and then decided to experiment with peer-to-peer alternatives. Changing from the cloud deployment to peer-to-peer using the KeepOnDevice and ConsensusRSM-P2P managers in our DM library required only a *single line change*, and improved performance (see Section 2.7.4) and allowed games to continue when the server is unavailable. In contrast, changing an application for one of today's systems from a cloud deployment to a peer-to-peer mobile device deployment would require significant application rewriting (and might even be impossible without an intermediary cloud component due to the client-server nature of existing systems).

Management Code Generality. We applied several DMs to multiple SOs within individual applications and across applications. For example, many of our applications have an object that is shared among a small number of users or devices (e.g., ToDoList, Document, etc.). To make reads faster while ensuring that users see immediate updates, we used the ConsistentCaching DM for all of these applications. Without the DM structure, the programmers would have to write the caching and synchronization code explicitly for each case.

Table 2.5: *Sapphire latency comparison.* Request latencies (ms) for local, server-to-server, tablet-to-server, server-to-tablet and tablet-to-tablet. Note that REST does not support communication to tablets.

RPC Protocol	Local	S→S	T→S	S→T	T→T
Sapphire	0.08	0.16	5.9	3.4	12.0
Java RMI	0.05	0.12	4.6	2.0	7.2
Thrift	0.04	0.11	2.0	2.0	3.6
REST	0.49	0.64	7.9	-	-

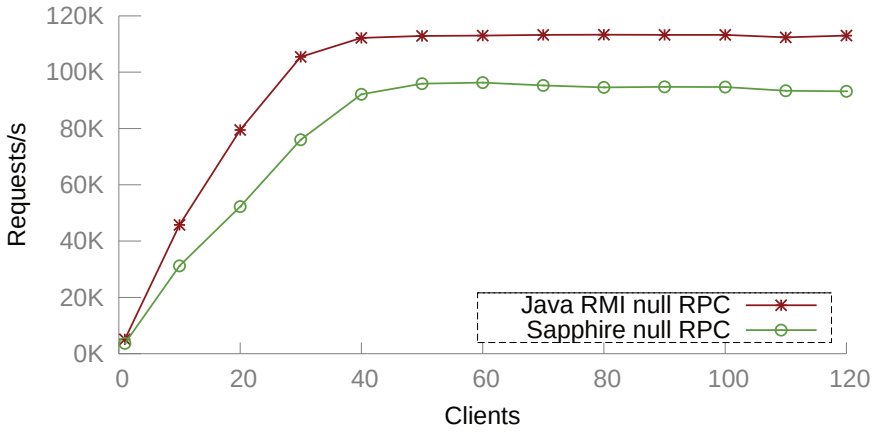


Figure 2.6: *Sapphire throughput measurement.* Throughput of a Sapphire Object versus an RMI Object.

Even within BlueBird, which has 10 Sapphire Object types, we could reuse several DMs. If the deployment code for each BlueBird SO had to be implemented in the application, the application would grow by at least 800 LoC, more than doubling in size! This number is conservative: it assumes the availability of the DM API and the DK for support. Without those mechanisms, even more code would be required.

2.7.2 *Experimental Setup*

Our experiments were performed on a homogeneous cluster of server machines and several types of devices (tablets and phones). Each server contained 2 quad-core Intel Xeon E5335 2.00GHz CPUs with 8GB of DRAM running Ubuntu 12.04 with Linux kernel version 3.2.0-26. The devices were Nexus 7 tablets, which run on a 1.3 GHz quad-core Cortex A9 with 1 GB of DRAM, and Nexus S phones with a 1 GHz single-core Hummingbird processor and 512MB of DRAM. The servers were all connected to one top-of-rack switch. The devices were located on the same local area network as the servers, and communicated with the server either through a wireless connection or T-mobile 3G links.

2.7.3 *Microbenchmarks*

We measured the DK for latency and throughput using closed-loop RPCs. Latencies were measured at the client. Before taking measurements, we first sent several thousand requests to warm up the JVM to avoid the effects of JIT and buffering optimizations.

RPC Latency Comparison. We compared the performance of Sapphire RPC to Java RMI and to two widely used communication models: Thrift and REST. Apache Thrift [13] is an open-source RPC library used by Facebook, Cloudera and Evernote. REST [68] is a popular low-level communication protocol for the Web; many sites have a public REST API, including Facebook and Twitter. We measured REST using a Java client running the standard `HttpURLConnection` class and a PHP script running on Apache 2.2 for method dispatch.

Table 2.5 shows request/response latencies for intra-node (local), server-to-server, tablet-to-server, server-to-tablet and tablet-to-tablet communications on null requests for all four systems. While Thrift was slightly faster in all cases, Java RMI and Sapphire were comparable and were both faster than the Java REST library.

Sapphire uses Java RMI for communication between DK servers; however, we dispatch method calls to SOs through the DK. This additional dispatch caused the latency difference between Java

RMI and Sapphire RPC. The extra cost was primarily due to instantiating and serializing Sapphire's RPC data object (which is not required for a null Java RMI RPC). We could reduce this cost by using a more efficient RPC and serialization infrastructure, such as Thrift.

Note that even without optimization, Sapphire was faster than REST, which is probably the most widely used communication framework today. Furthermore, we could not show REST performance for server-to-tablet and tablet-to-tablet because REST's client-server architecture cannot accept HTTP requests on the tablet. Thus, REST can be used only for tablet-to-server communication, requiring the application to explicitly manage communication forms such as server-to-client or client-to-client.

Throughput Comparison. We measured request throughput for the Sapphire DK and Java RMI. The results (Figure 2.6) showed similar throughput curves, with Java RMI object throughput approximately 15% higher than that for Sapphire Objects. This is because Sapphire null RPCs are not truly empty: they carry a serialized structure telling the DK how to direct the call. To break the cost down further, we measured the throughput of a Java RMI carrying a payload identical to that of the Sapphire null RPC. This reduced the throughput difference to 3.6%; this 3.6% is the additional cost of Sapphire's RPC dispatching in the DK, with the remainder due to the cost of serialization for the dispatching structure. Again, there are many ways to reduce the cost of this communication in Sapphire, but we leave those optimization to future work.

Sapphire DK Operation Cost. We measured the latency of several DK services. DK call latency depends on the size and complexity of the object, since we use Java serialization. Table 2.6 shows latency results for creating, replicating, and moving SOs on servers and tablets. Operation latencies were low when executed on cloud servers. Tablets were considerably slower than cloud servers. However, we expect most management operations such as these to be performed in the cloud (i.e., we do not expect tablets to create large numbers of SOs).

The SO instantiation process can be expensive because the DK must create several objects locally: the SO, the SO stub, the DM Proxy and the DM Instance Manager. The DK must also create the DM Coordinator remotely on a DK-FT node and register the SO with the OTS. Communication with the

DK-FT node and the OTS accounted for nearly half the instantiation latency.

Table 2.6: *Sapphire Deployment Kernel API latency measurement (ms).*

	create		**replicate**		**move (over WiFi)**			
Object	**S**	**T**	**S**	**T**	**S→S**	**T→S**	**S→T**	**T→T**
Table	1.1	28	0.5	15	1.9	42	16	66
Game	1.1	29	0.5	16	2.1	49	19	67
TableMgr	1.1	27	0.6	18	2.2	50	16	78

2.7.4 Deployment Manager Performance

We measured the performance of five categories of DMs: caching, replication, peer-to-peer, mobility, and scalability. Our goal was to examine their effectiveness as extensions to the DK and the costs and trade-offs of employing different DMs.

Caching. We evaluated two caching DMs: LeaseCaching and ConsistentCaching. As expected, caching significantly improved the latency of reads in both cases. For the TodoList SO, which uses the LeaseCaching DM, caching reduced read latency from 6 ms to 0.5 ms, while write latency increased from 6.1 ms to 7.5 ms. For the Game SO, which uses the ConsistentCaching DM, all read latencies decreased, from 7-13 ms to 2-3 ms. With consistent caching, the write cost to keep the caches and cloud synchronized was significant, increasing from 29 ms to 77 ms. Overhead introduced by the DM was due to the use of serialization to determine read vs. write operations. For writes, the whole object was sent to be synchronized with the cloud, instead of a compact patch.

Code offloading. We measured our ported, compute-intensive applications with the CodeOffloading DM for two platforms: the Nexus 7 tablet and the Galaxy S smartphone. Figure 2.7 shows the latencies for running each application locally on the device (shown as Base), offloaded to the cloud over WiFi, and offloaded over 3G. The offloading trade-offs varied widely across the two platforms due to differences in CPU speed, wireless, and cellular network card performance. For example, for the

Calculus application, cloud offloading was better for the phone over both wireless and 3G; however, for the tablet it was better only over wireless. For the Physics engine, offloading was universally better, but it was particularly significant for the mobile device, which was not able to provide real-time simulation without code offloading.

These cross-platform differences in performance show the importance of flexibility. An automated algorithm cannot always predict when to offload and can be costly. Therefore, it is important for the programmer to be able to easily change application deployment to adapt to new technologies.

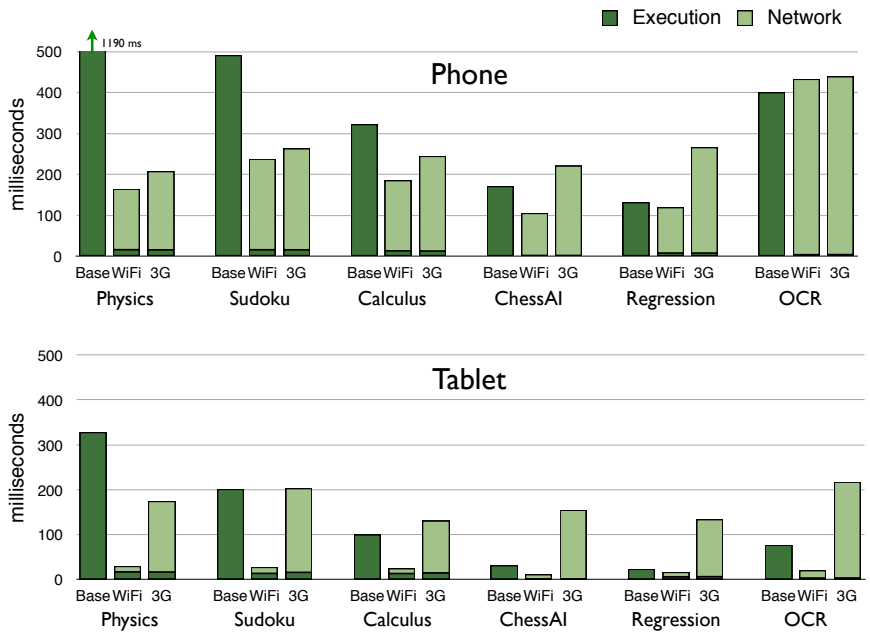


Figure 2.7: *Code offloading evaluation.*

Scalability. We built the LoadBalancedFrontEnd DM to scale a stateless SO under heavy load. The DM creates a given number of non-consistent replicas of an SO and assigns clients to the replicas in a round-robin fashion. Figure 2.8 shows the throughput of the SO serving null RPCs when the DM creates up to 3 replicas. Throughput scaled linearly with the number of replicas until 257,365 requests/second, at which point the 1Gb network was saturated.

Peer-to-Peer Deployments. Sapphire lets programmers move objects easily between clients and

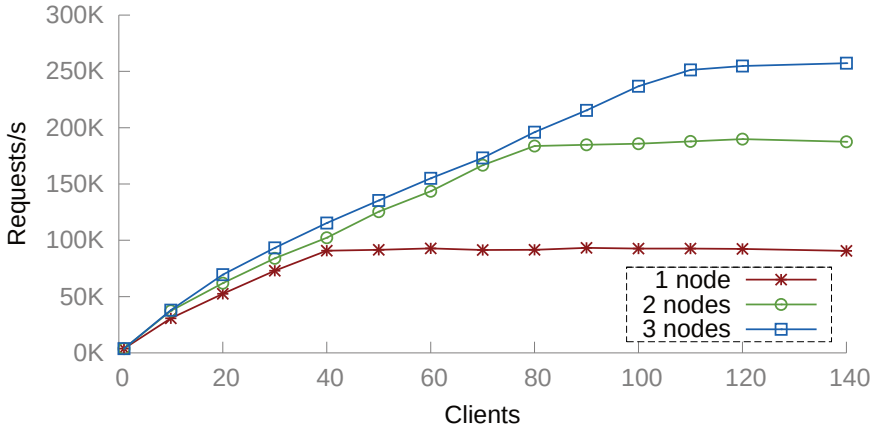


Figure 2.8: *DM evaluation.* Throughput using the LoadBalancedFrontEnd DM.

servers, enabling P2P deployments that would be difficult or impossible in existing systems. We measured three deployments for the Game SO from our multi-player game: (1) without a DM, which caused Sapphire to deploy the SO on the server where it is created; (2) with the KeepOnDevice DM, which dynamically moved the Game object to a device that accessed it; and (3) with the ConsensusRSM-P2P DM, which created synchronized replicas of the Game SO and placed them on the callers' devices.

For each deployment, Figure 2.9 shows the latency of the game's read methods (`getScrambleLetters()`, `getPlayerTurn()` and `getLastRoundStats()`) and write methods (`play()` and `pass()`). With the Game SO in the cloud, read and write latencies were high for both players. With the KeepOnDevice DM, the read and write latencies were extremely low for the device hosting the SO, but somewhat higher for the other player, compared to the cloud version. Finally, with the ConsensusRSM-P2P DM, read latencies were much lower for both devices, while write latencies were higher. In our scenario, the two tablets and the server were on the same network. In cases where the two players are close on the network and far from the server, the peer-to-peer DMs would provide a valuable deployment option.

With the DMs, no cloud servers were needed to support the Game SO; this reduced server load, but Game SOs were no longer available if the hosting device were disconnected. This experiment

shows the impact of different deployment options and the benefit of being able to flexibly choose alternative deployments to trade off application performance, availability, and server load.

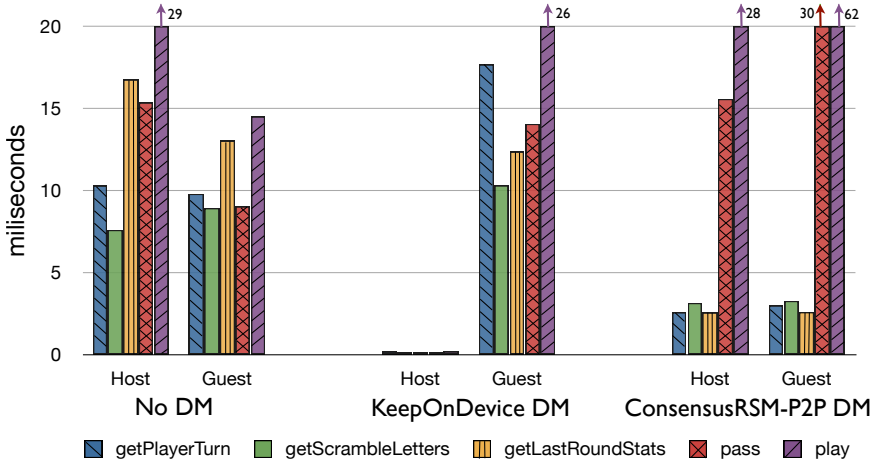


Figure 2.9: *Multi-player Game with different DMs.* Performance is measured in latency to make each application-level call.

2.8 Related Work

Researchers have built many systems to help applications cope with deployment issues. Code-offloading systems, like COMET [85], MAUI [52], and CloneCloud [42], automatically offload computationally intensive tasks from mobile devices to cloud servers. Distributed storage systems [56, 40, 48] are a popular solution for server-side scalability, durability and fault-tolerance. Systems like PADS [27], PRACTI [26] and WheelFS [196] explored configurable deployment of application *data* but not run-time management of the entire application. Systems like Bayou [199], Cimbiosys [169] and Simba [6] offer client-side caching and offline access for weakly connected environments. Each of these systems only solves a subset of the deployment challenges that mobile/cloud applications face. Sapphire is the first distributed system to provide a unified solution to deployment for mobile/cloud applications.

When building Sapphire's DM library, we drew inspiration from existing mobile/cloud deploy-

ment systems, including those providing: wide-area communication [107], load-balancing [88, 219], geographic replication [131, 188], consensus protocols [119, 156], and DHTs [193, 175, 138].

Similar to our goal with Sapphire, previous language and compiler systems have tried to unify the distributed environment. However, unlike Sapphire, these solutions have no flexibility. They either make all deployment decisions for the application – an approach that doesn't work for the wide range of mobile/cloud requirements – or they leave all deployment up to the programmer. Compilers like Coign [97], Links [47], Swift [41] and Hop [181] automatically partition applications, but give programmers no control over performance trade-offs. Single language domains like Node.js [155] and Google Web Toolkit [84] create a uniform programming language across browsers and servers, but leave deployment up to the application. For mobile devices, MobileHTML5 [143], MobiRuby [144] and Corona [49] support a single cross-platform language. Sapphire supports a more complete cross-platform environment, but programmers can select deployments from an extensive (and extensible) library.

The DK's single address space and distributed object model are related to early distributed programming systems such as Argus [129], Amoeba [198] and Emerald [108]. Modern systems like Orleans [36] and Tango [23] provide cloud- or server-side services. Fabric [130] extends the work in this space with language abstractions that provide security guarantees. These systems were intended for homogeneous, local-area networks, so do not have the customizability and extensibility of the Sapphire DK.

Overall, existing or early distributed programming systems are not *general-purpose, flexible* or *extensible* enough to support the requirements of modern mobile/cloud applications. Therefore, in designing Sapphire, we drew inspiration from work that has explored customizability and extensibility in other contexts: operating systems [64, 30, 71, 180, 124], distributed storage [27, 53, 196, 183, 73], databases [38, 25], and routers and switches [112, 139].

2.9 Summary

This chapter presented Sapphire, a system that automatically manages distributed computation for mobile/cloud applications. Sapphire's Deployment Kernel creates an integrated run-time environment

with location-independent communication across mobile devices and the cloud. Its novel deployment layer contains a library of Deployment Managers that handle application-specific distribution issues, such as load-scaling, replication, and performance caching. Our experience shows that Sapphire: (1) greatly eases the programming of heterogeneous, distributed cloud/mobile applications, (2) provides great flexibility in choosing and changing deployment decisions, and (3) gives programmers fine-grained control over performance, availability, and scalability.

3 Diamond

As listed in Section 1.4, the another key operating system function is memory management. In contrast to desktop applications, memory management in a mobile/cloud application is especially challenging because mobile/cloud applications are *reactive* [106]: giving the illusion of continuous synchronization across users' devices without requiring them to explicitly save, reload, and exchange shared data. This trend towards reactive applications is not limited to mobile/cloud applications, but also includes the latest distributed versions of traditional desktop apps on both Windows [37] and OSX [8].

Combined with the other new mobile/cloud requirements listed in Section 1.3, offering reactivity presents a challenging *distributed memory management* problem for application programmers. mobile/cloud applications consist of *widely distributed* processes sharing data across mobile devices, desktops, and cloud servers. These processes make concurrent data updates, can stop or fail at any time, and may be connected by slow or unreliable links. While distributed storage systems [48, 199, 40, 62, 56] provide persistence and availability, programmers still face the formidable challenge of synchronizing updates between application processes and distributed storage in a fault-tolerant manner.

This chapter presents *Diamond*, the first *reactive data management service* (RDS) for wide-area applications that continuously synchronizes shared application data across distributed processes. Specifically, Diamond performs the following functions on behalf of an application: (1) it ensures that updates to shared data are consistent and durable, (2) it reliably coordinates and synchronizes shared data updates across processes, and (3) it automatically triggers *reactive code* when shared data

changes so that processes can perform appropriate tasks. For example, when a user updates data on one device (e.g., a move in a multi-player game), Diamond persists the update, reliably propagates it to other users' devices, and transparently triggers application code on those devices to react to the changes.

Reactive data management in the wide-area context requires a balanced consideration of performance trade-offs and reasoning about complex correctness requirements in the face of concurrency. Diamond implements the difficult mechanisms required by these applications (such as logging and concurrency control), letting programmers focus on high-level data-sharing requirements (e.g., atomicity, concurrency, and data layout). Diamond introduces three new concepts:

1. **Reactive Data Map** (`rmap`), a primitive that lets applications create *reactive data types* – shared, persistent data structures – and map them into the Diamond data management service so it can automatically synchronize them across distributed processes and persistent storage.

2. **Reactive Transactions**, an interactive transaction type that automatically *re-executes* in response to shared data updates. These "live" transactions run *application code* to make local, application-specific updates (e.g., UI changes).

3. **Data-type Optimistic Concurrency Control** (DOCC), a mechanism that leverages data-type semantics to concurrently commit transactions executing commutative operations (e.g., writes to different list elements, increments to a counter). Our experiments show that DOCC copes with wide-area latencies very effectively, reducing abort rates by up to 5x.

We designed and implemented a Diamond prototype in C++ with language bindings for C++, Python, and Java on both x86 and Android platforms. We evaluate Diamond by building and measuring both Diamond and custom versions (using explicit data management) of four reactive apps. Our experiments show that Diamond significantly reduces the complexity and size of reactive applications, provides strong transactional guarantees that eliminate data races, and supports automatic reactivity with performance close to that of custom-written reactive apps.

3.1 Background

Reactive applications require synchronized access to distributed shared data, similar to shared virtual memory systems [125, 28]. For practical performance in the wide-area environment, apps must be able to control: (1) *what* data in each process is shared, (2) *how* often it is synchronized, and (3) *when* concurrency control is needed. Existing applications use one of several approaches to achieve synchronization with control. This section demonstrates that these approaches are all complex, error-prone, and make it difficult to reason about application data consistency.

As an example, we analyze a simple social game based on the 100 game [1]. Such games are played by millions [202], and their popularity changes constantly; therefore, game developers want to build them quickly and focus on game logic rather than data management. Because game play increasingly uses real money (almost $2 billion last year [63]), their design parallels other reactive applications where correctness is crucial (e.g., apps for first responders [145] and payment apps [210, 191]).

In the 100 game, players alternately add a number between 1 and 10 to the current sum, and the first to reach 100 wins. Players make moves and can join or leave the game at different times; application processes can fail at any time. Thus, for safety, the game must maintain traditional ACID guarantees – atomicity, consistency, isolation and durability – as well as *reactivity* for data updates. We call this combination of properties ACID+R. While a storage system provides ACID guarantees for its own data, those guarantees *do not extend to application processes*. In particular, pushing updates to storage on mobile devices is insufficient for reactivity because application processes must re-compute local data *derived* from shared data to make changes *visible* to users and other components.

3.1.1 Roll-your-own Data Management

Many current reactive apps "roll-their-own" *application-specific* synchronization across distributed processes *on top of* general-purpose distributed storage (e.g., Spanner [48], Dropbox [62]. Figure 3.1 shows a typical three-tiered architecture used by these apps (e.g., PlayFish uses it to serve over 50 million users/month [92]). Processes on *client devices* access stateless *cloud servers*, which store persistent game state in a *distributed storage* system and use a reliable *notification service* (e.g., Thialfi [4])

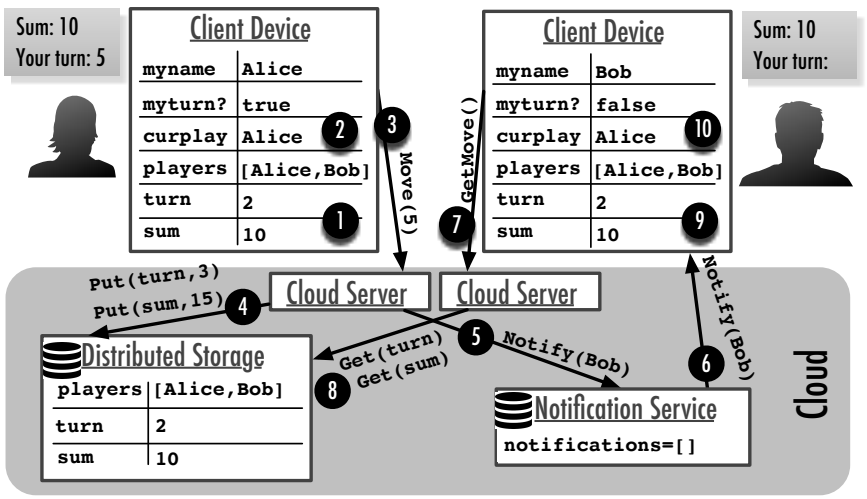


Figure 3.1: *Example 100 game architecture.* Each box is a separate address space. `players`, `turn` and `sum` are shared across address spaces and the storage system; `myturn?` and `curplay` are derived from shared data. When shared values change, the app manually updates distributed storage, other processes with the shared data, and any data in those processes derived from shared data, as shown by the numbered steps needed to propagate Alice's move to Bob.

to trigger changes in other processes for reactivity. While all application processes can fail, we assume strong guarantees – such as durability and linearizability – for the storage system and notification service. Although such apps could rely on a single server to run the game, this would create a centralized failure point. Clients cache game data to give users a responsive experience and to reduce load on the cloud servers [92].

The numbers in Figure 3.1 show the data management steps that the application must explicitly perform for Alice's move (adding 5 to the sum). Alice's client: (1) updates `turn` and `sum` locally, (2) calculates new values for `myturn?` and `curplay`, and (3) sends the move to a cloud server. The server: (4) writes `turn` and `sum` to distributed storage, and (5) sends a notification to Bob. The notification service: (6) delivers the notification to Bob's client, which (7) contacts a cloud server to get the latest move. The server: (8) reads from distributed storage and returns the latest `turn` and `sum`. Bob's client: (9) updates `turn` and `sum` locally, and (10) re-calculates `myturn?` and `curplay`.

Note that such data management must be customized to such games, making it difficult to implement a general-purpose solution. For example, only the application knows that: (1) clients share `turn` and `sum` (but not `myname`), (2) it needs to synchronize `turn` and `sum` after each turn (but not `players`), and (3) it does not need concurrency control because `turn` already coordinates moves.

Correctly managing this application data demands that the programmer reason about failures and data races at every step. For example, the cloud server could fail in the middle of step 4, violating atomicity. It could also fail between steps 4 and 5, making the application appear as if it is no longer reactive.

A new player, Charlie, could join the game while Bob makes his move, leading to a race; if Alice receives Bob's notification first, but Charlie writes to storage first, then both Alice and Charlie would think that it was their turn, violating isolation.

Finally, even if the programmer were to correctly handle every failure and data race *and* write bug-free code, reasoning about the consistency of application data would prove difficult. Enforcing a single global ordering of join, leave and move operations requires application processes to either forgo caching shared data (or data derived from shared data) altogether or invalidate all cached copies and update the storage system atomically on every operation. The first option is not realistic in a wide-area environment, while the second is not possible when clients may be unreachable.

3.1.2 Wide-area Storage Systems

A simple, alternative way to manage data manually is to store shared application data in a wide-area storage system (e.g., Dropbox [62]). That is, rather than calling `move` in step 3, the application stores and updates `turn` and `sum` in a wide-area storage system. Though simple, this design can be very expensive. Distributed file systems are not designed to frequently synchronize small pieces of data, so their coarse granularity can lead to moving more data than necessary and false sharing.

Further, while this solution synchronizes Alice's updates with the cloud, it does not ensure that Bob receives Alice's updates. To simulate reactive behavior and ensure that Bob sees Alice's updates, Alice must still use a wide-area notification system (e.g., Apple Push Notifications [15]) to notify Bob's client after her update. Unfortunately, this introduces a race condition: if Bob's client receives

the notification before the wide-area storage system synchronizes Alice's update, then Bob will not see Alice's changes. Worse, Bob will never check the storage system again, so he will never see Alice's update, leaving him unable to make progress. Thus, this solution retains all of the race conditions described in Section 3.1.1 *and* introduces some new ones.

3.1.3 Reactive Programming Frameworks

Several programming frameworks (e.g., Firebase [69], Parse [160] with React [170], Meteor [141]) have recently been commercially developed for reactive applications. These frameworks combine storage and notification systems and automate data management and synchronization across systems. However, they do not provide a clear consistency model, making it difficult for programmers to reason about the guarantees provided by their synchronization mechanisms. Further, they offer no distributed concurrency control, leaving application programmers to contend with race conditions; for example, they can lead to the race condition described in Section 3.1.1.

3.2 Architecture & Programming Model

Diamond is a new programming platform designed to simplify the development of wide-area reactive applications. This section specifies its data and transaction models and system call API.

3.2.1 System Model

Diamond applications consist of processes running on mobile devices and cloud servers. Processes can communicate through Diamond or over a network, which can vary from local IPC to the Internet. Every application process is linked to a client-side library, called *LibDiamond*, which provides access to the shared *Diamond cloud* – a highly available, fault-tolerant, durable storage system. Diamond subsumes some applications' server-side functionality, but our goal is not to eliminate such code. We expect cloud servers to continue providing reliable and efficient access to computation and datacenter services (e.g., data mining) while accessing shared data needed for these tasks through Diamond.

Figure 3.2 shows the 100 game data model using Diamond. Compared to Figure 3.1, the application

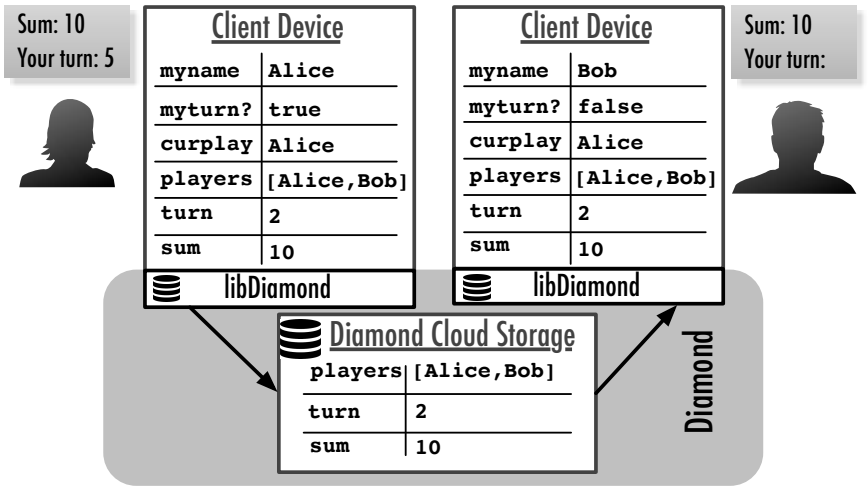


Figure 3.2: *Diamond 100 game data model.* The app `rmaps` `players`, `turn` and `sum`, updates them in read-write transactions and computes `myturn?` and `curplay` in a reactive transaction.

can directly read and write to shared data in memory, and Diamond ensures updates are propagated to cloud storage and other processes. Further, Diamond's strong transactional guarantees eliminate the need for programmers to reason about failures and concurrency.

3.2.2 Data Model

Diamond supports *reactive data types* for fine-grained synchronization, efficient concurrency control, and persistence. As with popular data structure stores [55], such as Redis [173] and Riak [174], we found that simple data types are general enough to support a wide range of applications and provide the necessary semantics to enable commutativity and avoid false sharing. Table 3.1 lists the supported persistent data types and their operations. In addition to primitive data types, like `String`, Diamond supports simple Conflict-free Replicated Data-types (CRDTs) [182] (e.g., `Counter`) and collection types (e.g., `LongSet`) with efficient type-specific interfaces. Using the most specific type possible provides the best performance (e.g., using a `Counter` for records that are frequently incremented).

A single Diamond *instance* provides a set of *tables*; each table is a key-to-data-type map, where each entry, or *record*, has a single persistent data type. Applications access Diamond through language

Table 3.1: *Reactive data types (RDTs).* Diamond's RDTs include primitives, collections and conflict-free replicated data types.

Type	Operations	Description
`Boolean`	`Get()`, `Put(bool)`	Primitive boolean
`Long`	`Get()`, `Put(long)`	Primitive number
`String`	`Get()`, `Put(str)`	Primitive string
`Counter`	`Get()`, `Put(long)`	Long Counter
	`Increment(long)`	
	`Decrement(long)`	
`IDGen`	`GetUID()`	Unique ID generator
`LongSet`	`Get(idx)`, `Contains(long)`	Ordered number set
	`Insert(long)`	
`LongList`	`Get(idx)`, `Set(idx, long)`	Number list
	`Append(long)`	
`StringSet`	`Get(idx)`, `Contains(str)`	Ordered string set
	`Insert(str)`	
`StringList`	`Get(idx)`, `Set(idx, str)`	String list
	`Append(str)`	
`HashTable`	`Get(key)`, `Set(key, val)`	Unordered map

bindings; however, applications need not be written in a single language. We currently support C++, Python and Java on both x86 and Android but could easily add support for other languages (e.g., Swift [14]).

3.2.3 System Calls

While apps interact with Diamond largely through reactive data types, we provide a minimal system call interface, shown in Table 3.2, to support transactions and `rmap`.

The `rmap` Primitive

`rmap` is Diamond's key abstraction for providing shared memory that is flexible, persistent, and reactive across wide-area application processes. Applications call `rmap` with an application variable and a key to the Diamond record, giving them control over what data in their address space is shared and how it is organized. In this way, different application processes (e.g., an iOS and an Android client) and different application versions (e.g., a new and current code release) can effectively share data. When `rmap`ping records to variables, the data types must match. Diamond's system call library checks at run time and returns an error from the `rmap` call if a mismatch occurs.

Transaction model

Application processes use Diamond transactions to read and write `rmap`ped variables. Diamond transactions are *interactive* [194], i.e., they let applications interleave application code with accesses to reactive data types. We support both standard *read-write transactions* and new *reactive transactions*. Applications cannot execute transactions across `rmap`ped variables from different tables, while operations executed outside transactions are treated as single-op transactions.

Read-write transactions. Diamond's read-write transactions let programmers safely and easily access shared reactive data types despite failures and concurrency. Applications invoke read-write transactions using `execute_txn`. The application passes closures for both the transaction and a completion callback. Within the transaction closure, the application can read or write `rmap`ped variables and variables in the closure, but it cannot modify program variables outside the closure. This limitation ensures: (1) the transaction can access all needed variables when it executes asynchronously (and they have not changed), and (2) the application is not affected by the side effects of aborted

Table 3.2: *Diamond system calls.* Applications use this interface to access tables, `rmap` and transactions.

System call	Description
`create(table, [isolation])`	Create table
`status = rmap(var, table, key)`	Bind `var` to `key`
`id = execute_txn(func, cb)`	Start read-write transaction
`id = register_reactxn(func)`	Start reactive transaction
`reactxn_stop(txn_id)`	Stop re-executing
`commit_txn()`, `abort_txn()`	Commit/Abort and exit

transactions. Writes to `rmap`ped variables are buffered locally until commit, while reads go to the client-side cache or to cloud storage.

Before `execute_txn` returns, Diamond logs the transaction, with its read and write sets, to persistent storage. This step guarantees that the transaction will eventually execute and that the completion callback will eventually execute even if the client crashes and restarts. This guarantee lets applications buffer transactions if the network is unavailable and easily implement custom retry functionality in the completion callback. If the callback reports that the transaction successfully committed, then Diamond guarantees ACID+R semantics for all accesses to `rmap`ped records; we discuss these in more detail in Section 3.2.4. On abort, Diamond rolls back all local modifications to `rmap`ped variables.

Reactive transactions. Reactive transactions help application processes automatically propagate changes made to reactive data types. Each time a read-write transaction modifies an `rmap`ped variable in a reactive transaction's read set, the reactive transaction re-executes, propagating changes to derived local variables. As a result, reactive transactions provide a "live" view that gives the illusion of reactivity while maintaining an imperative programming style comfortable to application programmers. Further, because they read a consistent snapshot of `rmap`ped data, reactive transactions avoid application-level bugs common to reactive programming models [135].

Applications do not explicitly invoke reactive transactions; instead, they register them by passing a closure to `register_reactxn`, which returns a `txn_id` that can be used to unregister the transaction with `reactxn_stop`. Within the reactive transaction closure, the application can read but not write `rmap`ped records, preventing potential data flow cycles. Since reactive transactions are designed to propagate changes to local variables, the application can read and write to local variables at any time and trigger side-effects (i.e., print-outs, updating the UI). Diamond guarantees that reactive transactions never abort because it commits read-only transactions locally at the client. Section 3.3 details the protocol for reactive transactions.

Reactive transactions run in a background thread, concurrently with application threads. Diamond transactions do not protect accesses to local variables, so the programmer must synchronize with locks or other mechanisms. The read set of a reactive transaction can change on every execution; Diamond tracks the read set from the latest execution. Section 3.5.2 explains how to use reactive transactions to build general-purpose, reactive UI elements.

3.2.4 Reactive Data Management Guarantees

Diamond's guarantees were designed to meet the requirements of reactive applications specified in Section 3.1, eliminating the need for each application to implement its own complex data management. To do so, Diamond enforces *ACID+R* guarantees for reactive data types:

- **Atomicity:** All or no updates to shared records in a read-write transaction succeed.
- **Consistency:** Accesses in all transactions reflect a consistent view of shared records.[1]
- **Isolation:** Accesses in all transactions reflect a global ordering of committed read-write transactions.
- **Durability:** Updates to shared records in committed read-write transactions are never lost.

[1]The C in ACID is not well defined outside a database context. Diamond simply guarantees that each transaction reads a consistent snapshot.

Table 3.3: *Diamond's isolation levels.* Isolation levels for read-write transactions and associated ones for reactive transactions.

Read-write Isolation Level	Reactive Isolation Level
Strict Serializability	Serializable Snapshot
Snapshot Isolation	Serializable Snapshot
Read Committed	Read Committed

- **Reactivity:** Accesses to modified records in registered reactive transactions will eventually re-execute.

These guarantees create a *producer-consumer* relationship: Diamond's read-write transactions *produce* updates to reactive data types, while reactive transactions *consume* those updates and propagate them to locally derived data. However, unlike the traditional producer-consumer paradigm, this mechanism is *transparent* to applications because the ACID+R guarantees ensure that Diamond *automatically* re-executes the appropriate reactive transactions when read-write transactions commit.

Table 3.3 lists Diamond's isolation levels, which can be set per table. Diamond's default is strict serializability because it eliminates the need for application programmers to deal with inconsistencies caused by data races and failures. Lowering the isolation level leads to fewer aborts and more concurrency; however, more anomalies arise, so applications should either expect few conflicts, require offline access, or tolerate inaccuracy (e.g., Twitter's most popular hash tag statistics). Section 3.4.1 describes how DOCC increases concurrency and reduces aborts for transactions even at the highest isolation levels.

3.2.5 A Simple Code Example

To demonstrate the power of Diamond to simplify reactive applications, Figure 3.3 shows code to implement the 100 game from Section 3.1 in Diamond. This implementation provides persistence,

```
int main(int argc, char **argv) {
   DStringSet players; DCounter sum, turn;
   string myname = string(argv[1]);

   // Map game state
   create("100game", STRICT_SERIALIZABLE);
   rmap(players, "100game", "players");
   rmap(sum, "100game", "sum");
   rmap(turn, "100game", "turn");

   // General-purpose callback, exit if txn failed
   auto cb = [] (txn_func_t txn, int status) {
      if (status == REPLY_FAIL) exit(1); };

   // Add user to the game
   execute_txn([myname] () {
      players.Insert(myname); }, cb);

   // Set up our print outs
   register_reactxn([myname] () {
      string curplay =
         players[turn % players.size()];
      bool myturn = myname == curplay;
      cout << "Sum: " << sum << "\n";
      if (sum >= 100)
         cout << curplay << " won!";
      else if (myturn)
         cout << "Your turn: ";
   });

   // Cycle on user input
   while (1) {
      int inc; cin >> inc;
      execute_txn([myname, inc] () {
         bool myturn =
            myname == players[turn % players.size()];
         // check inputs
         if (!myturn || inc < 1 || inc > 10) {
            abort_txn(); return;
         }
         sum += inc; if (sum < 100) turn++;
      }, cb);
   }
   return 0;
}
```

Figure 3.3: *Diamond code example.* Implementation of the 100 game using Diamond. Omitting includes, set up, and error handling, this code implements a working, C++ version of the 100 game [1]. `DStringSet`, `DLong` and `DCounter` are reactive data types provided by the Diamond C++ library.

atomicity, isolation and reactivity for every join and move operation in only 34 lines of code. We use three reactive data types for shared game data, declared on line 2 and `rmap`ped in lines 7-9. It is important to ensure a strict ordering of updates, so we create a table in strict serializable mode on line 6. On line 12, we define a general-purpose transaction callback for potential transaction failures. On line 16, we execute a read-write transaction to add the player to the game, passing `myname` by value into the transaction closure. Using DOCC allows Diamond to commit two concurrent executions of this transaction while guaranteeing strict serializability.

Line 20 registers a reactive transaction to print out the score and current turn. Diamond's ACID+R guarantees ensure that the transaction re-executes if `players`, `turn` or `sum` change, so the user always has a consistent, up-to-date view. Note that we can print to stdout because the reactive transaction will not abort, and the printouts reflect a serializable snapshot, avoiding reactive glitches [135]. On line 32, we wait for user input in the `while` loop and use a read-write transaction to commit the entered move.

Diamond's strong guarantees eliminate the need for programmers to reason about data races or failures. Taking our examples from Section 3.1, Diamond ensures that when the game commits Alice's move, the move is never lost and Bob eventually sees it. Diamond also ensures that, if Charlie joins before Bob makes his move, Alice either sees Charlie join without Bob's move, or both, but never sees Bob's move without seeing Charlie join. As a result, programmers no longer need to reason about race conditions, greatly simplifying the game's design. To our knowledge, no other system provides all of Diamond's ACID+R properties.

3.2.6 Offline Support

Wi-Fi and cellular data networks have become widely available, and reactive applications typically have limited offline functionality; thus, Diamond focuses on providing *online reactivity*, unlike storage systems (e.g., Bayou [199] and Simba [161]). However, Diamond still provides limited offline support. If the network is unavailable, `execute_txn` logs and transparently retries, while Diamond's CRDTs make it more likely that transactions commit after being retried. For applications with higher contention, Diamond's read committed mode enables commits locally at the client while offline, and

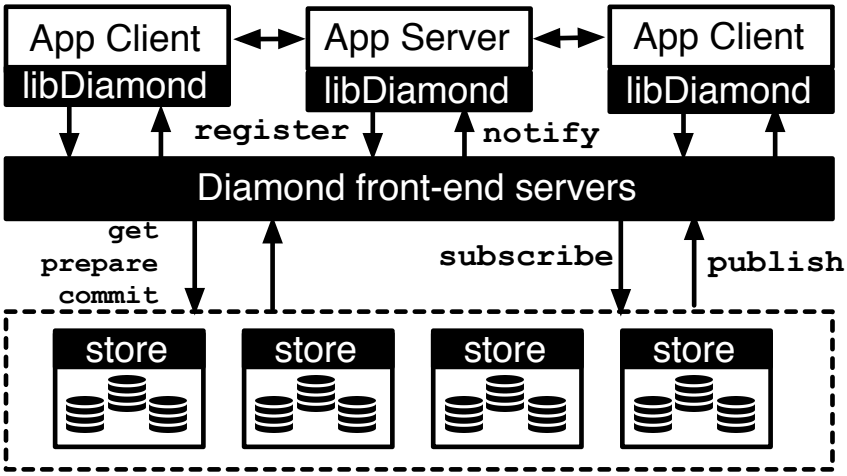


Figure 3.4: *Diamond architecture.* Distributed processes share a single instance of the Diamond storage system.

any modifications eventually converge to a consistent state for Diamond's CRDTs.

3.2.7 Security

Similar to existing client-focused services, like Firebase [69] and Dropbox [62], Diamond trusts application clients not to be malicious. Application clients authenticate with the Diamond cloud through their LibDiamond client before they can `rmap` or access reactive data types. Diamond supports isolation between users through *access control lists* (ACLs); applications can set `rmap`, read, and write permissions per table. Within tables, keys function as capabilities; a client with a key to a record has permission to access it. Applications can defend against potentially malicious clients by implementing server-side security checks using reactive transactions on a secure cloud server.

3.3 System Design

This section relates Diamond's architecture, the design of `rmap`, and its transaction protocols.

3.3.1 Data Management Architecture

Figure 3.4 presents an overview of Diamond's key components. Each LibDiamond client provides client-side caching and access to cloud storage for the application process. It also registers, tracks and re-executes reactive transactions and keeps a persistent transaction log to handle device and network failures.

The Diamond cloud consists of *front-end servers* and *back-end storage* servers, which together provide durable storage and reliable notifications for reactive transactions. Front-end servers are scalable, stateless nodes that provide LibDiamond clients access to Diamond's back-end storage, which is partitioned for scalability and replicated (using Viewstamped Replication (VR) [156]) for fault tolerance. LibDiamond clients could directly access back-end storage, but front-end servers give clients a single connection point to the Diamond cloud, avoiding the need for them to authenticate with many back-end servers or track the partitioning scheme.

3.3.2 `rmap` and Language Bindings

Diamond language bindings implement the library of reactive data types for apps to use as `rmap` variables. Diamond *interposes* on every operation to an `rmap`ped variable. During a transaction, LibDiamond collects an *operation set* for Docc to later check for conflicts. Reads may hit the LibDiamond client-side cache or require a wide-area access to the Diamond cloud, while writes (and increments, appends, etc.) are buffered in the cache until commit.

3.3.3 Transaction Coordination Overview

Figure 3.5 shows the coordination needed across LibDiamond clients, front-end servers and back-end storage for both read-write and reactive transactions. This section briefly describes the transaction protocols.

Diamond uses *timestamp ordering* to enforce isolation across LibDiamond clients and back-end storage; it assigns every read-write transaction a unique *commit timestamp* that is provided by a replicated *timestamp service* (tss) (not shown in Figure 3.4). Commit timestamps reflect the

transaction commit order, e.g., in strict serializability mode, they reflect a single linearizable ordering of committed, read-write transactions. Both Diamond's client-side cache and back-end storage are *multi-versioned* using these commit timestamps.

Running Distributed Transactions

Read-write and reactive transactions execute similarly; however, as Section 3.4 relates, reactive transactions can commit locally and often avoid wide-area accesses altogether. We lack the space to cover Diamond's transaction protocol in depth; however, it is similar to Spanner's [48] with two key differences: (1) Diamond uses DOCC for concurrency control rather than a locking mechanism, and (2) Diamond uses commit timestamps from the timestamp service (tss) rather than TrueTime [48].

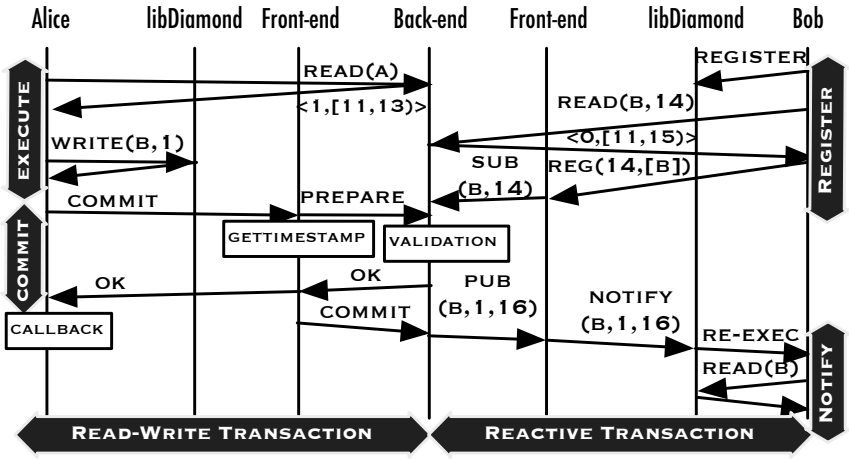


Figure 3.5: *Diamond transaction coordination.* Left: Alice executes a read-write transaction that reads `A` and writes `B`. Right: Bob registers a reactive transaction that reads `B` (we omit the `txn_id`). When Alice commits her transaction, the back-end server publishes the update to the front-end, which pushes the notification and the update to Bob's LIBDIAMOND, which can then re-execute the reactive transaction locally.

As shown in Figure 3.5 (left), transactions progress through two phases, *execution* and *commit*. During the execution phase, LIBDIAMOND runs the application code in the transaction closure passed

into `txn_execute`. It runs the code locally on the LibDiamond client node (i.e., not on a storage node like a stored procedure).

The execution phase completes when the application exits the transaction closure or calls `txn_commit` explicitly. Reactive transactions commit locally; for read-write transactions, LibDiamond sends the operation sets to the front-end server, which acts as the *coordinator* for a *two-phase commit* (2PC) protocol, as follows:

1. It sends `Prepare` to all participants (i.e., partitions of the Diamond back-end that hold records in the operation sets), which replicate it via VR.

2. Each participant runs a Docc validation check (described in Section 3.4); if Docc validation succeeds, the participant adds the transaction to a *prepared list* and returns `true`; otherwise, it returns `false`.

3. As an optimization, the front-end server concurrently retrieves a commit timestamp from the tss.

4. If all participants respond `true`, the front-end sends `Commits` to the participants with the commit timestamp; otherwise, it sends `Aborts`. Then, it returns the transaction outcome to the LibDiamond client.

When the client receives the response, it logs the transaction outcome and invokes the transaction callback.

Managing Reactive Transactions

As shown in Figure 3.5 (right), when an application registers a reactive transaction, the LibDiamond client: (1) gives the reactive transaction a `txn_id`, (2) executes the reactive transaction at its latest known timestamp, and (3) sends the `txn_id`, the timestamp, and the read set in a `Register` request to the front-end server. For each key in the read set, the front-end server creates a `Subscribe` request and sends those requests, along with the timestamp, to each key's back-end partition.

For efficiency, LibDiamond tracks read set changes between executions and re-registers. We expect each reactive transaction's read set to change infrequently, reducing the overhead of registrations; if it changes often, we can use other techniques (e.g., `map_objectrange` described in Section 3.5.2) to improve performance.

When read-write transactions commit, Diamond executes the following steps for each updated record:

1. The leader in the partition sends a `Publish` request with the transaction's commit timestamp to each front-end subscribed to the updated record.

2. For each `Publish`, the front-end server looks up the reactive transactions that have the updated record in their read sets and checks if the commit timestamp is bigger than the last notification sent to that client.

3. If so, the front-end server sends a `Notify` request to the client with the commit timestamp and the reactive transaction id.

4. The client logs the notification on receipt, updates its latest known timestamp, and re-executes the reactive transaction at the commit timestamp.

For keys that are updated frequently, back- and front-end servers batch updates. Application clients can bound the batching latency (e.g., to 5 seconds), ensuring that reactive transactions refresh at least once per batching latency when clients are connected.

Handling Failures

While both the back-end storage and tss are replicated using VR, Diamond can suffer failures of the LibDiamond clients or front-end servers. On client failure, LibDiamond runs a *client recovery protocol* using its transaction log to ensure that read-write transactions eventually commit. For each completed but unprocessed transaction (i.e., in the log but with no outcome), LibDiamond retries the commit. If the cloud store has a record of the transaction, it returns the outcome; otherwise, it

re-runs 2PC. For each reactive transaction, the application re-registers on recovery. LibDiamond uses its log to find the last timestamp at which it ran the transaction.

Although front-end servers are stateless, LibDiamond clients must set up a new front-end server connection when they fail. They use the client recovery protocol to do this and re-register each reactive transaction with its latest notification timestamp. Front-end servers also act as coordinators for 2PC, so back-end storage servers use the *cooperative termination protocol* [29] if they do not receive `Commit` requests after some timeout.

3.4 Wide-area Optimizations

This section discusses Diamond's optimizations to reduce wide-area overhead.

3.4.1 Data-type Optimistic Concurrency Control

Diamond uses an optimistic concurrency control (OCC) mechanism to avoid locking across wide-area clients. Unfortunately, OCC can perform poorly across the wide area due to the higher latency between a transaction's read of a record and its subsequent commit. This raises the likelihood that a concurrent write will invalidate the read, thereby causing a transaction abort. For example, to increment a counter, the transaction reads the current value, increments it, and then commits the updated value; if another transaction attempts the same operation at the same time, an abort occurs.

Docc tackles this issue in two ways. First, it uses fine-grained concurrency control based on the *semantics* of reactive data types, e.g., allowing concurrent updates to different list elements. Second, it uses conflict-free data types with commutative operations, such as counters and ordered sets. As noted in Section 3.3.3, LibDiamond collects an operation set for every data type operation during the transaction's execution phase. For each operation, it collects the key and table. It also collects the read version for every `Get`, the written value for every `Put`, the index (e.g., list index or hash table key) for every collection operation, and the diff (e.g., the increment value or the insert or append element) for every commutative CRDT operation. We show in Section 3.5 that although fine-grained tracking slightly increases Docc overhead, it improves overall performance.

Using operation sets, DOCC runs a *validation* procedure that checks every committing transaction for potential violations of isolation guarantees. A *conflicting access* occurs for an operation if the table, key, and index (for collection types) match an operation in a prepared transaction. For a read, a conflict also occurs if the latest write version (or commutative CRDT operation) to the table, key, and index is bigger than the read version. For each, DOCC makes an abort decision, as noted in Table 3.4.

Table 3.4: *DOCC validation matrix.* Matrix shows whether the committing transaction can commit (C) or must abort (A) on conflicts. Each column is further divided by the isolation level (RC=read committed, SI=snapshot isolation, SS=strict serializability). Commutative CRDT operations have the same outcome.

Prepared Op / Isolation Level / Commiting Op	read			write			CRDT op		
	RC	**SI**	**SS**	**RC**	**SI**	**SS**	**RC**	**SI**	**SS**
read	C	C	C	C	C	A	C	C	A
write	C	C	A	C	A	A	C	A	A
CRDT op	C	C	A	C	A	A	C	C	C

Since transactions that contain only commutative operations can concurrently commit, DOCC can allow many concurrent transactions that modify the same keys. This property is important for workloads with high write contention, e.g., the Twitter "like" counter for popular celebrities [96]. Further, because Diamond runs read-only and reactive transactions in serializable snapshot mode, they do not conflict with read-write transactions with commutative CRDT operations.

3.4.2 Client Caching with Bounded Validity Intervals

Some clients in the wide-area setting may occasionally be unavailable, making it impossible to atomically invalidate all cache entries on every write to enforce strong ordering. Diamond therefore uses multi-versioning in both the client-side cache and back-end storage to enforce a *global ordering of transactions*. To do this, it tags each version with a *validity interval* [166], which begins at the *start*

timestamp and is terminated by the *end timestamp*. In Diamond's back-end storage, a version's start timestamp is the commit timestamp of the transaction that wrote the version. The end timestamp is either the commit timestamp of the transaction writing the *next* version (making that version out-of-date) or *unbounded* for the latest version. Figure 3.6 shows an example of back-end storage with three keys.

On reads, the Diamond cloud tags the returned value with a validity interval for the LibDiamond client-side cache. These validity intervals are conservative; back-end storage *guarantees* that the returned version is valid *at least* within the validity interval, although it may be valid beyond. If the version is the latest, back-end storage will *bound* the validity interval by setting the end timestamp to the latest commit timestamp of a transaction that accessed that record. For example, in Figure 3.6, the validity interval of the latest version of `B` and `C` are capped at timestamp 16 in the cache, while they are unbounded in storage. Most importantly, bounded validity intervals *eliminate* the need for cache invalidations because the version is always valid within the validity interval. Diamond eventually garbage collects cached versions as they become too outdated to use.

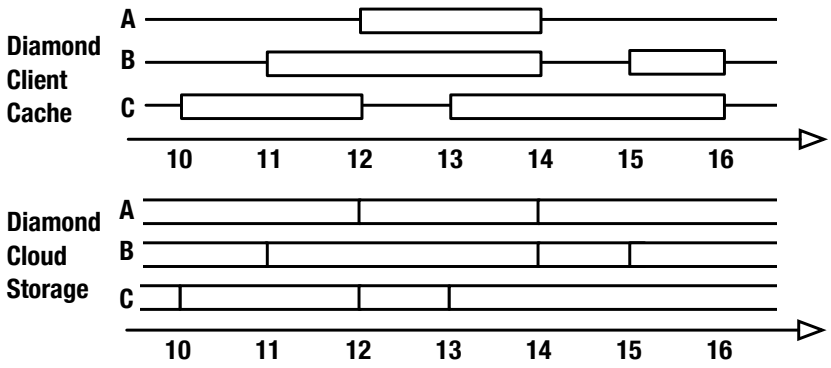


Figure 3.6: *Diamond versioned cache.* Every Diamond client has a cache of the versions of records stored by the Diamond cloud storage system. The bottom half shows versions for three keys (A, B and C), and the top half shows cached versions of those same keys. Note that the cache is missing some versions, and all of the validity intervals in the cache are bounded.

3.4.3 Data Push Notifications

Reactive transactions require many round-trips to synchronously fetch each update; these can be expensive in a wide-area network. Fortunately, unlike stand-alone notifications services (e.g., Thialfi), Diamond has insight into what data the application is likely to access when the reactive transaction re-executes. Thus, Diamond uses *data push notifications* to batch updates along with notifications, reducing wide-area round trips.

When front-end servers receive `Publish` requests from back-end storage, they perform a snapshot read of every key in the reactive transaction's last read set at the updating transaction's commit timestamp, then piggyback the results with the `Notify` request to the LibDiamond client. LibDiamond re-executes the reactive transaction at the commit timestamp; therefore, if its read set has not changed, then it requires no additional wide-area requests. Further, since the reads were done at the commit timestamp, LibDiamond knows that the transaction can be serialized at that timestamp and committed locally, eliminating all wide-area communication.

3.5 Experience & Evaluation

This section evaluates Diamond with respect to both programming ease and performance. Overall, our results demonstrate that Diamond simplifies the design of reactive applications, provides stronger guarantees than existing custom solutions, and supports automated reactivity with low performance overhead.

3.5.1 Prototype Implementation

We implemented a Diamond prototype in 11,795 lines of C++, including support for C++, Python and Java language bindings on both x86 and ARM. The Java bindings (939 LoC) use javacpp [104], and the Python bindings (115 LoC) use Boost [3]. We cross-compiled Diamond and its dependencies for Android using the NDK standalone toolchain [77]. We implemented most Diamond data types, but not all are supported by Docc. Our current prototype does not include client-side persistence and relies on in-memory replication for the back-end store; however, we expect disk latency on SSDs

to have a low performance impact compared to wide-area network latency, with NVRAM reducing storage latency even further in the future.

3.5.2 Programming Experience

This section evaluates our experience in building new Diamond apps, porting existing apps to Diamond, and creating libraries to support the needs of reactive programs.

Simplifying Reactive Applications

To evaluate Diamond's programming benefits, we implemented applications both with and without Diamond. Table 3.5 shows the lines of code for both cases. For all of the apps, Diamond simultaneously decreased program size and added important reliability or correctness properties. We briefly describe the programs and results below.

Table 3.5: *Application comparison.* Diamond both reduces code size and adds to the application's ACID+R guarantees.

Application	**Lines of Code w/o Diamond**	**Lines of Code w/ Diamond**	**% Saved**	**Added Properties**				
				A	**C**	**I**	**D**	**R**
100 Game	46	34	26%	✓	✓	✓		
Chat Room	355	225	33%	✓	✓	✓		✓
PyScrabble	8729	7603	13%	✓			✓	
Twitter clone	14278	12554	13%	✓	✓	✓		

100 Game. Our non-Diamond version of the 100 game is based on the design in Figure 3.1. For simplicity, we used Redis [173] for both storage and notifications. We found several data races between storage updates and notifications when running experiments for Figure 3.9, forcing us to include

updates in the notifications to ensure clients did not read stale data from the store. The Diamond version eliminated these bugs and the complexities described in Section 1.4 and guaranteed correctness with atomicity and isolation; in addition, it reduced the code size by 26%.

Chat Room. As another simple but representative example of a reactive app, we implemented two versions of a chat room. Our version with explicit data management used Redis for storage and the Jetty [105] web server to implement a REST [68] API. It used `POST` requests to send messages and polled using `GET` requests for displaying the log. This design is similar to that used by Twitter [206, 94] to manage its reactive data (e.g., Twitter has `POST` and `GET` requests for tweets, timelines, etc.). The Diamond version used a `StringList` for the chat log, a read-write transaction to append messages, and a reactive transaction to display the log. In comparison, Diamond not only eliminated the need for a server or storage system, it also provided atomicity (the Redis version has no failure guarantees), isolation (the Redis version could not guarantee that all clients saw a consistent view of the chat log), and reactivity (the Redis version polled for new messages). Diamond also shrunk the 355-line app by 130 lines, or 33%.

PyScrabble and Diamond Scrabble. To evaluate the impact of reactive data management in an existing application, we built a Diamond version of PyScrabble [44], an open-source, multiplayer Scrabble game. The original PyScrabble does not implement persistence (i.e., it has no storage system) and uses a centralized server to process moves and notify players. The centralized server enforces isolation and consistency only if there are no failures. We made some changes to add persistence and accommodate Diamond's transaction model. We chose to directly `rmap` the Scrabble board to reactive data types and update the UI in a reactive transaction, so our implementation had to commit and share every update to make it visible to the user; thus, other users could see the player lay down tiles in real-time rather than at the end of the move, as in the original design. Overall, our port of PyScrabble to Diamond removed over 1000 lines of code from the 8700-line app (13%) while transparently simplifying the structure (removing the server), adding fault tolerance (persistence) and atomicity, and retaining strong isolation.

Twimight and Diamond Dove. As another modern reactive application, we implemented a subset of Twitter using an open-source Android Twitter client (Twimight [205]) and a custom back-end. The Diamond version eliminated much of the data management in the Twimight version, i.e., pushing and retrying updates to the server and maintaining consistency between a client-side SQLite [190] cache and back-end storage. Diamond directly plugged into UI elements and published updates with read-write transactions. As a result, it simplified the design, eliminated 1700 lines (13%) from the 14K-line application, transparently provided stronger atomicity and isolation guarantees, and eliminated inconsistent behaviors (e.g., a user seeing a retweet before the original tweet).

Simplifying Reactive Libraries

In addition to simplifying the design and programming of reactive apps, we found that Diamond facilitates the creation of general-purpose reactive libraries. As one example, Diamond transactions naturally lend themselves to managing UI elements. For instance, a check box usually `rmap`s a `Boolean`, re-draws a UI element in a reactive transaction, and writes to the `Boolean` in a read-write transaction when the user checks/unchecks the box. We implemented a general library of Android UI elements, including a text box and check box. Each element required under 50 lines of code yet provided strong ACID+R guarantees. Note that these elements tie the user's UI to shared data, making it impossible to update the UI only locally; for example, if a user wants to preview a message before sharing it with others, the app must update the UI in some other way.

For generality, Diamond makes no assumptions about an app's data model, but we can build libraries using `rmap` for common data models. For example, we implemented object-relational mapping for Java objects whose fields were Diamond data types. Using Java reflection, `rmap_object` maps each Diamond data type inside an object to a key derived from a base key and the field's name. We also support `rmap` for subsets of Diamond collections, e.g., `rmap_range` for Diamond's primitive list types, which binds a subset of the list to an array, and `rmap_objectrange`, which maps a list of objects using `rmap_object`.

These library functions were easy to build (under 75 lines of code) and greatly simplified several applications; for example, our Diamond Twitter implementation stores a user's timeline as a `LongList`

of tweet ids and uses `map_objectrange` to directly bind the tail of the user's timeline into a custom Android adapter, which then plugs into the Twimight Android client and automatically manages reactivity. In addition to reducing application complexity, these abstractions also provide valuable hints for prefetching and for how reactive transaction read sets might change. Overall, we found Diamond's programming model to be extremely flexible, powerful, and easy to generalize into widely useful libraries.

3.5.3 Performance Evaluation

Our performance measurements demonstrate that Diamond's automated data management and strong consistency impose a low performance cost relative to custom-written applications. Using transactions with strong isolation properties lowers throughput, as one would expect. We also show that Diamond's DOCC improves performance of transactional guarantees, and that data push notifications reduce the latency of wide-area transactions. Finally, our experiments prove that Diamond has low overhead on mobile devices and can recover quickly from failures.

Experimental Setup

We ran experiments on Google Compute Engine [80] using 16 front-end servers and 5 back-end partitions, each with 3 replicas placed in different availability zones in the same geographic region (US-Central). Our replication protocol used adaptive batching with a batch size of 64. We placed clients in a different geographic region in the same country (US-East). The latency between zones was ≈1 ms, while the latency between regions was ≈36 ms. For our mobile device experiments, we used Google Nexus 7 LRX22G tablets connected via Wi-Fi and, for desktop experiments, we used a Dell workstation with an Intel Xeon E5-1650 CPU and 16 GB RAM.

We used a benchmark based on Retwis [122], a Redis-based Twitter clone previously used to benchmark transactional storage systems [220]. The benchmark was designed to be a representative, although not realistic, reflection of a Twitter-like workload that provides control over contention. It ran a mix of five transactions that range from 4-21 operations, including: loading a user's home

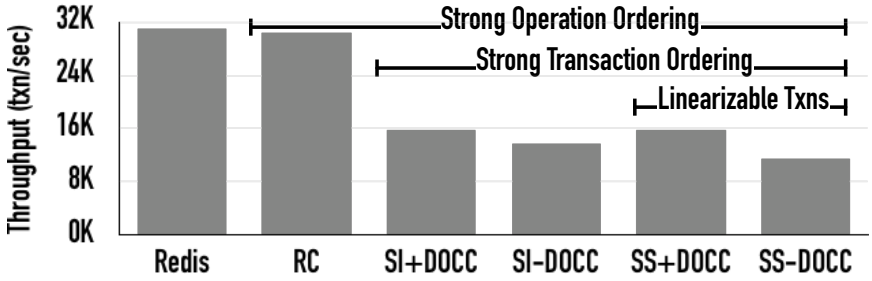


Figure 3.7: *Peak throughput for explicit data management vs Diamond.* We compare an implementation using Redis and Jetty to Diamond at different isolation levels with and without DOCC. We label the ordering guarantees provided by each configuration. In all cases, the back-end servers were the bottleneck.

timeline (50%), posting a tweet (20%), following a user (5%), creating a new user (1%), and "like"-ing a tweet (24%). To increase contention, we used 100K keys and a Zipf distribution with a co-efficient of 0.8.

Overhead of Automated Data Management

For comparison, we built an implementation of the Retwis benchmark that explicitly manages reactive data using Jetty [105] and Redis [173]. The Redis `WAIT` command offers synchronous in-memory replication, which matches Diamond's fault-tolerance guarantees but provides no operation or transaction ordering [172]. The leftmost bar in Figure 3.7 shows the peak Retwis throughput of 31K trans./sec. for the Redis-based implementation, while the second bar in Figure 3.7 shows the Diamond read-committed (RC) version, whose performance (30.5K trans./sec.) is nearly identical. Unlike the Redis-based implementation, however, the Diamond benchmark provides strong consistency based on VR, i.e., it enforces a single global order of operations but not transactions. The Diamond version also provides all of its reactivity support features. Diamond therefore provides better consistency properties and simplifies programming at little additional cost.

As we add stronger isolation through transactions, throughput declines because two-phase commit

requires each back-end server to process an extra message per transaction. As the graph shows, snapshot isolation (SI) and strict serializability (SS) reduce throughput by nearly 50% from RC. The graph also shows SI and SS both with and without DOCC; eliminating DOCC hurts SS more than SI (27% vs. 13%) because SI lets transactions with read-write conflicts commit (leading to write skew).

From this experiment, we conclude that Diamond's general-purpose data management imposes almost no throughput overhead. Also, achieving strong transactional isolation guarantees does impose a cost due to the more complex message protocol required. Depending on the application, programmers can choose to offset the cost by allocating more servers or tolerate inconsistencies that result from weaker transactional guarantees.

Benefit of DOCC

DOCC's benefit depends on both contention and transaction duration. To evaluate this effect, we measured the throughput improvement of DOCC for each type of Retwis transaction with at least one CRDT operation (Figure 3.8).

The `add_user` and `like` transactions are short and thus unlikely to abort, but they still see close to a 2x improvement. `add_follower` gets a larger benefit (4x) because it is a longer transaction with more commutative operations. Even `get_timeline`, a read-only transaction, gets a tiny improvement (2.5%) due to reduced load on the servers from aborting transactions. Further, because `get_timeline` runs in serializable snapshot mode, `post_tweet` transactions can commit concurrently with `get_timeline` transactions.

The `post_tweet` transaction appends a user's new tweet to his timeline and his followers' home timelines (each user has between 5 and 20 followers). If a user follows a large number of people that tweet frequently, conventional OCC makes it highly likely that a conflicting `Append` would cause the entire transaction to fail. With DOCC, all `Appends` to a user's home timeline can commute, avoiding these aborts. As a result, we saw a 5x improvement in abort rate with DOCC over conventional OCC for `post_tweet`, leading to a 25x improvement in throughput. Overall, these results show that Diamond's support for data types in its API and concurrency control mechanism is crucial to reducing the cost of transactional guarantees.

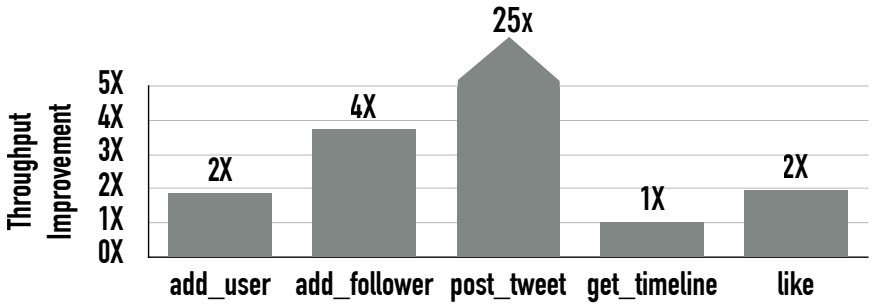


Figure 3.8: *Throughput improvement with* Docc *for each Retwis transaction type.*

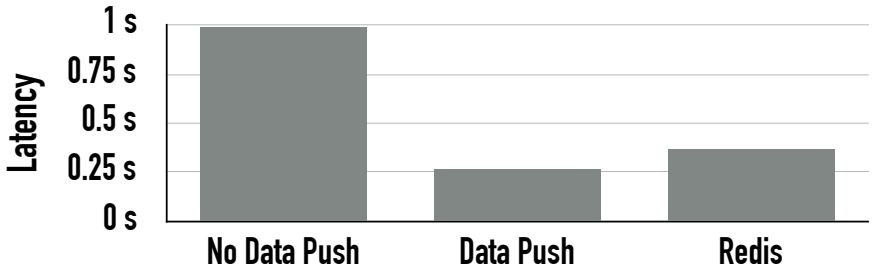


Figure 3.9: *Latency comparison for 100 game rounds with data push notifications.* Each round consist of 1 move by each of 2 players; latency is measured from 1 client. We implemented explicit data management and notifications using Redis and Diamond notifications with and without batched updates.

Benefit of Data Push Notifications

Although Diamond's automated data management imposes a low throughput overhead, it can hurt latency due to wide-area round trips to the Diamond cloud. For example, the latency of a Retwis transaction is twice as high for Diamond relative to our Redis implementation because Diamond requires two round trips per transaction, one to read and one to commit, while Redis needs only one.

Data push notifications reduce this latency by batching updates with reactive transaction notifications to populate the client-side cache. We turned our implementation of the 100 game from Figure 3.3 into a benchmark: two players join each game, and players make a move as soon as the

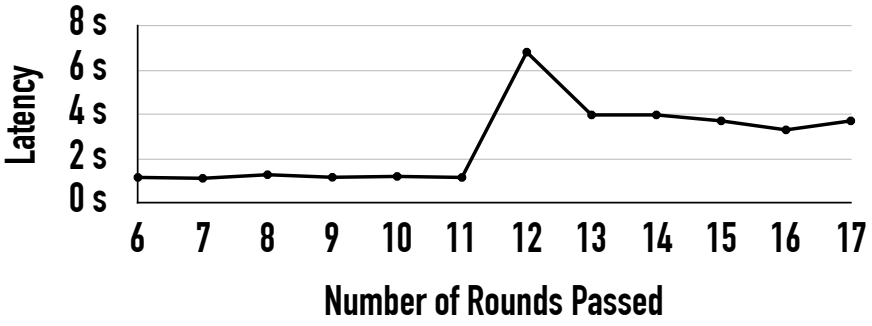


Figure 3.10: *Latency of 100 game rounds during failure.* We measured the latency for both players to make a move and killed the leader of the storage partition after about 15 seconds. After recovery, the leader moves to another geographic region, increasing overall messaging latency on each move.

other player finishes (i.e., zero "think" time). This experiment is ideal because the read set of the reactive transaction does not change, and it overlaps with the read set of the read-write transaction. We also design an implementation using Redis, where notifications carry updates to clients as a manual version of data push notifications. We measure the latency from one player's client for each player to take a turn or for one *round* of the game. Figure 3.9 shows that data push notifications reduce the overall latency by almost 50% by eliminating wide-area reads for both the reactive and read-write transactions in the game. As a result, Diamond has 30% lower latency and stronger transactional guarantees than our Redis implementation.

Impact of Wide-area Storage Server Failures

Failures affect the latency of both reactive and read-write transactions. To measure this impact, we used the same 100 game workload and killed a back-end server during the game. To increase the recovery overhead, we geo-replicated the back-end servers across Asia, US-Central and Europe, while clients remained in US-East.

Figure 3.10 shows the latency of each round. Note that the latency is higher than that in the previous experiment because the VR leader has to wait for a response from a quorum of replicas,

which take at least 100 ms, and up to 150 ms, to contact. About 15 seconds into the game, we kill the leader in US-Central, switching it to Europe. The latency of each round increases to almost 4 seconds afterwards: the latency between the front-end servers and the leader in Europe increases to 100 ms, and the latency from the leader to the remaining replica in Asia increases to 250 ms. Despite this, the round during the failure takes only 7 seconds, meaning that Diamond can detect the failure and replace the leader in less than 3 seconds.

End-user Application Latency

To evaluate Diamond's impact on the user experience, we measure the latency of user operations in two apps from Section 3.5.2 built with and without Diamond. PyScrabble is a desktop application, while our Chat Room app runs on Android. The ping times to the Diamond cloud were ≈38 ms on the desktop and ≈46 ms on the Android tablet.

Figure 3.11 (left) shows two operations for PyScrabble: `MakeMove` commits a transaction that updates the user's move, and `DisplayMove` includes MakeMove plus the notification and reactive transaction to make it visible. Compared to the original PyScrabble, Diamond's latency is slightly higher (9% and 16%, respectively). Figure 3.11 (right) shows operations for the Chat Room on an Android tablet. `ReadLog` gets the full chat log, and `PostMessage` gets the chat log, appends a message, and commits it back. The Diamond version is a few percent faster than the Redis version because it runs in native C++, while the Redis version uses a Java HTTP client. Overall, we found the latency differences between Diamond and non-Diamond operations were not perceivable to users.

3.6 Related Work

Diamond takes inspiration from wide-area storage systems, transactional storage systems and databases, reactive programming, distributed programming frameworks, shared memory systems and notification systems.

Several commercial platforms [141, 69, 160] provide an early form of reactive data management without distributed transactions. Other open source projects [100, 148, 57, 157, 187] have replicated

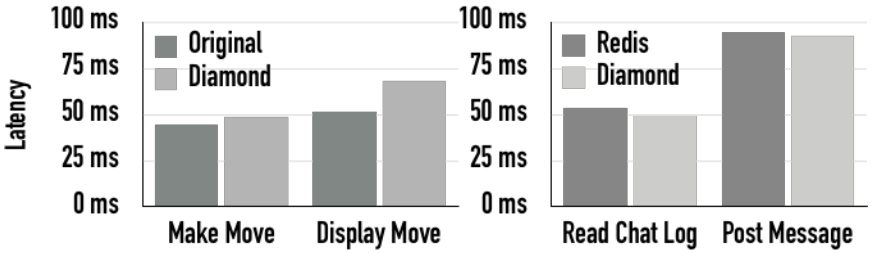


Figure 3.11: *End-user operation latency for PyScrabble and Chat Room on Diamond and non-Diamond implementations.*

the success of their commercial counterparts. Combined, they comprise a mobile back-end market of $1.32 billion dollars [136].

However, these products do not meet the requirements of reactive applications, still requiring programmers to address failures and race conditions. Meteor [141] lets client-side code directly access the database interface. However, because it uses existing databases (MongoDB [146], and most recently, Postgres [168]) that do not support distributed transactions and offer weak consistency guarantees by default, programmers must still reason about race conditions and consistency bugs. Parse [160] and Firebase [69] similarly enable clients to read, write, and subscribe to objects that are automatically synchronized across mobile devices; however, these systems offer no concurrency control or transactions. As demonstrated by these Stack Overflow questions [149, 137], programmers find this to be a significant issue with these systems. Diamond addresses this clear developer need by providing ACID+R guarantees for reactive applications.

There has been significant work in wide-area storage systems for distributed and mobile applications, including numerous traditional instantiations [199, 111, 150] as well as more recent work [53, 27, 195, 161, 196]. Many mobile applications today use commercial storage services such as Dropbox and others [62, 61, 99], while users can also employ revision-based storage (e.g., git [74]). Applications often combine distributed storage with notifications [4, 15]. As discussed, these systems help with data management, but none offers a complete solution.

Diamond shares a data-type-based storage model with data structure stores [173, 174]. Document

stores (e.g., MongoDB [146]) support application objects; this prevents them from leveraging semantics for better performance. These datastores, along with more traditional key-value and relational storage systems [40, 22, 117, 197], were not designed for wide-area use although they could support reactive applications with additional work.

Reactive transactions in Diamond are similar to database triggers [126], events [39], and materialized views [32]. They differ from these mechanisms because they modify local application state and execute application code rather than database queries that update storage state. Diamond's design draws on Thialfi [4]; however, Thialfi cannot efficiently support data push notifications without insight into the application's access patterns.

DOCC is similar to Herlihy [90, 89] and Weihl's [215] work on concurrency control for abstract data types. However, Diamond applies their techniques to CRDTs [182] over a range of isolation levels in the wide area. DOCC is also related to MDCC [113] and Egalitarian Paxos [147]; however, DOCC uses commutativity for transactional concurrency control rather than Paxos ordering and supports more data types. DOCC extends recent work on software transactional objects [91] for single-node databases to the wide area; integrating the two would let programmers implement custom data types in Diamond.

Diamond does not strive to support a fully reactive, data-flow-based programming model, like functional reactive or constraint-based programming [213, 16]; however, reactive transactions are based on the idea of change propagation. Recent interest in reactive programming for web client UIs has resulted in Facebook's popular React.js [170], the ReactiveX projects [171], and Google's Agera[75]. DREAM [135], a recently proposed, distributed reactive platform, lacks transactional guarantees. Sapphire [221], another recent programming platform for mobile/could applications, does not support reactivity, distributed transactions, or general-purpose data management.

3.7 Summary

This paper described Diamond, the first data management service for wide-area reactive applications. Diamond introduced three new concepts: the `rmap` primitive, reactive transactions, and DOCC. Our evaluation demonstrated that: (1) Diamond's programming model greatly simplifies reactive applica-

tions, (2) Diamond's strong transactional guarantees eliminate data race bugs, and (3) Diamond's low performance overhead has no impact on the end-user.

4 TAPIR

Distributed storage systems help mobile/cloud applications meet many of the requirements previously covered in Section 1.3. Because mobile/cloud applications have significant amounts of concurrency, programmers need storage systems with support for distributed transactions to achieve strong consistency guarantees. Several recent systems [113, 22, 65, 45] meet this need, notably Google's Spanner system [48], which guarantees linearizable transaction ordering.[1]

For application programmers, distributed transactional storage with strong consistency comes at a price. These systems commonly use replication for fault-tolerance, and replication protocols with strong consistency, like Paxos, impose a high performance cost, while more efficient, weak consistency protocols fail to provide strong system guarantees.

Significant prior work has addressed improving the performance of transactional storage systems – including systems that optimize for read-only transactions [22, 48], more restrictive transaction models [113, 7, 50], or weaker consistency guarantees [132, 188, 21]. However, none of these systems have addressed both latency *and* throughput for general-purpose, replicated, read-write transactions with strong consistency.

In this chapter, we use a new approach to reduce the cost of replicated, read-write transactions and make transactional storage more affordable for programmers. Our key insight is that existing transactional storage systems waste work and performance by incorporating a distributed transaction protocol and a replication protocol that *both* enforce strong consistency. Instead, we show that it is possible to provide distributed transactions with better performance and the same transaction and

[1]Spanner's linearizable transaction ordering is also referred to as strict serializable isolation or external consistency.

consistency model using replication with *no consistency*.

To demonstrate our approach, we designed *TAPIR* – the Transactional Application Protocol for Inconsistent Replication. TAPIR uses a new replication technique, called *inconsistent replication* (IR), that provides fault tolerance without consistency. Rather than an ordered operation log, IR presents an *unordered operation set* to applications. Successful operations execute at a majority of the replicas and survive failures, but replicas can execute them in any order. Thus, IR needs no cross-replica coordination or designated leader for operation processing. However, unlike eventual consistency, IR allows applications to enforce higher-level invariants when needed.

Thus, despite IR's weak consistency guarantees, TAPIR provides *linearizable read-write transactions* and supports globally-consistent reads across the database at a timestamp – the same guarantees as Spanner. TAPIR efficiently leverages IR to distribute read-write transactions in a *single round-trip* and order transactions globally across partitions and replicas *with no centralized coordination*.

We implemented TAPIR in a new distributed transactional key-value store called *TAPIR-KV*, which supports linearizable transactions over a partitioned set of keys. Our experiments found that TAPIR-KV had: (1) 50% lower commit latency and (2) more than 3× better throughput compared to systems using conventional transaction protocols, including an implementation of Spanner's transaction protocol, and (3) comparable performance to MongoDB [146] and Redis [173], widely-used eventual consistency systems.

This chapter presents the following contributions to the design of distributed, replicated transaction systems:

- We define *inconsistent replication*, a new replication technique that provides fault tolerance without consistency.

- We design *TAPIR*, a new distributed transaction protocol that provides strict serializable transactions using inconsistent replication for fault tolerance.

- We build and evaluate TAPIR-KV, a key-value store that combines inconsistent replication and TAPIR to achieve high-performance transactional storage.

4.1 Background

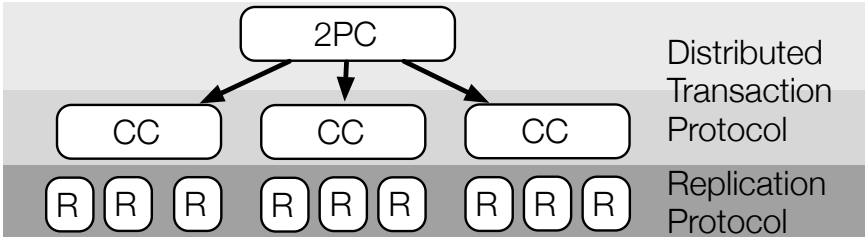


Figure 4.1: *Today's common architecture for distributed transactional storage systems.* The distributed transaction protocol consists of an atomic commitment protocol, commonly Two-Phase Commit (2PC), and a concurrency control (CC) mechanism. This runs atop a replication (R) protocol, like Paxos.

Replication protocols have become an important component in distributed storage systems. Modern storage systems commonly partition data into *shards* for scalability and then replicate each shard for fault-tolerance and availability [22, 40, 48, 134]. To support transactions with strong consistency, they must implement both a *distributed transaction protocol* – to ensure atomicity and consistency for transactions across shards – and a *replication protocol* – to ensure transactions are not lost (provided that no more than half of the replicas in each shard fail at once). As shown in Figure 4.1, these systems typically place the transaction protocol, which combines an atomic commitment protocol and a concurrency control mechanism, on top of the replication protocol (although alternative architectures have also occasionally been proposed [134]).

Distributed transaction protocols assume the availability of an *ordered, fault-tolerant log*. This ordered log abstraction is easily and efficiently implemented with a spinning disk but becomes more complicated and expensive with replication. To enforce the serial ordering of log operations, transactional storage systems must integrate a costly replication protocol with strong consistency (e.g., Paxos [119], Viewstamped Replication [156] or virtual synchrony [31]) rather than a more efficient, weak consistency protocol [116, 178].

The traditional log abstraction imposes a serious performance penalty on replicated transactional

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 transaction protocol enforces a serial ordering of transactions across shards. 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.

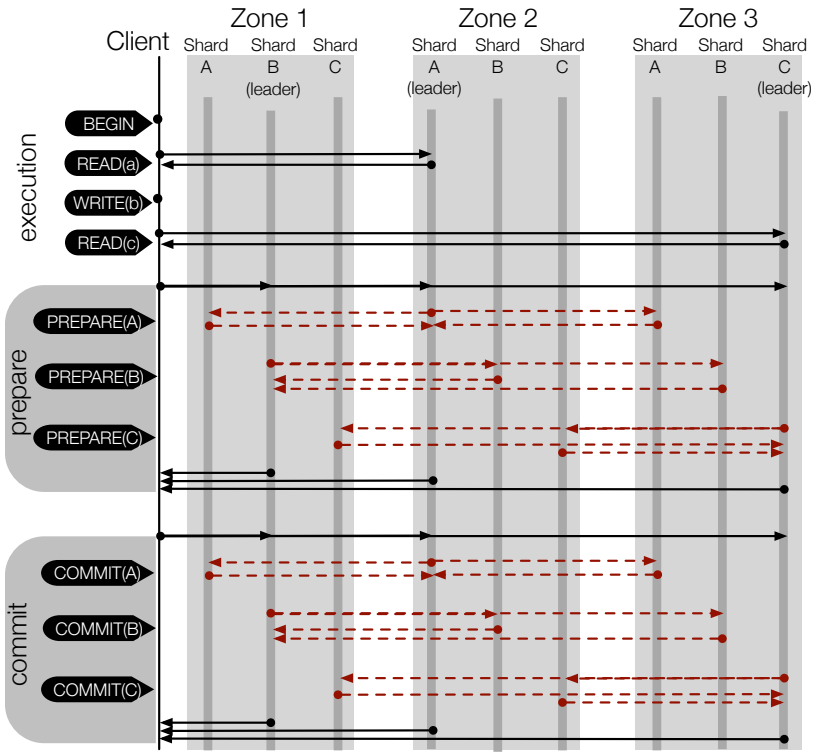


Figure 4.2: *Example read-write transaction using two-phase commit, viewstamped replication and strict two-phase locking.* Availability *zones* represent either a cluster, datacenter or geographic region. Each *shard* holds a partition of the data stored in the system and has replicas in each zone for fault tolerance. The red, dashed lines represent redundant coordination in the replication layer.

As an example, Figure 4.2 shows the redundant coordination required for a single read-write transaction in a system like Spanner. Within the transaction, `Read` operations go to the shard leaders (which may be in other datacenters), because the operations must be ordered across replicas, even though they are not replicated. To `Prepare` a transaction for commit, the transaction protocol must coordinate transaction ordering across shards, and then the replication protocol coordinates the `Prepare` operation ordering across replicas. As a result, it takes at least two round-trips to commit the transaction.

In the TAPIR and IR design, we eliminate the redundancy of strict serial ordering over the two layers and its associated performance costs. IR is the first replication protocol to provide *pure fault tolerance* without consistency. Instead of an ordered operation log, IR presents the abstraction of an *unordered operation set*. Existing transaction protocols cannot efficiently use IR, so TAPIR is the first transaction protocol designed to provide linearizable transactions on IR.

4.2 Inconsistent Replication

Inconsistent replication (IR) is an efficient replication protocol designed to be used with a higher-level protocol, like a distributed transaction protocol. IR provides fault-tolerance without enforcing any consistency guarantees of its own. Instead, it allows the higher-level protocol, which we refer to as the *application protocol*, to decide the outcome of conflicting operations and recover those decisions through IR's fault-tolerant, unordered operation set.

4.2.1 IR Overview

Application protocols invoke operations through IR in one of two modes:

- **inconsistent** – operations can execute in any order. Successful operations persist across failures.
- **consensus** – operations execute in any order, but return a single *consensus result*. Successful operations and their consensus results persist across failures.

inconsistent operations are similar to operations in weak consistency replication protocols: they

can execute in different orders at each replica, and the application protocol must resolve conflicts afterwards. In contrast, **consensus** operations allow the application protocol to *decide* the outcome of conflicts (by executing a *decide* function specified by the application protocol) and recover that decision afterwards by ensuring that the chosen result persists across failures as the consensus result. In this way, **consensus** operations can serve as the basic building block for the higher-level guarantees of application protocols. For example, a distributed transaction protocol can decide which of two conflicting transactions will commit, and IR will ensure that decision persists across failures.

IR Application Protocol Interface

Figure 4.3 summarizes the IR interfaces at clients and replicas. Application protocols invoke operations through a client-side IR library using `InvokeInconsistent` and `InvokeConsensus`, and then IR runs operations using the `ExecInconsistent` and `ExecConsensus` upcalls at the replicas.

If replicas return conflicting/non-matching results for a **consensus** operation, IR allows the application protocol to decide the operation's outcome by invoking the *decide* function – passed in by the application protocol to `InvokeConsensus` – in the client-side library. The *decide* function takes the list of returned results (the candidate results) and returns a single result, which IR ensures will persist as the *consensus result*. The application protocol can later recover the consensus result to find out its decision to conflicting operations.

Some replicas may miss operations or need to reconcile their state if the consensus result chosen by the application protocol does not match their result. To ensure that IR replicas eventually converge, they periodically *synchronize*. Similar to eventual consistency, IR relies on the application protocol to reconcile inconsistent replicas. On synchronization, a single IR node first upcalls into the application protocol with `Merge`, which takes records from inconsistent replicas and merges them into a *master record* of successful operations and consensus results. Then, IR upcalls into the application protocol with `Sync` at each replica. `Sync` takes the *master record* and reconciles application protocol state to make the replica consistent with the chosen consensus results.

Client Interface

`InvokeInconsistent`(op)

`InvokeConsensus`(op), *decide*(*results*)) → *result*

Replica Upcalls

`ExecInconsistent`(op) `ExecConsensus`(op) → *result*

`Sync`(R) `Merge`(d, u) → *record*

Client State

- *client id* - unique identifier for the client
- *operation counter* - # of sent operations

Replica State

- *state* - current replica state; either NORMAL or VIEW-CHANGING
- *record* - unordered set of operations and consensus results

Figure 4.3: *Summary of IR interfaces and client/replica state.*

IR Guarantees

We define a *successful operation* to be one that returns to the application protocol. The *operation set* of any IR group includes all successful operations. We define an operation X as being *visible* to an operation Y if one of the replicas executing Y has previously executed X. IR ensures the following properties for the operation set:

P1. [Fault tolerance] At any time, every operation in the operation set is in the record of at least one replica in any quorum of $f + 1$ non-failed replicas.

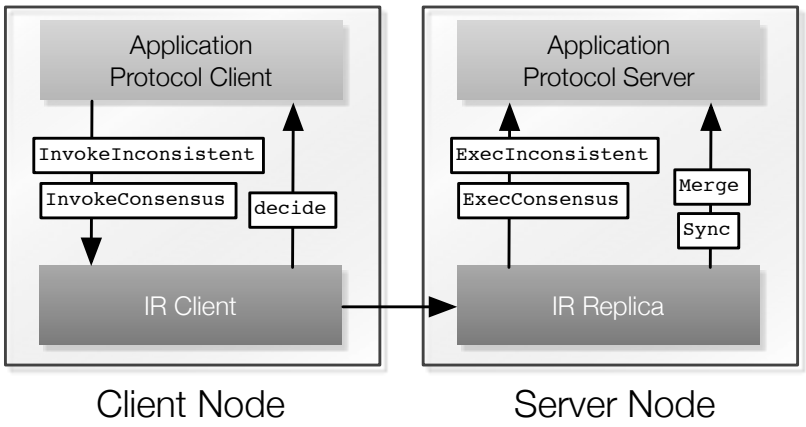


Figure 4.4: *IR Call Flow.*

P2. **[Visibility]** For any two operations in the operation set, at least one is visible to the other.

P3. **[Consensus results]** At any time, the result returned by a successful **consensus** operation is in the record of at least one replica in any quorum. The only exception is if the consensus result has been explicitly modified by the application protocol through `Merge`, after which the outcome of `Merge` will be recorded instead.

IR ensures guarantees are met for up to f simultaneous failures out of $2f + 1$ replicas[2] and any number of client failures. Replicas must fail by crashing, without Byzantine behavior. We assume an asynchronous network where messages can be lost or delivered out of order. IR *does not* require synchronous disk writes, ensuring guarantees are maintained even if clients or replicas lose state on failure. IR makes progress (operations will eventually become successful) provided that messages that are repeatedly resent are eventually delivered before the recipients time out.

[2] Using more than $2f + 1$ replicas for f failures is possible but illogical because it requires a larger quorum size with no additional benefit.

Application Protocol Example: Fault-Tolerant Lock Server

As an example, we show how to build a simple lock server using IR. The lock server's guarantee is mutual exclusion: a lock cannot be held by two clients at once. We replicate `Lock` as a **consensus** operation and `Unlock` as an **inconsistent** operation. A client application acquires the lock only if `Lock` successfully returns OK as a consensus result.

Since operations can run in any order at the replicas, clients use unique ids (e.g., a tuple of client id and a sequence number) to identify corresponding `Lock` and `Unlock` operations and only call `Unlock` if `Lock` first succeeds. Replicas will therefore be able to later match up `Lock` and `Unlock` operations, regardless of order, and determine the lock's status.

Note that **inconsistent** operations *are not commutative* because they can have side-effects that affect the outcome of **consensus** operations. If an `Unlock` and `Lock` execute in different orders at different replicas, some replicas might have the lock free, while others might not. If replicas return different results to `Lock`, IR invokes the lock server's decide function, which returns OK if $f + 1$ replicas returned OK and NO otherwise. IR only invokes `Merge` and `Sync` on recovery, so we defer their discussion until Section 4.2.2.

IR's guarantees ensure correctness for our lock server. P1 ensures that held locks are persistent: a `Lock` operation persists at one or more replicas in any quorum. P2 ensures mutual exclusion: for any two conflicting `Lock` operations, one is visible to the other in any quorum; therefore, IR will never receive $f + 1$ matching OK results, precluding the *decide* function from returning OK. P3 ensures that once the client application receives OK from a `Lock`, the result will not change. The lock server's `Merge` function will not change it, as we will show later, and IR ensures that the OK will persist in the record of at least one replica out of any quorum.

4.2.2 IR Protocol

Figure 4.3 shows the IR state at the clients and replicas. Each IR client keeps an *operation counter*, which, combined with the *client id*, uniquely identifies operations. Each replica keeps an unordered *record* of executed operations and results for **consensus** operations. Replicas add **inconsistent** oper-

ations to their record as TENTATIVE and then mark them as FINALIZED once they execute. **consensus** operations are first marked TENTATIVE with the result of locally executing the operation, then FINALIZED once the record has the consensus result.

IR uses four sub-protocols – *operation processing*, *replica recovery/synchronization*, *client recovery*, and *group membership change*. We discuss the first three here; the last is identical to that of Viewstamped Replication [128].

Operation Processing

We begin by describing IR's normal-case **inconsistent** operation processing protocol without failures:

1. The client sends $\langle \text{PROPOSE}, \ id, \ op \rangle$ to all replicas, where id is the operation id and op is the operation.

2. Each replica writes id and op to its record as TENTATIVE, then responds to the client with $\langle \text{REPLY}, \ id \rangle$.

3. Once the client receives $f + 1$ responses from replicas (retrying if necessary), it returns to the application protocol and asynchronously sends $\langle \text{FINALIZE}, \ id \rangle$ to all replicas. (FINALIZE can also be piggy-backed on the client's next message.)

4. On FINALIZE, replicas upcall into the application protocol with `ExecInconsistent`(op) and mark op as FINALIZED.

Due to the lack of consistency, IR can successfully complete an **inconsistent** operation with a *single round-trip* to $f + 1$ replicas and no coordination across replicas. If the IR client does not receive a response to its PREPARE from $f + 1$ replicas, it will retry until it does.

Next, we describe the normal-case **consensus** operation processing protocol, which has both a *fast path* and a *slow path*. IR uses the fast path when it can achieve a *fast quorum* of $\lceil \frac{3}{2} f \rceil + 1$ replicas that *return matching results* to the operation. Similar to Fast Paxos and Speculative Paxos [167], IR requires a fast quorum to ensure that a majority of the replicas in any quorum agrees to the consensus

result. This quorum size is necessary to execute operations in a single round trip when using a replica group of size $2f+1$ [121]; an alternative would be to use quorums of size $2f+1$ in a system with $3f+1$ replicas.

When IR cannot achieve a fast quorum, either because replicas did not return enough matching results (e.g., if there are conflicting concurrent operations) or because not enough replicas responded (e.g., if more than $\frac{f}{2}$ are down), then it must take the slow path. We describe both below:

1. The client sends $\langle$PROPOSE, id, $op\rangle$ to all replicas.

2. Each replica calls into the application protocol with `ExecConsensus`(op) and writes id, op, and *result* to its record as TENTATIVE. The replica responds to the client with $\langle$REPLY, id, $result\rangle$.

3. If the client receives at least $\lceil \frac{3}{2}f \rceil + 1$ matching *result*s (within a timeout), then it takes the *fast path*: the client returns *result* to the application protocol and asynchronously sends $\langle$FINALIZE, id, $result\rangle$ to all replicas.

4. Otherwise, the client takes the *slow path*: once it receives $f+1$ responses (retrying if necessary), then it sends $\langle$FINALIZE, id, $result\rangle$ to all replicas, where *result* is obtained from executing the *decide* function.

5. On receiving FINALIZE, each replica marks the operation as FINALIZED, updating its record if the received *result* is different, and sends $\langle$CONFIRM, $id\rangle$ to the client.

6. On the slow path, the client returns *result* to the application protocol once it has received $f+1$ CONFIRM responses.

The fast path for **consensus** operations takes a single round trip to $\lceil \frac{3}{2}f \rceil + 1$ replicas, while the slow path requires two round-trips to at least $f+1$ replicas. Note that IR replicas can execute operations in different orders and *still* return matching responses, so IR can use the fast path without a strict serial ordering of operations across replicas. IR can also run the fast path and slow path in parallel as an optimization.

Replica Recovery and Synchronization

IR uses a single protocol for recovering failed replicas and running periodic synchronizations. On recovery, we must ensure that the failed replica applies all operations it may have lost or missed in the operation set, so we use the same protocol to periodically bring all replicas up-to-date.

To handle recovery and synchronization, we introduce *view changes* into the IR protocol, similar to Viewstamped Replication (VR) [156]. These maintain IR's correctness guarantees across failures. Each IR view change is run by a leader; leaders coordinate only view changes, *not* operation processing. During a view change, the leader has just one task: to make at least $f + 1$ replicas up-to-date (i.e., they have applied all operations in the operation set) and consistent with each other (i.e., they have applied the same consensus results). IR view changes require a leader because polling inconsistent replicas can lead to conflicting sets of operations and consensus results. Thus, the leader must decide on a *master record* that replicas can then use to synchronize with each other.

To support view changes, each IR replica maintains a current *view*, which consists of the identity of the leader, a list of the replicas in the group, and a (monotonically increasing) *view number* uniquely identifying the view. Each IR replica can be in one of the three states: NORMAL, VIEW-CHANGING or RECOVERING. Replicas process operations only in the NORMAL state. We make four additions to IR's operation processing protocol:

1. IR replicas send their current view number in every response to clients. For an operation to be considered successful, the IR client must receive responses with matching view numbers. For **consensus** operations, the view numbers in REPLY and CONFIRM must match as well. If a client receives responses with different view numbers, it notifies the replicas in the older view.

2. On receiving a message with a view number that is higher than its current view, a replica moves to the VIEW-CHANGING state and requests the master record from any replica in the higher view. It replaces its own record with the master record and upcalls into the application protocol with `Sync` before returning to NORMAL state.

3. On PROPOSE, each replica first checks whether the operation was already FINALIZED by a view

change. If so, the replica responds with $\langle$REPLY, id, FINALIZED, v, $[result]\rangle$, where v is the replica's current view number and *result* is the consensus result for **consensus** operations.

4. If the client receives REPLY with a FINALIZED status for **consensus** operations, it sends $\langle$FINALIZE, id, $result\rangle$ with the received *result* and waits until it receives $f + 1$ CONFIRM responses in the same view before returning *result* to the application protocol.

IR's view change protocol is similar to VR's. Each view change is finalized by a leader (the leader is unique per view and deterministically chosen). However, in IR the leader *merges* records, rather than taking the longest log from the latest view. Records must be merged because, without the ordering requirement, any single record could be incomplete. While in VR the leader is used to process operations as well, in contrast, IR uses the leader only for performing view changes. On recovery, the IR replica performs a view change to make sure it either receives all operations it might have sent a reply for, or cause a retry for all the on-going operations it might have sent a reply for. This contrast with VR's recovery protocol where the recovering replica did not need to perform a view change because it could interrogate a sufficiently up-to-date leader for on-going operations it might have participated in. The full view change protocol follows:

1. A replica that notices the need for a view change advances its view-number and sets its status to either VIEW-CHANGING or RECOVERING – if the replica just started a recovery. A replica notices the need for a view change either based on its own timer (the current view change was not finalized in time by its leader), or because it is a recovering replica, or because it received a DO-VIEW-CHANGE message for a view with a larger number than its own current view-number.

2. If the replica is not a recovering replica, it first saves to disk the updated view number. The replica then sends a $\langle$DO-VIEW-CHANGE, rec, v, $v'\rangle$ message to the new leader, *except when the sending replica is a recovering replica*, and the same message, without the rec field, to the other replicas. Here v identifies the new view, v' is the latest view in which the replica's status was NORMAL, and rec is its unordered record of executed operations. The new view, v, sent in a DO-VIEW-CHANGE message is either the incremented view-number of the replica if the replica

is non-faulty, or the view number received in a DO-VIEW-CHANGE message, or the number a recovering replica read from its disc.

3. Once the non-faulty leader receives f records from f non-recovering replicas, it uses a merge function, shown in Figure 4.5, to join them into a master record R.The merge function uses v''s (i.e. latest view number replicas were in NORMAL state) when computing RECORD-WITH-MAX-VIEW.

4. The leader updates its view number to v_{new}, where v_{new} is the view number from the received messages, and its status to NORMAL. It then informs the other replicas of the completion of the view change by sending a $\langle$START-VIEW, v_{new}, $R\rangle$, where R is the master record.

5. Any replica that receives START-VIEW: if v_{new} is smaller than the replica's current view number, the replica stays in its current state. Otherwise, if v_{new} is higher than or equal to its current view number, the replica replaces its own record with R and upcalls into the application protocol with `Sync` as well.

6. Once the replica finishes, it updates its current view number, v, saves $v + 1$ to disc (signifying that it will never recover in a view lower or equal than v), and enters the NORMAL state.

Once the leader has received $f + 1$ records, it merges the records from replicas in the latest view into a master record, R, using IR-MERGE-RECORDS($records$) (see Figure 4.5), where $records$ is the set of received records from replicas in the highest view. IR-MERGE-RECORDS starts by adding all **inconsistent** operations and **consensus** operations marked FINALIZED to R and calling `Sync` into the application protocol. These operations must persist in the next view, so we first apply them to the leader, ensuring that they are visible to any operations for which the leader will decide consensus results next in `Merge`. As an example, `Sync` for the lock server matches up all corresponding `Lock` and `Unlock` by id; if there are unmatched `Lock`s, it sets *locked* = TRUE; otherwise, *locked* = FALSE.

IR asks the application protocol to decide the consensus result for the remaining TENTATIVE **consensus** operations, which either: (1) have a matching result, which we define as the *majority result*,

```
IR-MERGE-RECORDS(records)
 1  R, d, u = ∅
 2  for ∀op ∈ records
 3      if op.type == inconsistent
 4          R = R ∪ op
 5      elseif op.type == consensus and op.status == FINALIZED
 6          R = R ∪ op
 7      elseif op.type == consensus and op.status == TENTATIVE
 8          if op.result in more than f/2 + 1 records
 9              d = d ∪ op
10          else
11              u = u ∪ op
12  Sync(R)
13  return R ∪ Merge(d, u)
```

Figure 4.5: *Merge function for the master record.* We merge all records from replicas in the latest view, which is always a strict super set of the records from replicas in lower views.

in at least $\lceil \frac{f}{2} \rceil + 1$ records or (2) do not. IR places these operations in d and u, respectively, and calls $\texttt{Merge}(d, u)$ into the application protocol, which must return a consensus result for every operation in d and u.

IR must rely on the application protocol to decide consensus results for several reasons. For operations in d, IR cannot tell whether the operation succeeded with the majority result on the fast path, or whether it took the slow path and the application protocol *decide*'d a different result that was later lost. In some cases, it is not safe for IR to keep the majority result because it would violate application protocol invariants. For example, in the lock server, OK could be the majority result if only $\lceil \frac{f}{2} \rceil + 1$ replicas replied OK, but the other replicas might have accepted a conflicting lock request.

However, it is also possible that the other replicas *did* respond OK, in which case OK would have been a successful response on the fast-path.

The need to resolve this ambiguity is the reason for the caveat in IR's consensus property (P3) that consensus results can be changed in `Merge`. Fortunately, the application protocol can ensure that successful consensus results *do not change* in `Merge`, simply by maintaining the majority results in d on `Merge` *unless they violate invariants*. The merge function for the lock server, therefore, does not change a majority response of OK, *unless* another client holds the lock. In that case, the operation in d could not have returned a successful consensus result to the client (either through the fast or the slow path), so it is safe to change its result.

For operations in u, IR needs to invoke *decide* but cannot without at least $f + 1$ results, so uses `Merge` instead. The application protocol can decide consensus results in `Merge` without $f + 1$ replica results and still preserve IR's visibility property because IR has already applied all of the operations in R and d, which are the only operations definitely in the operation set, at this point.

The leader adds all operations returned from `Merge` and their consensus results to R, then sends R to the other replicas, which call `Sync`(R) into the application protocol and *replace their own records with R*. The view change is complete after at least $f + 1$ replicas have exchanged and merged records and SYNC'd with the master record. A replica can only process requests in the new view (in the NORMAL state) *after* it completes the view change protocol. At this point, any recovering replicas can also be considered recovered. If the leader of the view change does not finish the view change by some timeout, the group will elect a new leader to complete the protocol by starting a new view change with a larger view number.

Client Recovery

We assume that clients can lose some or all of their state on failure. On recovery, a client must ensure that: (1) it recovers its latest operation counter, and (2) any operations that it started but did not finish are FINALIZED. To do so, the recovering client requests the *id* for its latest operation from a majority of the replicas. This poll gets the client the largest *id* that the group has seen from it, so the client takes the largest returned *id* and increments it to use as its new operation counter.

A view change finalizes all TENTATIVE operation on the next synchronization, so the client does not need to finish previously started operations and IR does not have to worry about clients failing to recover after failure.

4.2.3 Correctness

For correctness, we must show that IR provides the following properties for operations in the *operation set*:

P1. [**Fault tolerance**] At any time, every operation in the operation set is in the record of at least one replica in any quorum of $f + 1$ non-failed replicas.

P2. [**Visibility**] For any two **consensus** operations in the operation set, at least one is visible to the other.

P3. [**Consensus results**] At any time, every successful consensus result is in the record of at least one replica in any quorum. Again, the only exception being that the application protocol modified the result through `Merge`.

P4. [**Eventual Consistency**] Given a sufficiently long period of synchrony, any operation in the operation set (and its consensus result, if applicable) will eventually have executed or `Synced` at every non-faulty replica.

In Appendix B, we give a TLA+ specification, which we have model-checked. In addition, we have added an *eventual consistency* property, which is not necessary for correctness, but is useful for application protocols. As this is a liveness property, it holds only during periods of synchrony, when messages that are repeatedly resent are eventually delivered before the recipient times out [70].

We begin our proof of correctness by defining the following terms:

D1. An operation is *applied* at a replica if that replica has executed (through `ExecInconsistent` or `ExecConsensus`) or synchronized (through `Sync`) the operation.

D2. An operation X is *visible* to a **consensus** operation Y if one of the replicas providing candidate results for Y has previously applied X.

D3. The *persistent operation set* is the set of operations applied at at least one replica in any quorum of $f + 1$ non-failed replicas.

We first prove a number of invariants about the persistent operation set. Given these invariants, we can show that the IR properties hold.

I1. *The size of persistent operation set is monotomically increasing.*

I1 holds at every replica during normal operation because replicas never roll back executed operations. I1 also hold across view changes. The leader merges all operations from the records of $f + 1$ non-faulty replicas into the master record, so by quorum intersection, the master record contains every operation in the persistent operation set. Then, at least $f + 1$ non-faulty replicas replace their record with the master record and applies the master record (through `Sync`), so any persistent operations before the view change will continue to persist after the view change.

I2. *All operations in the persistent operation set are visible to any* ***consensus*** *operation added to the set.*

consensus operations are added to the persistent set by either: (1) executing at at least a quorum of $f + 1$ replicas or (2) being merged by the leader into the master record. In case 1, by definition, every operation already in the persistent operation set must be applied at at least 1 replica out of the quorum and will be visible to the added **consensus** operation. In case 2, the leader applies all operations in the persistent operation set (through `Sync`) before running `Merge`, ensuring that every operation already in the persistent operation set is visible to operations added to the persistent operation set through `Merge`.

I3. *The result of any* ***consensus*** *operation in the persistent operation set is* either *the successful consensus result or the* `Merge` *result.*

The result of any **consensus** operations in the persistent set is either: (1) a matching result from executing the operation (through `ExecConsensus`) at a fast quorum of $\lceil \frac{3}{2}f \rceil + 1$, (2) a result from executing the application protocol-specific *decide* function in the client-side library, or (3) a result from executing `Merge` at the leader during a view change. In case 1, the matching result will be both the result in the persistent operation set and the successful consensus result. The same holds for the result returned from *decide* in case 2. During a view change, the leader may get an operation that has already fulfilled either case 1 or case 2, and change the result in `Merge`. The result from `Merge` will be in the record and applied to at least $f + 1$ replicas. Thus, either the successful consensus result or, if the application protocol changed the result in `Merge`, the `Merge` result, will continue to persist in the persistent operation set.

I4. *All operations and consensus results in the persistent operation set in all previous view must be applied at a replica before it executes any operations in the new view.*

IR clients require that all responses come from replicas in the same view. Thus, any replica in a view lower than the majority, (i.e., some view v, where at least $f + 1$ replicas are in a view V, where $V > v$), must join the higher view before participating in processing operations. In order to join the higher view, the replica in the lower view must find a replica in the higher view, get the master record and `Sync` with the master record from the higher view. As the master record contains all operations in the persistent operation set, the replica will apply all operations from the persistent operation set before processing operations in the new view.

Given these four invariants for the persistent operation set, we can show that the four properties of IR hold. Any operation in the operation set must have executed at and received responses from $f + 1$ of $2f + 1$ replicas, so by quorum intersection, all operations in the operation set must be in the persistent operation set. Thus, I1 directly implies P1, as any operation in the persistent operation set will continue to be in the set. I1 and I2 imply P2 because, for any **consensus** operation X, all operations added to the persistent operation set before X are visible to X and X will be visible to all operations added to the persistent operation set after it. I1 and I3 implies P3 because either the successful consensus result will remain in the persistent operation set or the `Merge` result will. I4

implies P4 because, if all replicas are non-faulty for long enough, they will eventually all attempt to participate in processing operations, which will cause them to `Sync` all operations in the persistent operation set.

4.3 Building Atop IR

IR obtains performance benefits because it offers weak consistency guarantees and relies on application protocols to resolve inconsistencies, similar to eventual consistency protocols such as Dynamo [56] and Bayou [199]. However, unlike eventual consistency systems, which expect applications to resolve conflicts *after they happen*, IR allows application protocols to *prevent conflicts before they happen*. Using **consensus** operations, application protocols can enforce higher-level guarantees (e.g., TAPIR's linearizable transaction ordering) across replicas despite IR's weak consistency.

However, building strong guarantees on IR requires careful application protocol design. IR cannot support certain application protocol invariants. Moreover, if misapplied, IR can even provide applications with *worse* performance than a strongly consistent replication protocol. In this section, we discuss the properties that application protocols need to have to correctly and efficiently enforce higher-level guarantees with IR and TAPIR's techniques for efficiently providing linearizable transactions.

4.3.1 **IR Application Protocol Requirement:** *Invariant checks must be performed pairwise.*

Application protocols can enforce certain types of invariants with IR, but not others. IR guarantees that in any pair of **consensus** operations, at least one will be visible to the other (P2). Thus, IR readily supports invariants that can be safely checked by examining *pairs* of operations for conflicts. For example, our lock server example can enforce mutual exclusion. However, application protocols cannot check invariants that require the entire history, because each IR replica may have an incomplete history of operations. For example, tracking bank account balances and allowing withdrawals only if the balance remains positive is problematic because the invariant check must consider the entire history of deposits and withdrawals.

Despite this seemingly restrictive limitation, application protocols can still use IR to enforce useful invariants, including lock-based concurrency control, like Strict Two-Phase Locking (S2PL).

As a result, distributed transaction protocols like Spanner [48] or Replicated Commit [134] would work with IR. IR can also support optimistic concurrency control (OCC) [115] because OCC checks are pairwise as well: each committing transaction is checked against every previously committed transaction, so **consensus** operations suffice to ensure that *at least one replica sees any conflicting transaction* and aborts the transaction being checked.

4.3.2 **IR Application Protocol Requirement:** *Application protocols must be able to change* ***consensus*** *operation results.*

Inconsistent replicas could execute **consensus** operations with one result and later find the group agreed to a different consensus result. For example, if the group in our lock server agrees to reject a `Lock` operation that one replica accepted, the replica must later free the lock, and vice versa. As noted above, the group as a whole continues to enforce mutual exclusion, so these temporary inconsistencies are tolerable and are always resolved by the end of synchronization.

In TAPIR, we take the same approach with distributed transaction protocols. 2PC-based protocols are always prepared to abort transactions, so they can easily accommodate a `Prepare` result changing from PREPARE-OK to ABORT. If ABORT changes to PREPARE-OK, it might temporarily cause a conflict at the replica, which can be correctly resolved because the group as a whole could not have agreed to PREPARE-OK for two conflicting transactions.

Changing `Prepare` results does sometimes cause unnecessary aborts. To reduce these, TAPIR introduces two `Prepare` results in addition to PREPARE-OK and ABORT: ABSTAIN and RETRY. ABSTAIN helps TAPIR distinguish between conflicts with *committed* transactions, which will not abort, and conflicts with *prepared* transactions, which may later abort. Replicas return RETRY if the transaction has a chance of committing later. The client can retry the `Prepare` *without* re-executing the transaction.

4.3.3 **IR Performance Principle:** *Application protocols should not expect operations to execute in the same order.*

To efficiently achieve agreement on consensus results, application protocols should not rely on operation ordering for application ordering. For example, many transaction protocols [86, 22, 113] use Paxos operation ordering to determine transaction ordering. They would perform worse with IR because replicas are unlikely to agree on which transaction should be next in the transaction ordering.

In TAPIR, we use *optimistic timestamp ordering* to ensure that replicas agree on a single transaction ordering despite executing operations in different orders. Like Spanner [48], every committed transaction has a timestamp, and committed transaction timestamps reflect a linearizable ordering. However, TAPIR clients, not servers, propose a timestamp for their transaction; thus, if TAPIR replicas agree to commit a transaction, they have all agreed to the same transaction ordering.

TAPIR replicas use these timestamps to order their transaction logs and multi-versioned stores. Therefore, replicas can execute `Commit` in different orders but still converge to the same application state. TAPIR leverages loosely synchronized clocks at the clients for picking transaction timestamps, which improves performance but is not necessary for correctness.

4.3.4 **IR Performance Principle:** *Application protocols should use cheaper* ***inconsistent*** *operations whenever possible rather than* ***consensus*** *operations.*

By concentrating invariant checks in a few operations, application protocols can reduce **consensus** operations and improve their performance. For example, in a transaction protocol, any operation that decides transaction ordering must be a **consensus** operation to ensure that replicas agree to the same transaction ordering. For locking-based transaction protocols, this is any operation that acquires a lock. Thus, every `Read` and `Write` must be replicated as a **consensus** operation.

TAPIR improves on this by using optimistic transaction ordering and OCC, which reduces **consensus** operations by concentrating all ordering decisions into a single set of validation checks at the proposed transaction timestamp. These checks execute in `Prepare`, which is TAPIR's only

consensus operation. `Commit` and `Abort` are **inconsistent** operations, while `Read` and `Write` are not replicated.

4.4 TAPIR

This section details TAPIR – the Transactional Application Protocol for Inconsistent Replication. As noted, TAPIR is designed to efficiently leverage IR's weak guarantees to provide high-performance linearizable transactions. Using IR, TAPIR can order a transaction in a *single round-trip* to all replicas in all participant shards without *any centralized coordination*.

TAPIR is designed to be layered atop IR in a replicated, transactional storage system. Together, TAPIR and IR eliminate the redundancy in the replicated transactional system, as shown in Figure 4.2. As a comparison, Figure 4.6 shows the coordination required for the same read-write transaction in TAPIR with the following benefits: (1) TAPIR does not have any leaders or centralized coordination, (2) TAPIR `Reads` always go to the closest replica, and (3) TAPIR `Commit` takes a single round-trip to the participants in the common case.

4.4.1 Overview

TAPIR is designed to provide distributed transactions for a scalable storage architecture. This architecture partitions data into shards and replicates each shard across a set of storage servers for availability and fault tolerance. Clients are front-end application servers, located in the same or another datacenter as the storage servers, not end-hosts or user machines. They have access to a directory of storage servers using a service like Chubby [35] or ZooKeeper [98] and directly map data to servers using a technique like consistent hashing [109].

TAPIR provides a general storage and transaction interface for applications via a client-side library. Note that TAPIR is the application protocol for IR; applications using TAPIR do not interact with IR directly.

A TAPIR application `Begins` a transaction, then executes `Reads` and `Writes` during the transaction's *execution period*. During this period, the application can `Abort` the transaction. Once it finishes execution, the application `Commits` the transaction. Once the application calls `Commit`, it can no

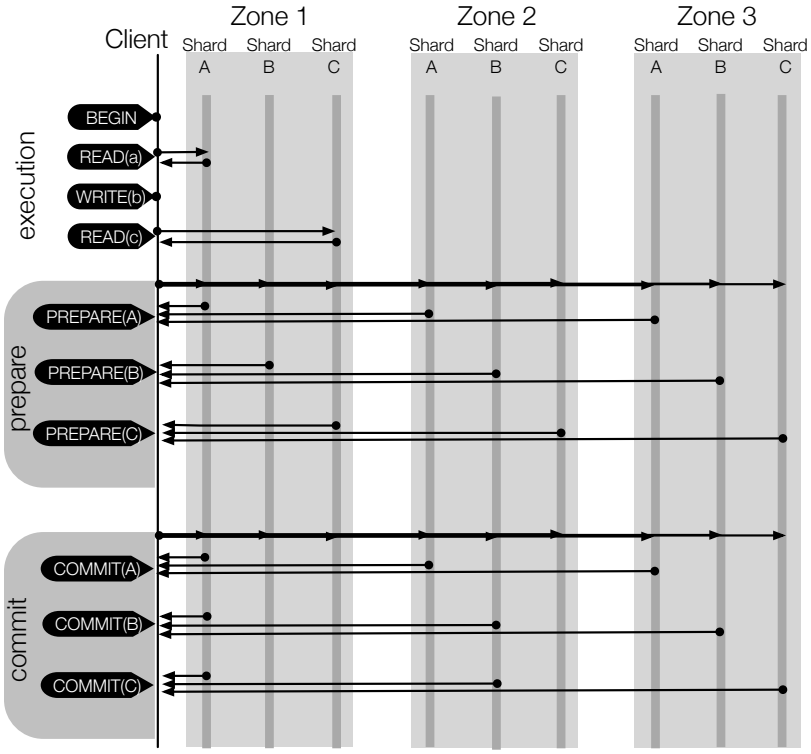


Figure 4.6: *Example read-write transaction in TAPIR.* TAPIR executes the same transaction pictured in Figure 4.2 with less redundant coordination. `Reads` go to the closest replica and `Prepare` takes a single round-trip to all replicas in all shards.

longer abort the transaction. The 2PC protocol will run to completion, committing or aborting the transaction based entirely on the decision of the participants. As a result, TAPIR's 2PC coordinators cannot make commit or abort decisions and do not have to be fault-tolerant. This property allows TAPIR to use clients as 2PC coordinators, as in MDCC [113], to reduce the number of round-trips to storage servers.

TAPIR provides the traditional ACID guarantees with the strictest level of isolation: strict serializability (or linearizability) of committed transactions.

4.4.2 Protocol

TAPIR provides transaction guarantees using a *transaction processing protocol*, *IR functions*, and a *coordinator recovery protocol*.

Figure 4.7 shows TAPIR's interfaces and state at clients and replicas. Replicas keep committed and aborted transactions in a *transaction log* in timestamp order; they also maintain a multi-versioned *data store*, where each version of an object is identified by the timestamp of the transaction that wrote the version. TAPIR replicas serve reads from the versioned data store and maintain the transaction log for synchronization and checkpointing. Like other 2PC-based protocols, each TAPIR replica also maintains a *prepared list* of transactions that it has agreed to commit.

Each TAPIR client supports one ongoing transaction at a time. In addition to its *client id*, the client stores the state for the ongoing *transaction*, including the *transaction id* and *read and write sets*. The transaction id must be unique, so the client uses a tuple of its client id and *transaction counter*, similar to IR. TAPIR does not require synchronous disk writes at the client or the replicas, as clients do not have to be fault-tolerant and replicas use IR.

Transaction Processing

We begin with TAPIR's protocol for executing transactions.

1. For `Write`(*key*, *object*), the client buffers *key* and *object* in the write set until commit and returns immediately.

2. For `Read`(*key*), if *key* is in the transaction's write set, the client returns *object* from the write set. If the transaction has already read *key*, it returns a cached copy. Otherwise, the client sends `Read`(*key*) to the replica.

3. On receiving `Read`, the replica returns *object* and *version*, where *object* is the latest version of *key* and *version* is the timestamp of the transaction that wrote that version.

Client Interface

`Begin()` `Read(`*key*`)→`*object* `Abort()`

`Commit()→`TRUE/FALSE `Write(`*key,object*`)`

Client State

- *client id* - unique client identifier
- *transaction* - ongoing transaction *id*, *read set*, *write set*

Replica Interface

`Read(`*key*`)→`*object,version* `Commit(`*txn,timestamp*`)`

`Abort(`*txn,timestamp*`)`

`Prepare(`*txn,timestamp*`)→`PREPARE-OK/ABSTAIN/ABORT/(RETRY, t)

Replica State

- *prepared list* - list of transactions replica is prepared to commit
- *transaction log* - log of committed and aborted transactions
- *store* - versioned data store

Figure 4.7: *Summary of TAPIR interfaces and client and replica state.*

4. On response, the client puts (*key, version*) into the transaction's read set and returns *object* to the application.

Once the application calls `Commit` or `Abort`, the execution phase finishes. To commit, the TAPIR client coordinates across all *participants* – the shards that are responsible for the keys in the read or write set – to find a single timestamp, consistent with the strict serial order of transactions, to assign the transaction's reads and writes, as follows:

```
TAPIR-EXEC-CONSENSUS(op)
 1  txn = op.args.txn
 2  timestamp = op.args.timestamp
 3  if txn.id ∈ txn-log
 4      if txn-log[txn.id].status == COMMITTED
 5          return PREPARE-OK
 6      else
 7          return ABORT
 8      elseif txn.id ∈ prepared-list
 9          return PREPARE-OK
10      else
11          return TAPIR-OCC-CHECK(txn, timestamp)
```

Figure 4.8: *TAPIR's consensus operation handler.* Since `Prepare` is TAPIR's only **consensus** operations, TAPIR-EXEC-CONSENSUS just runs TAPIR's prepare algorithm at replicas.

1. The TAPIR client selects a *proposed timestamp*. Proposed timestamps must be unique, so clients use a tuple of their local time and their *client id*.

2. The TAPIR client invokes `Prepare`(*txn*, *timestamp*) as an IR **consensus** operation, where *timestamp* is the proposed timestamp and *txn* includes the transaction id ($txn.id$) and the transaction read ($txn.read\_set$) and write sets ($txn.write\_set$). The client invokes `Prepare` on all participants through IR as a **consensus** operations.

3. Each TAPIR replica that receives `Prepare` (invoked by IR through `ExecConcensus`) first checks its transaction log for $txn.id$. If found, it returns PREPARE-OK if the transaction committed or ABORT if the transaction aborted.

4. Otherwise, the replica checks if $txn.id$ is already in its *prepared list*. If found, it returns PREPARE-OK.

5. Otherwise, the replica runs TAPIR's *OCC validation checks*, which check for conflicts with the transaction's read and write sets at *timestamp*, shown in Figure 4.9.

6. Once the TAPIR client receives results from all shards, the client sends `Commit`(*txn*, *timestamp*) if all shards replied PREPARE-OK or `Abort`(*txn*, *timestamp*) if any shards replied ABORT or ABSTAIN. If any shards replied RETRY, then the client retries with a new proposed timestamp (up to a set limit of retries).

7. On receiving a `Commit`, the TAPIR replica: (1) commits the transaction to its transaction log, (2) updates its versioned store with *w*, (3) removes the transaction from its prepared list (if it is there), and (4) responds to the client.

8. On receiving a `Abort`, the TAPIR replica: (1) logs the abort, (2) removes the transaction from its prepared list (if it is there), and (3) responds to the client.

Like other 2PC-based protocols, TAPIR can return the outcome of the transaction to the application as soon as `Prepare` returns from all shards (in Step 6) and send the `Commit` operations asynchronously. As a result, using IR, TAPIR can commit a transaction with a *single round-trip* to all replicas in all shards.

IR Support

Because TAPIR's `Prepare` is an IR **consensus** operation, TAPIR must implement a client-side $decide$ function, shown in Figure 4.10, which merges inconsistent `Prepare` results from replicas in a shard into a single result. TAPIR-DECIDE is simple: if a majority of the replicas replied PREPARE-OK, then it commits the transaction. This is safe because no conflicting transaction could also get a majority of the replicas to return PREPARE-OK.

```
TAPIR-OCC-CHECK(txn, timestamp)
 1  for ∀key, version ∈ txn.read-set
 2      if version < store[key].latest-version
 3          return ABORT
 4      elseif version < MIN(prepared-writes[key])
 5          return ABSTAIN
 6  for ∀key ∈ txn.write-set
 7      if timestamp < MAX(PREPARED-READS[key])
 8          return RETRY, MAX(PREPARED-READS[key])
 9      elseif timestamp < store[key].latestVersion
10          return RETRY, store[key].latestVersion
11  prepared-list[txn.id] = timestamp
12  return PREPARE-OK
```

Figure 4.9: *OCC validation function executed on* `Prepare`. PREPARED-READS and PREPARED-WRITES get the proposed timestamps for all transactions that the replica has prepared and read or write to *key*, respectively.

TAPIR also supports `Merge`, shown in Figure 4.11, and `Sync`, shown in Figure 4.12, at replicas. TAPIR-MERGE first removes any prepared transactions from the leader where the `Prepare` operation is TENTATIVE. This step removes any inconsistencies that the leader may have because it executed a `Prepare` differently – out-of-order or missed – by the rest of the group.

The next step checks d for any PREPARE-OK results that might have succeeded on the IR fast path and need to be preserved. If the transaction has not committed or aborted already, we re-run TAPIR-OCC-CHECK to check for conflicts with other previously prepared or committed transactions. If the transaction *conflicts*, then we know that its PREPARE-OK did not succeed at a fast quorum, so we can change it to ABORT; otherwise, for correctness, we must preserve the PREPARE-OK because TAPIR

```
TAPIR-DECIDE(results)
1  if ABORT ∈ results
2      return ABORT
3  if count(PREPARE-OK, results) ≥ f + 1
4      return PREPARE-OK
5  if count(ABSTAIN, results) ≥ f + 1
6      return ABORT
7  if RETRY ∈ results
8      return RETRY, max(results.retry-timestamp)
9  return ABORT
```

Figure 4.10: *TAPIR's decide function.* IR runs this if replicas return different results on `Prepare`.

may have moved on to the commit phase of 2PC. Further, we know that it is safe to preserve these PREPARE-OK results because, if they conflicted with another transaction, the conflicting transaction *must* have gotten its consensus result on the IR slow path, so if TAPIR-OCC-CHECK did not find a conflict, then the conflicting transaction's `Prepare` must not have succeeded.

Finally, for the operations in u, we simply decide a result for each operation and preserve it. We know that the leader is now consistent with $f + 1$ replicas, so it can make decisions on consensus result for the majority.

TAPIR's sync function, shown in Figure 4.12, runs at the other replicas to reconcile TAPIR state with the master records, correcting missed operations or consensus results where the replica did not agree with the group. It simply applies operations and consensus results to the replica's state: it logs aborts and commits, and prepares uncommitted transactions where the group responded PREPARE-OK.

```
TAPIR-MERGE(d, u)
 1  for ∀op ∈ d ∪ u
 2      txn = op.args.txn
 3      if txn.id ∈ prepared-list
 4          DELETE(prepared-list, txn.id)
 5  for op ∈ d
 6      txn = op.args.txn
 7      timestamp = op.args.timestamp
 8      if txn.id ∉ txn-log and op.result == PREPARE-OK
 9          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
10      else
11          R[op].result = op.result
12  for op ∈ u
13      txn = op.args.txn
14      timestamp = op.args.timestamp
15      R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
16  return R
```

Figure 4.11: *TAPIR's merge function.* IR runs this function at the leader on synchronization and recovery.

Coordinator Recovery

If a client fails while in the process of committing a transaction, TAPIR ensures that the transaction runs to completion (either commits or aborts). Further, the client may have returned the commit or abort to the application, so we must ensure that the client's commit decision is preserved. For this purpose, TAPIR uses the *cooperative termination protocol* defined by Bernstein [29] for coordinator

```
TAPIR-SYNC(R)
 1  for ∀op ∈ R
 2      if op ∉ r or op.result ≠ r[op].result
 3          txn = op.args.txn
 4          timestamp = op.args.timestamp
 5          if op.func == Prepare
 6              if op.result == PREPARE-OK
 7                  if txn.id ∉ prepared-list and txn.id ∉ txn-log
 8                      prepared-list[txn.id] = timestamp
 9              elseif txn.id ∈ prepared-list
10                  DELETE(prepared-list, txn.id)
11          else
12              txn-log[txn.id].txn = txn
13              txn-log[txn.id].timestamp = timestamp
14              if op.func == Commit
15                  txn-log[txn.id].status = COMMITTED
16              else
17                  txn-log[txn.id].status = ABORTED
18              if txn.id ∈ prepared-list
19                  DELETE(prepared-list, txn.id)
```

Figure 4.12: *TAPIR's function for synchronizing inconsistent replica state.* IR runs this on each replica except the leader during synchronization. *r* is the replica's local record.

recovery and used by MDCC [113]. TAPIR designates one of the participant shards as a *backup shard*, the replicas in which can serve as a backup coordinator if the client fails. As observed by MDCC,

because coordinators cannot unilaterally abort transactions (i.e., if a client receives $f + 1$ PREPARE-OK responses from each participant, it must commit the transaction), a backup coordinator can safely complete the protocol without blocking. However, we must ensure that no two coordinators for a transaction are active at the same time.

There are several ways to achieve this goal. We could log the currently active backup coordinator with a service like Chubby [35] or ZooKeeper [98]. We could give each backup coordinator a lease in turn, where the initial lease is given the the client as the default coordinator.

However, we chose to use a coordinator change protocol, similar to IRâĂŹs view change protocol. For each transaction, we designate one of the participant shards as a *backup shard*. The initial coordinator for every transaction is the client. In every subsequent view, the currently active backup coordinator is a replica from the backup shard, identified by indexing into the shard with a coordinator-view number.

For every transaction in its *prepared-list*, each TAPIR replica keeps the transaction's backup shard and a current *coordinator view*. Replicas *only* process and respond to `Prepare`, `Commit` and `Abort` operations from the active coordinator designated by the current view. Replicas also keep a *no-votelist* with transactions that the replica knows a backup coordinator may abort.

If the current coordinator is suspected to have failed, any of the participants can initiate a coordinator change. In doing so, it keeps the client or any previous backup coordinator from sending operations to the participating replicas. The new coordinator can then poll the participant using `Prepare`, and make a commit decision without interference from other coordinators. The election protocol for a new backup coordinator progresses as follows:

1. Any replica in any participant shard calls `CoordinatorChange` through IR as a **consensus** operation on the backup shard.

2. Each replica that executes `CoordinatorChange` through IR, increments and returns v, where is v is its current view number. If the replica is not already in the VIEW-CHANGING state, it sets its state to VIEW-CHANGING and stops responding to operations for the transaction.

3. The *decide* function for `CoordinatorChange` returns the biggest v returned by the replicas.

4. Once `CoordinatorChange` returns successfully, the replica sends `StartCoordinatorView`(v_{new}), where v_{new} is the returned view number from `CoordinatorChange`, as an IR **inconsistent** operation to *all* participant shards, including its own.

5. Any replica that receives `StartCoordinatorView` checks if v_{new} is higher or equal to its current view. If so, the replica updates its current view number and begins accepting `Prepare`, `Commit` and `Abort` from the active backup coordinator designated by the new view. If the replica is in the backup shard, it can set its state back to NORMAL.

6. As soon as the new backup coordinator executes `StartCoordinatorView` for the view where it is the designated backup coordinator, it begins the cooperative termination protocol.

The `Merge` function for `CoordinatorChange` preserves the consensus result if it is bigger than or equal to the current view number at the leader during synchronization. The `Sync` function for `CoordinatorChange` sets the replica state to VIEW-CHANGING if the consensus result is larger than the replica's current view number. The `Sync` function for `StartCoordinatorView` just executes the function: it updates the replica's current view number if v_{new} is greater than or equal to it and sets the state back to NORMAL if the replica is in the backup shard.

The backup coordination protocol executed by the active coordinator is similar to that described by Bernstein [29], with changes to accommodate IR and TAPIR. The most notable changes are that the backup coordinators *do not* propose timestamps. If the client successfully prepared the transaction at a timestamp t (i.e., achieved at least $f+1$ PREPARE-OK in every participant shard), then the transaction will commit at t. Otherwise, the backup coordinator will eventually abort the transaction.

Next, in Bernstein's algorithm, any single participant can abort the transaction if they have not yet voted (i.e., replied to a coordinator). However, with IR, no single replica can abort the transaction without information about the state of the other replicas in the shard. As a result, replicas return a NO-VOTE response and add the transaction to their *no-vote-list*. Once a replica adds a transaction to the NO-VOTE-LIST, it will always respond NO-VOTE to `Prepare` operations. Eventually, all replicas in

```
TAPIR-RECOVERY-DECIDE(results)
1  if ABORT ∈ results
2      return ABORT if count(NO-VOTE, results) ≥ f + 1
3      return ABORT
4  if count(PREPARE-OK, results) ≥ f + 1
5      return PREPARE-OK
6  return RETRY
```

Figure 4.13: *TAPIR's decide function for `Prepare` on coordinator recovery.* IR runs this if replicas return different results on `Prepare`. This function differs from the normal case execution decide because it is not safe to return ABORT unless it is sure the original coordinator did not receive PREPARE-OK.

the shard will either converge to a response (i.e., PREPARE-OK, ABORT) to the original coordinator's `Prepare` or to a NO-VOTE response. TAPIR's modified cooperative termination protocol proceeds as follows:

1. The backup coordinator polls the participants with `Prepare` with no proposed timestamp by invoking `Prepare` as a **consensus** operation in IR with the *decide* function outlined in Figure 4.13.

2. Any replica that receives `Prepare` with no propose timestamp, returns PREPARE-OK if it has committed or prepared the transaction, ABORT if it has received an `Abort` for the transaction or committed a conflicting transaction and NO-VOTE if it does not have the transaction in its *prepared-list* or *txn-log*. If the replica returns NO-VOTE, it adds the transaction to its *no-vote-list*.

3. The coordinator continues to send `Prepare` as an IR operation until it either receives a ABORT or PREPARE-OK from all participant shards.

4. If all participant shards return PREPARE-OK, the coordinator sends `Commit`; otherwise, it sends `Abort`.

Assuming $f+1$ replicas are up in each participant shard and shards are able to communicate, this process will eventually terminate with a backup coordinator sending `Commit` or `Abort` to all participants.

We must also incorporate the NO-VOTE into our `Merge` and `Sync` handlers for `Prepare`. We make the following changes to `Merge` for the final function shown in Figure 4.14: (lines 5-6) delete any tentative NO-VOTES from the *no-vote-list* at the leader for consistency, (lines 10-11) return NO-VOTE without running TAPIR-OCC-CHECK if the transaction is already in the *no-vote-list* because any result to the original `Prepare` could not have succeeded, (lines 18-19) do the same for operations without majority result where the original coordinator's `Prepare` definitely did not succeed. If the consensus result to the `Prepare` is NO-VOTE in `Sync`, we add transactions to the *no-vote-list* and remove it from the *prepared-list*, as shown in lines 11-12 of Figure 4.15.

4.4.3 Correctness

To prove correctness, we show that TAPIR maintains the following properties[3] given up to f failures in each replica group and any number of client failures:

- **Isolation.** There exists a global linearizable ordering of committed transactions.
- **Atomicity.** If a transaction commits at any participating shard, it commits at them all.
- **Durability.** Committed transactions stay committed, maintaining the original linearizable order.

Appendix B gives a TLA+ [118] specification for TAPIR with IR, which we have model-checked for correctness.

[3] We do not prove database consistency, as it depends on application invariants; however, strict serializability is sufficient to enforce consistency.

```
TAPIR-MERGE(d, u)
 1  for ∀op ∈ d ∪ u
 2      txn = op.args.txn
 3      if txn.id ∈ prepared-list
 4          DELETE(prepared-list, txn.id)
 5      if txn.id ∈ no-vote-list
 6          DELETE(no-vote-list, txn.id)
 7  for op ∈ d
 8      txn = op.args.txn
 9      timestamp = op.args.timestamp
10      if txn.id ∈ no-vote-list
11          R[op].result = NO-VOTE
12      elseif txn.id ∉ txn-log and op.result == PREPARE-OK
13          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
14      else
15          R[op].result = op.result
16  for op ∈ u
17      txn = op.args.txn
18      if txn.id ∈ no-vote-list
19          R[op].result = NO-VOTE
20      else
21          timestamp = op.args.timestamp
22          R[op].result = TAPIR-OCC-CHECK(txn, timestamp)
23  return R
```

Figure 4.14: *TAPIR's merge function.* IR runs this function at the leader on synchronization and recovery. This version handles NO-VOTE results.

```
TAPIR-SYNC(R)
 1  for ∀op ∈ R
 2      if op ∉ r or op.result ≠ r[op].result
 3          txn = op.args.txn
 4          timestamp = op.args.timestamp
 5          if op.func == Prepare
 6              if op.result == PREPARE-OK
 7                  if txn.id ∉ prepared-list and txn.id ∉ txn-log
 8                      prepared-list[txn.id] = timestamp
 9              elseif txn.id ∈ prepared-list
10                  DELETE(prepared-list, txn.id)
11                  if op.result == NO-VOTE and txn.id ∉ txn-log
12                      no-vote-list[txn.id] = timestamp
13          else
14              txn-log[txn.id].txn = txn
15              txn-log[txn.id].timestamp = timestamp
16              if op.func == Commit
17                  txn-log[txn.id].status = COMMITTED
18              else
19                  txn-log[txn.id].status = ABORTED
20              if txn.id ∈ prepared-list
21                  DELETE(prepared-list, txn.id)
```

Figure 4.15: *TAPIR's function for synchronizing inconsistent replica state.* This version handles NO-VOTE results.

Isolation

For correctness, we must show that any two conflicting transactions, A and B, that violate the linearizable transaction ordering cannot both commit. If A and B have a conflict, then there must be at least one common shard that is participating in both A and B. We show that, in that shard, `Prepare`(A) and `Prepare`(B) cannot both return PREPARE-OK, so one transaction must abort.

In the common shard, IR's visibility property (P2) guarantees that `Prepare`(A) must be *visible* to `Prepare`(B) (i.e., executes first at one replica out of every $f+1$ quorum) *or* `Prepare`(B) is visible to `Prepare`(A). Without loss of generality, suppose that `Prepare`(A) is visible to `Prepare`(B) and the group returns PREPARE-OK to `Prepare`(A). Any replica that executes TAPIR-OCC-CHECK for both A and B will not return PREPARE-OK for both, so at least one replica out of any $f+1$ quorum will not return PREPARE-OK to `Prepare`(B). IR will not get a fast quorum of matching PREPARE-OK results for `Prepare`(B), and TAPIR's *decide* function will not return PREPARE-OK because it will never get the $f+1$ matching PREPARE-OK results that it needs. Thus, IR will never return a consensus result of PREPARE-OK for `Prepare`(B). The same holds if `Prepare`(B) is visible to `Prepare`(A) and the group returns PREPARE-OK to `Prepare`(B). Thus, IR will never return a successful consensus result of PREPARE-OK to executing both `Prepare`(A) and `Prepare`(B) in the common participant shard and TAPIR will not be able to commit both transactions.

Further, once decided, the successful consensus results for `Prepare`(A) and `Prepare`(B) will persist in the record of at least one replica out of every quorum, unless it has been modified by the application through `Merge`. TAPIR will never change another result to a PREPARE-OK, so the shard will never respond PREPARE-OK to both transactions. IR will ensure that the successful consensus result is eventually `Sync`'d at all replicas. Once a TAPIR replica prepared a transaction, it will continue to return PREPARE-OK until it receives a `Commit` or `Abort` for the transaction. As a result, if the shard returned PREPARE-OK as a successful consensus result to `Prepare`(A), then it will never allow `Prepare`(B) to also return PREPARE-OK (unless A aborts), ensuring that B is never able to commit. The opposite also holds true.

Atomicity

If a transaction commits at any participating shard, the TAPIR client must have received a successful PREPARE-OK from every participating shard on `Prepare`. Barring failures, it will ensure that `Commit` eventually executes successfully at every participant. TAPIR replicas always execute `Commit`, even if they did not prepare the transaction, so `Commit` will eventually commit the transaction at every participant if it executes at one participant.

If the coordinator fails, then at least one replica in a participant shard will detect the failure and initiate the coordinator recovery protocol. Assuming no more than f simultaneous failures in the backup shard, the coordinator change protocol will eventually pick a new active backup coordinator from the backup shard. At this point, the participants will have stopped processing operations from the client and any previous backup coordinators.

Backup coordinators do not propose timestamps, so if any replica in a participant shard received a `Commit`, then the client's `Prepare` must have made it into the operation set of every participant shard with PREPARE-OK as the consensus result. IR's consensus result and eventual consistency properties (P_3 and P_4) ensure that the PREPARE-OK will eventually be applied at all replicas in every participant shard and TAPIR ensures that successful PREPARE-OK results are not changed in `Merge` (as shown above). Once a TAPIR replica applies PREPARE-OK, it will continue to return PREPARE-OK, so once replicas in participant groups have stopped processing operations from previous coordinators, all non-failed replicas in all shards will eventually return PREPARE-OK. As a result, the backup coordinator must eventually receive PREPARE-OK as well from all participants.

In the meantime, the backup coordinator is guaranteed to not receive an ABORT from a participant shard. A participant shard will only return an ABORT if: (1) a conflicting transaction committed, (2) a majority of the replicas return NO-VOTE because they did not have a record of the transaction, or (3) the transaction was aborted on the shard. Case (1) is not possible because the conflicting transaction could not have also received a successful consensus result of PREPARE-OK (based on our isolation proof) and IR's consensus result property (P_3) ensures that the conflicting transaction could never get a PREPARE-OK consensus result, so the conflicting transaction cannot commit. Case (2) is not possible

because the client could not have received PREPARE-OK as a consensus result if a majority of the replicas do not have the transaction in their *prepared-list* and IR's P3 and P4 ensures the transaction eventually makes its way into the *prepared-list* of every replica. Case (3) is not possible because the client could not have sent `Abort` if it got PREPARE-OK from all participant shards and no previous backup coordinator could have sent `Abort` because cases (1) and (2) will never happen. As a result, the backup coordinator will not abort the transaction.

Durability

For all committed transactions, either the client or a backup coordinator will eventually execute `Commit` successfully as an IR **inconsistent** operation. IR guarantees that the `Commit` will never be lost (P1) and every replica will eventually execute or synchronize it. On `Commit`, TAPIR replicas use the transaction timestamp included in `Commit` to order the transaction in their log, regardless of when they execute it, thus maintaining the original linearizable ordering. If there are no coordinator failures, a transaction would eventually be finalized through an IR inconsistent operation (`Commit/Abort`), which ensures that the decision would never be lost. As described above, for coordinator failures, the coordinator recovery protocol ensures that a backup coordinator would eventually send `Commit` or `Abort` to all participants.

4.5 TAPIR Extensions

We now describe four useful extensions to TAPIR.

4.5.1 Read-only Transactions

Since it uses a multi-versioned store, TAPIR easily supports globally-consistent read-only transactions at a timestamp. However, since TAPIR replicas are inconsistent, it is important to ensure that: (1) reads are up-to-date and (2) later write transactions do not invalidate the reads. To achieve these properties, TAPIR replicas keep a read timestamp for each object.

TAPIR's read-only transactions have a single round-trip fast path that sends the `Read` to only

one replica. If that replica has a *validated version* of the object – where the write timestamp precedes the snapshot timestamp and the read timestamp follows the snapshot timestamp – we know that the returned object is valid, because it is up-to-date, and will remain valid, because it will not be overwritten later. If the replica lacks a validated version, TAPIR uses the slow path and executes a `QuorumRead` through IR as an inconsistent operation. A `QuorumRead` updates the read timestamp, ensuring that at least $f + 1$ replicas do not accept writes that would invalidate the read.

The protocol for read-only transactions follows:

1. The TAPIR client chooses a *snapshot timestamp* for the transaction; for example, the client's local time.

2. The client sends `Read`(*key,version*), where *key* is what the application wants to read and *version* is the snapshot timestamp.

3. If the replica has a validated version of the object, it returns it. Otherwise, it returns a failure.

4. If the client could not get the value from the replica, it executes a `QuorumRead`(*key,version*) through IR as an inconsistent operation.

5. Any replica that receives `QuorumRead` returns the latest version of the object from the data store. It also writes the `Read` to the transaction log and updates the data store to ensure that it will not prepare for transactions that would invalidate the `Read`.

6. The client returns the object with the highest timestamp to the application.

As a quick sketch of correctness, it is always safe to read a version of the key that is *validated* at the snapshot timestamp. The version will always be valid at the snapshot timestamp because the write timestamp for the version is earlier than the snapshot timestamp and the read timestamp is after the snapshot timestamp. If the replica does not have a validated version, the replicated `QuorumRead` ensures that: (1) the client gets the latest version of the object (because at least 1 of any $f + 1$ replicas

must have it), and (2) a later write transaction cannot overwrite the version (because at least 1 of the $f + 1$ `QuorumRead` replicas will block it).

Since TAPIR also uses loosely synchronized clocks, it could be combined with Spanner's algorithm for providing externally consistent read-only transactions as well. This combination would require Spanner's TrueTime technology and waits at the client for the TrueTime uncertainty bound.[4]

4.5.2 Serializability

TAPIR is restricted in its ability to accept transactions out of order because it provides linearizability. Thus, TAPIR replicas cannot accept writes that are older than the last write for the same key, and they cannot accept reads of older versions of the same key.

However, if TAPIR's guarantees were weakened to *serializability*, then it can then accept proposed timestamps any time in the past as long as they respect serializable transaction ordering. This optimization requires tracking the timestamp of the transaction that last read and wrote each version.

With this optimization, TAPIR can now accept: (1) reads of past versions, as long as the read timestamp precedes the write timestamp of the next version, and (2) writes in the past (Tomas Write Rule), as long as the write timestamp follows the read timestamp of the previous version and precedes the write timestamp of the next version.

4.5.3 Synchronous Log Writes

Given the ability to synchronously log to durable storage (e.g. hard disk, NVRAM), we can reduce TAPIR's quorum requirements. As long as we can recover the log after failures, we can reduce the replica group size to $2f + 1$ and reduce all consensus and synchronization quorums to $f + 1$.

---

[4] While TAPIR provides external consistency for read-write transactions regardless of clock skew, this read-only protocol would provide linearizability guarantees only if the clock skew did not exceed the TrueTime bound, like Spanner. [48]

4.5.4 Retry Timestamp Selection

A client can increase the likelihood that participant replicas will accept its proposed timestamp by proposing a very large timestamp; this decreases the likelihood that the participant replicas have already accepted a higher timestamp. Thus, to decrease the chances of retrying forever, clients can exponentially increase their proposed timestamp on each retry.

4.5.5 Tolerating Very High Skew

If there is significant clock skew between servers and clients, TAPIR can use waits at the participant replicas to decrease the likelihood that transactions will arrive out of timestamp order. On receiving each `Prepare` message, the participant replica can wait (for the error-bound period) to see if other transactions with smaller timestamps will arrive. After the wait, the replica can process transactions in timestamp order. This wait increases the chances that the participant replica can process transactions in timestamp order and decreases the number of transactions that it will have to reject for arriving out of order.

4.6 Evaluation

In this section, our experiments demonstrate the following:

- TAPIR provides better latency *and* throughput than conventional transaction protocols in both the datacenter and wide-area environments.
- TAPIR's abort rate scales similarly to other OCC-based transaction protocols as contention increases.
- Clock synchronization sufficient for TAPIR's needs is widely available in both datacenter and wide-area environments.
- TAPIR provides performance comparable to systems with weak consistency guarantees and no transactions.

4.6.1 Experimental Setup

We ran our experiments on Google Compute Engine [80] (GCE) with VMs spread across 3 geographical regions – US, Europe and Asia – and placed in different availability zones within each geographical region. Each server has a virtualized, single core 2.6 GHz Intel Xeon, 8 GB of RAM and 1 Gb NIC.

Testbed Measurements

As TAPIR's performance depends on clock synchronization and round-trip times, we first present latency and clock skew measurements of our test environment. As clock skew increases, TAPIR's latency increases and throughput decreases because clients may have to retry more `Prepare` operations. It is important to note that TAPIR's performance depends on the *actual* clock skew, not a worst-case bound like Spanner [48].

We measured the clock skew by sending a ping message with timestamps taken on either end. We calculate skew by comparing the timestamp taken at the destination to the one taken at the source plus half the round-trip time (assuming that network latency is symmetric). Table 4.1 reports the average skew and latency between the three geographic regions. Within each region, we average over the availability zones. Our VMs benefit from Google's reliable wide-area network infrastructure; although we use UDP for RPCs over the wide-area, we saw negligible packet drops and little variation in round-trip times.

The average RTT between US-Europe was 110 ms; US-Asia was 165 ms; Europe-Asia was 260 ms. We found the clock skew to be low, averaging between 0.1 ms and 3.4 ms, demonstrating the feasibility of synchronizing clocks in the wide area. However, there was a long tail to the clock skew, with the worst case clock skew being around 27 ms – making it significant that TAPIR's performance depends on actual rather than worst-case clock skew. As our measurements show, the skew in this environment is low enough to achieve good performance.

Table 4.1: *RTTs and clock skews between Google Compute VMs.*

	Latency (ms)			Clock Skew (ms)		
	US	**Europe**	**Asia**	**US**	**Europe**	**Asia**
US	1.2	111.3	166.5	3.4	1.3	1.86
Europe	–	0.8	261.8	–	0.1	1.9
Asia	–	–	10.8	–	–	.3

Implementation

We implemented TAPIR in a transactional key-value storage system, called *TAPIR-KV*. Our prototype consists of 9094 lines of C++ code, not including the testing framework.

We also built two comparison systems. The first, *OCC-STORE*, is a "standard" implementation of 2PC and OCC, combined with an implementation of Multi-Paxos [119]. Like TAPIR, OCC-STORE accumulates a read and write set with read versions at the client during execution and then runs 2PC with OCC checks to commit the transaction. OCC-STORE uses a centralized timestamp server to generate transaction timestamps, which we use to version data in the multi-versioned storage system. We verified that this timestamp server was not a bottleneck in our experiments.

Our second system, *LOCK-STORE* is based on the Spanner protocol [48]. Like Spanner, it uses 2PC with S2PL and Multi-Paxos. The client acquires read locks during execution at the Multi-Paxos leaders and buffers writes. On `Prepare`, the leader replicates these locks and acquires write locks. We use loosely synchronized clocks at the leaders to pick transaction timestamps, from which the coordinator chooses the largest as the commit timestamp. We use the client as the coordinator, rather than one of the Multi-Paxos leaders in a participant shard, for a more fair comparison with TAPIR-KV. Lacking access to TrueTime, we set the TrueTime error bound to 0, eliminating the need to wait out clock uncertainty and thereby giving the benefit to this protocol.

Table 4.2: *Transaction profile for Retwis workload.*

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%

Workload

We use two workloads for our experiments. We first test using a synthetic workload based on the Retwis application [122]. Retwis is an open-source Twitter clone designed to use the Redis key-value storage system [173]. Retwis has a number of Twitter functions (e.g., add user, post tweet, get timeline, follow user) that perform `Puts` and `Gets` on Redis. We treat each function as a transaction, and generate a synthetic workload based on the Retwis functions as shown in Table 4.2.

Our second experimental workload is YCSB+T [58], an extension of YCSB [46] – a commonly-used benchmark for key-value storage systems. YCSB+T wraps database operations inside simple transactions such as read, insert or read-modify-write. However, we use our Retwis benchmark for many experiments because it is more sophisticated: transactions are more complex – each touches 2.5 shards on average – and longer – each executes 4-10 operations.

4.6.2 Single Datacenter Experiments

We begin by presenting TAPIR-KV's performance within a single datacenter. We deploy TAPIR-KV and the comparison systems over 10 shards, all in the US geographic region, with 3 replicas for each shard in different availability zones. We populate the systems with 10 million keys and make transaction requests with a Zipf distribution (coefficient 0.75) using an increasing number of closed-loop clients.

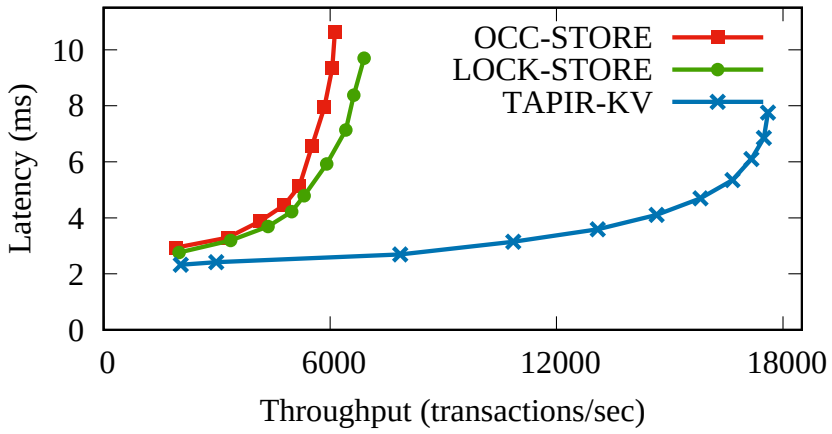


Figure 4.16: *TAPIR-KV datacenter comparison (Zipf coefficient 0.75).* We plot the average Retwis transaction Latency versus throughput.

Figure 4.16 shows the average latency for a transaction in our Retwis workload at different throughputs. At low offered load, TAPIR-KV has lower latency because it is able to commit transactions in a single round-trip to all replicas, whereas the other systems need two; its commit latency is thus reduced by 50%. However, Retwis transactions are relatively long, so the difference in *transaction* latency is relatively small.

Compared to the other systems, TAPIR-KV is able to provide roughly 3× the peak throughput, which stems directly from IR's weak guarantees: it has no leader and does not require cross-replica coordination. Even with moderately high contention (Zipf coefficient 0.75), TAPIR-KV replicas are able to inconsistently execute operations and still agree on ordering for transactions at a high rate.

4.6.3 Wide-Area Latency

For wide-area experiments, we placed one replica from each shard in each geographic region. For systems with leader-based replication, we fix the leader's location in the US and move the client between the US, Europe and Asia. Figure 4.17 gives the average latency for Retwis transactions using the same workload as in previous section.

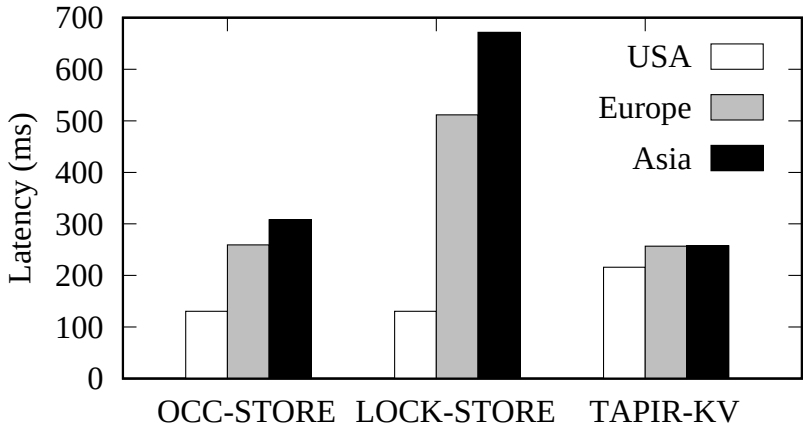


Figure 4.17: *TAPIR-KV wide-area evaluation.* We plot the average wide-area latency for Retwis transactions with the leader located in the US and client in US, Europe or Asia.

When the client shares a datacenter with the leader, the comparison systems are faster than TAPIR-KV because TAPIR-KV must wait for responses from all replicas, which takes 160 ms to Asia, while OCC-STORE and LOCK-STORE can commit with a round-trip to the local leader and one other replica, which is 115 ms to Europe.

When the leader is in a different datacenter, LOCK-STORE suffers because it must go to the leader on `Read` for locks, which takes up to 160 ms from Asia to the US, while OCC-STORE can go to a local replica on `Read` like TAPIR-KV. In our setup TAPIR-KV takes longer to `Commit`, as it has to contact the *furthest* replica, and the RTT between Europe and Asia is more expensive than two round-trips between US to Europe (likely because Google's traffic goes through the US). In fact, in this setup, IR's slow path, at two RTT to the two closest replicas, is *faster* than its fast path, at one RTT to the furthest replica. We do not implement the optimization of running the fast and slow paths in parallel, which could provide better latency in this case.

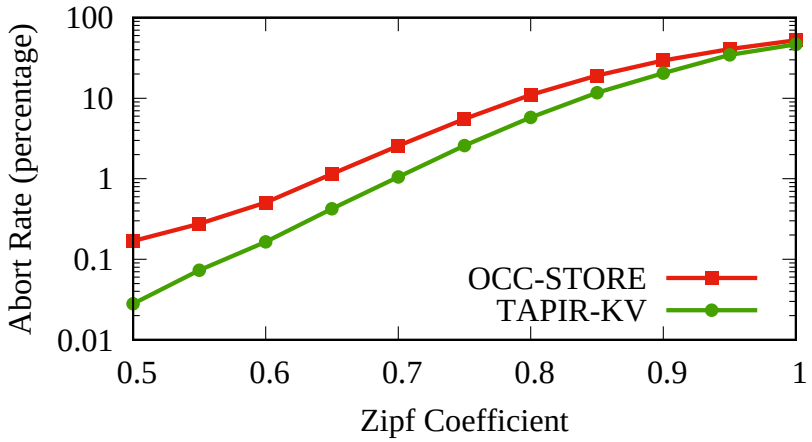


Figure 4.18: *TAPIR-KV abort rates.* We plot abort rates at varying Zipf co-efficients with a constant load of 5,000 transactions/second in a single datacenter.

4.6.4 Abort and Retry Rates

TAPIR is an optimistic protocol, so transactions can abort due to conflicts, as in other OCC systems. Moreover, TAPIR transactions can also be forced to abort or retry when conflicting timestamps are chosen due to clock skew. We measure the abort rate of TAPIR-KV compared to OCC-STORE, a conventional OCC design, for varying levels of contention (varying Zipf coefficients). These experiments run in a single region with replicas in three availability zones. We supply a constant load of 5,000 transactions/second.

With a uniform distribution, both TAPIR-KV and OCC-STORE have very low abort rates: 0.005% and 0.04%, respectively. Figure 4.18 gives the abort rate for Zipf co-efficients from 0.5 to 1.0. At lower Zipf co-efficients, TAPIR-KV has abort rates that are roughly an order of magnitude lower than OCC-STORE. TAPIR's lower commit latency and use of optimistic timestamp ordering reduce the time between `Prepare` and `Commit` or `Abort` to a single round-trip, making transactions less likely to abort.

Under heavy contention (Zipf coefficient 0.95), both TAPIR-KV and OCC-STORE have moderately high abort rates: 36% and 40%, respectively, comparable to other OCC-based systems like MDCC [113].

These aborts are primarily due to the most popular keys being accessed very frequently. For these workloads, locking-based systems like LOCK-STORE would make better progress; however, clients would have to wait for extended periods to acquire locks.

TAPIR rarely needs to retry transactions due to clock skew. Even at moderate contention rates, and with simulated clock skew of up to 50 ms, we saw less than 1% TAPIR retries and negligible increase in abort rates, demonstrating that commodity clock synchronization infrastructure is sufficient.

4.6.5 Comparison with Weakly Consistent Systems

We also compare TAPIR-KV with three widely-used eventually consistent storage systems, MongoDB [146], Cassandra [117], and Redis [173]. For these experiments, we used YCSB+T [58], with a single shard with 3 replicas and 1 million keys. MongoDB and Redis support master-slave replication; we set them to use synchronous replication by setting `WriteConcern` to `REPLICAS_SAFE` in MongoDB and the `WAIT` command [179] for Redis. Cassandra uses `REPLICATION_FACTOR=2` to store copies of each item at any 2 replicas.

Figure 4.19 demonstrates that the latency and throughput of TAPIR-KV is comparable to these systems. We do not claim this to be an entirely fair comparison; these systems have features that TAPIR-KV does not. At the same time, the other systems do not support distributed transactions – in some cases, not even single-node transactions – while TAPIR-KV runs a distributed transaction protocol that ensures strict serializability. Despite this, TAPIR-KV's performance remains competitive: it outperforms MongoDB, and has throughput within a factor of 2 of Cassandra and Redis, demonstrating that strongly consistent distributed transactions are not incompatible with high performance.

4.7 Related Work

Inconsistent replication shares the same principle as past work on commutativity, causal consistency and eventual consistency: operations that do not require ordering are more efficient. TAPIR leverages IR's weak guarantees, in combination with optimistic timestamp ordering and optimistic concurrency control, to provide semantics similar to past work on distributed transaction protocols but with both lower latency and higher throughput.

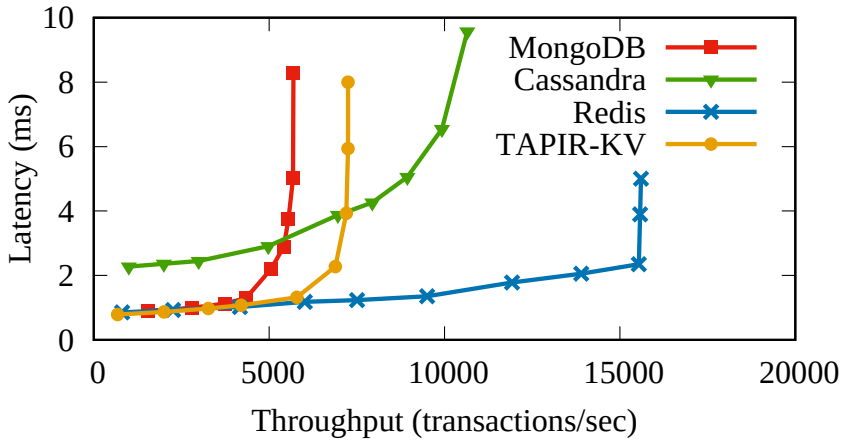


Figure 4.19: *TAPIR-KV comparison with weakly consistent storage systems.*

4.7.1 Replication

Transactional storage systems currently rely on strict consistency protocols, like Paxos [119] and VR [156]. These protocols enforce a strict serial ordering of operations and no divergence of replicas. In contrast, IR is more closely related to eventually consistent replication protocols, like Bayou [199], Dynamo [56] and others [116, 178, 117]. The key difference is that applications resolve conflicts after they happen with eventually consistent protocols, whereas IR **consensus** operations allow applications to decide conflicts and recover that decision later. As a result, applications can enforce higher-level guarantees (e.g., mutual exclusion, strict serializability) that they cannot with eventual consistency.

IR is also related to replication protocols that avoid coordination for *commutative operations* (e.g., Generalized Paxos [120], EPaxos [147]). These protocols are more general than IR because they do not require application invariants to be pairwise. For example, EPaxos could support invariants on bank account balances, while IR cannot. However, these protocols consider two operations to commute if their order does not matter when applied to *any* state, whereas IR requires only that they produce the same results *in a particular execution*. This is a form of state-dependent commutativity similar to SIM-commutativity [43]. As a result, in the example from Section 4.2.1, EPaxos would consider any operations on the same lock to conflict, whereas IR would allow two unsuccessful `Lock`

Table 4.3: *Comparison of read-write transaction protocols in replicated transactional storage systems.* For each system, we list the replication protocol used, the message delays for reading and committing a transaction, the number of messages processed by the bottlenecked node, and the isolation level and transaction model provided by the system.

Transaction System	Replication Protocol	Read Latency	Commit Latency	Msg At Bottleneck	Isolation Level	Transaction Model
Spanner [48]	Multi-Paxos [119]	2 (leader)	4	$2n$ + reads	Strict Serializable	Interactive
MDCC [113]	Gen. Paxos [120]	2 (any)	3	$2n$	Read-Committed	Interactive
Repl. Commit [134]	Paxos [119]	$2n$	4	2	Serializable	Interactive
CLOCC [5, 127]	VR [156]	2 (any)	4	$2n$	Serializable	Interactive
Lynx [222]	Chain Repl. [209]	–	$2n$	2	Serializable	Stored procedure
TAPIR	IR	2 (to any)	2	2	Strict Serializable	Interactive

operations to the same lock to commute.

4.7.2 Distributed Transactions

A technique similar to optimistic timestamp ordering was first explored by Thomas [201], while CLOCC [5] was the first to combine it with loosely synchronized clocks. We extend Thomas's algorithm to: (1) support multiple shards, (2) eliminate synchronous disk writes, and (3) ensure availability across coordinator failures. Spanner [48] and Granola [50] are two recent systems that use loosely synchronized clocks to improve performance for read-only transactions and independent transactions, respectively. TAPIR's use of loosely synchronized clocks differs from Spanner's in two key ways: (1) TAPIR depends on clock synchronization only for performance, not correctness, and (2) TAPIR's performance is tied to the *actual* clock skew, not TrueTime's worst-case estimated bound. Spanner's approach for read-only transactions complements TAPIR's high-performance read-write transactions, and the two could be easily combined.

CLOCC and Granola were both combined with VR [127] to replace synchronous disk writes with in-memory replication. These combinations still suffer from the same redundancy – enforcing ordering both at the distributed transaction and replication level – that we discussed in Section 4.1. Other layered protocols, like the examples shown in Table 4.3, have a similar performance limitation.

Some previous work included in Table 4.3 improves throughput (e.g., Warp [65], Transaction Chains [222], Tango [23]), while others improve performance for read-only transactions (e.g., MegaStore [22], Spanner [48]) or other limited transaction types (e.g., Sinfonia's mini-transactions [7], Granola's independent transactions [50], Lynx's transaction chains [222], and MDCC's commutative transactions [113]) or weaker consistency guarantees [132, 188]. In comparison, TAPIR is the first transaction protocol to provide better performance (both throughput and latency) for general-purpose, read-write transactions using replication.

4.8 Summary

This chapter demonstrates that it is possible to build distributed transactions with better performance and strong consistency semantics by building on a replication protocol with *no* consistency. We present inconsistent replication, a new replication protocol that provides fault tolerance without consistency, and TAPIR, a new distributed transaction protocol that provides linearizable transactions using IR. We combined IR and TAPIR in TAPIR-KV, a distributed transactional key-value storage system. Our experiments demonstrate that TAPIR-KV lowers commit latency by 50% and increases throughput by 3× relative to conventional transactional storage systems. In many cases, it matches the performance of weakly-consistent systems while providing much stronger guarantees.

5 Conclusion

Mobile/cloud applications are the most common user-facing applications today. However, much like the first mainframe applications, programmers continue to build them without the benefit of general-purpose system abstractions. This thesis presented a vision for a new mobile/cloud operating system designed to meet the requirements of these modern applications.

We introduced three systems – Sapphire, Diamond and TAPIR – each designed to support mobile/cloud applications with new abstractions and mechanisms. Sapphire presents a new form of run-time management with Sapphire Objects as the basic unit of deployment and pluggable Deployment Managers as customizable run-time management libraries. Diamond provides memory management for mobile/cloud applications by supporting a new form of distributed shared data types – reactive data types – and transactions that automatically and reliably propagate updates to shared data with strong guarantees. TAPIR is a new storage system designed to meet the needs of mobile/cloud applications for distributed transactions and low latency, giving programmers the benefit of both strong guarantees and good performance.

Together, these systems form the basis for a new mobile/cloud operating system. Like desktop operating systems before them, this new OS simplifies the development and management of new applications, making it easier for programmers to build larger, more complex applications in the future.

5.1 Looking Forward: The Path to Adoption

A crucial question for any researcher is the path to adoption for their research. Unfortunately for operating system researchers, it has rarely been easy to gain the momentum needed for the adoption of a new OS. Issues like backward compatibility for existing applications and support for a variety of existing hardware platforms make it almost impossible today for a new desktop operating system to achieve a significant user base.

A mobile/cloud OS avoids most of these issues. First, it does not require privileged mode execution or complete control over the entire machine. Mobile/cloud applications can bundle their OS along with their code for installation on both mobile devices and cloud servers. Next, a mobile/cloud OS needs no support for legacy desktop applications, and the traditional OS on each mobile device and cloud server ensures that legacy mobile/cloud applications can run alongside the new OS. In fact, many mobile/cloud OSes could be developed and adopted in parallel with little conflict. Finally, the traditional OS also unifies a diversity of hardware platforms and devices. As a result, a mobile/cloud OS needs to run only on a limited number of popular OSes that support today's mobile devices and cloud servers. In particular, almost all mobile devices run iOS or a variant of Android; thus, any mobile/cloud OS would need to support just two mobile OSes.

While limited barriers have lead to the plethora of mobile/cloud systems today, barriers to adoption still exist. The first and most significant is portability. Because mobile/cloud applications expect to run and persist forever, they must be able to easily switch between systems if the system becomes unsupported, too expensive or unable to meet the application's need. The solution to this problem is standardization. Virtual machines and containers avoid portability issues by having a largely standardized interface that lets programmers move them across systems when needed.

Today's morass of mobile/cloud systems is ripe for standardization. POSIX was a significant step forward for UNIX-like operating systems because it enabled the development of many interchangeable operating systems with different features and hardware support. A similar standardized interface for a distributed mobile/cloud OS would be an enormous step towards a world that is easier for application programmers to navigate. Mobile/cloud programmers could finally develop applications that are

agnostic to the mobile OS or hardware. Further, programmers could switch between mobile/cloud operating systems as necessary, eliminating their worries about portability.

There are other more practical barriers including: (1) incentives for programmers to use a mobile/cloud OS instead of existing solutions, and (2) incentives for companies or communities to build mobile/cloud OSes. The former is not significant; programmers today develop new mobile/cloud applications at such a rapid rate that even were a small fraction of them use to an OS, the OS would still have a significant user base. Further, classes on developing mobile/cloud applications – always a popular topic with start-up savvy undergraduates – could introduce mobile/cloud OSes to programmers. The latter is a more significant issue because it is unclear how to monetize such an OS (e.g., it would not be sold with computers like a traditional OS). However, Docker has been successful thus far in selling subscriptions for its container platform [103], so selling OS cloud services presents a viable option.

5.2 Concluding Remarks

Mobile/cloud applications have literally defined a new generation of people, dubbed the *iGen* by a recent book and Atlantic article [204]. Even as it becomes increasingly clear that new systems are needed to support today's applications, systems research continues to focus on the same types of systems that it did 40 years ago (e.g., traditional operating systems, file systems, distributed storage systems). Instead, much of today's innovations in systems are driven by start-ups and industry. They build one-off solutions to their immediate problems, which has lead to an ad-hoc collection of systems that are impossible for programmers to navigate. In the past, it has always been researchers that have the motivation and vision to design general-purpose systems with a principled and careful approach (e.g., UNIX). Thus, today's researchers must apply themselves to inventing new systems, not re-designing old ones, if application programmers are to avoid being systems experts in the future.

Bibliography

[1] Nim, Feb 2016. https://en.wikipedia.org/wiki/Nim#The_100_game.

[2] 2Do. webpage, 2017. https://www.2doapp.com/.

[3] David Abrahams and Stefan Seefeld. Boost C++ libraries, 2015. http://www.boost.org/doc/libs/1_60_0/libs/python/doc/html/index.html.

[4] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. Thialfi: A client notification service for internet-scale applications. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2011.

[5] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proceedings of ACM SIGMOD Conference*, 1995.

[6] Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. Mobile data sync in a blink. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[7] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2007.

[8] Steph Alvos-Bock. The convergence of iOS and OSX user interface design, July 2015. http://www.solstice-mobile.com/blog/the-convergence-of-ios-and-os-x-user-interface-design.

[9] Amazon. Amazon Elastic Cloud Compute, 2013. http://aws.amazon.com/ec2/.

[10] webpage. https://aws.amazon.com/elasticbeanstalk/.

[11] webpage. https://aws.amazon.com/lambda/.

[12] Amazon retail. webpage, 2017. https://www.amazon.com/.

[13] Apache. Apache Thrift, 2013. http://thrift.apache.org.

[14] Apple. The Swift programming language, 2016. https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/#//apple_ref/doc/uid/TP40014097-CH3-ID0.

[15] Apple push notification service, 2015. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html.

[16] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[17] webpage. https://azure.microsoft.com/en-us/services/app-service/.

[18] webpage. https://azure.microsoft.com/en-us/services/container-service/.

[19] webpage. https://aws.amazon.com/ecs/details/.

[20] webpage. https://azure.microsoft.com/en-us/services/functions/.

[21] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2014.

[22] Jason Baker, Chris Bond, James C Corbett, J.J. Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of CIDR*, 2011.

[23] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2013.

[24] Basil for iOS. webpage, 2017. http://basil-app.com/.

[25] D.S. Batoory, J.R. Barnett, Jorge F. Garza, Kenneth Paul Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. Genesis: An extensible database management system. *IEEE Transactions on Software Engineering*, 1988.

[26] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[27] Nalini Moti Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Michael Dahlin, and Robert Grimm. PADS: A policy architecture for distributed storage systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[28] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1990.

[29] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[30] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1995.

[31] Ken Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1987.

[32] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of ACM SIGMOD Conference*, 1986.

[33] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of USENIX Annual Technical Conference*, 2013.

[34] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, April 2016.

[35] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[36] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2011.

[37] John Callaham. Yes, windows 10 is the next version of windows phone. Windows Central, Sept 2014. http://www.windowscentral.com/yes-windows-10-next-version-windows-phone.

[38] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. *Object and file management in the EXODUS extensible database system*. Computer Sciences Department, University of Wisconsin, 1986.

[39] Sharma Chakravarthy. Sentinel: an object-oriented DBMS with event-based rules. In *Proceedings of ACM SIGMOD Conference*, 1997.

[40] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008.

[41] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2007.

[42] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[43] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2013.

[44] Kevin Conaway. Pyscrabble. http://pyscrabble.sourceforge.net/.

[45] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[46] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2010.

[47] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the Symposium on Formal Methods for Components and Objects (FMCO)*, 2006.

[48] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[49] Corona SDK, 2013. http://www.coronalabs.com/.

[50] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of USENIX Annual Technical Conference*, 2012.

[51] James Cowling, Dan R.K. Ports, Barbara Liskov, Raluca Ada Popa, and Abhijeet Gaikwad. Census: Location-aware membership management for large-scale distributed systems. *Proceedings of USENIX Annual Technical Conference*, 2009.

[52] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

[53] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramana, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[54] Day One | A simple and elegant journal for iPhone, iPad, and Mac. webpage, 2017. http://dayoneapp.com/.

[55] DB-engine's ranking of key-value stores, 10 2015. http://db-engines.com/en/ranking/key-value+store.

[56] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2007.

[57] deepstream. deepstream.io: a scalable server for realtime web apps. https://deepstream.io/.

[58] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *Proceedings of IEEE International Conference on Data Engineering Workshops (ICDEW)*, 2014.

[59] Diaro: Diary, Journal, Notes for iOS and Android. webpage, 2017. http://www.diaroapp.com/.

[60] Dan Diephouse and Paul Brown. Building a highly scalable, open source Twitter clone, 2009. http://fr.slideshare.net/multifariousprb/building-a-highly-scalable-open-source-twitter-clone.

[61] Google Drive, 2016. http://drive.google.com.

[62] Dropbox, 2013. http://dropbox.com.

[63] eMarketer. Mobile game revenues to grow 16.5% in 2015, surpassing $3 billion, Feb 2015. http://www.emarketer.com/Article/Mobile-Game-Revenues-Grow-165-2015-Surpassing-3-Billion/1012063.

[64] Dawson R. Engler, M. Frans Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1995.

[65] Robert Escriva, Bernard Wong, and Emin GÃijn Sirer. Warp: Multi-key transactions for key-value stores. Technical report, Cornell, Nov 2013.

[66] Evernote. webpage, 2017. https://evernote.com/.

[67] Facebook. Wikipedia, 2017. https://en.wikipedia.org/wiki/Facebook.

[68] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[69] Firebase, 2015. https://www.firebase.com/.

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

[71] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1997.

[72] webpage. https://www.gamesparks.com/.

[73] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key-value store. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[74] Git, 2015. https://git-scm.com/.

[75] Google. Agera. https://github.com/google/agera.

[76] Google. Google App Engine developer's guide, 2013. https://developers.google.com/appengine/docs/.

[77] Google. Android standalone toolchain, 2016. http://developer.android.com/ndk/guides/standalone_toolchain.html.

[78] Google. Wikipedia, 2017. https://en.wikipedia.org/wiki/Google.

[79] Google App Engine. https://cloud.google.com/appengine/.

[80] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[81] Google Docs. Wikipedia, 2017. https://en.wikipedia.org/wiki/Google_Docs,_Sheets_and_Slides.

[82] webpage. https://cloud.google.com/functions/.

[83] 2013. https://developers.google.com/google-apps/marketplace/sso.

[84] Google web toolkit. https://developers.google.com/web-toolkit/, October 2012.

[85] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. COMET: Code offload by migrating execution transparently. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[86] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 2006.

[87] Grid Diary. webpage, 2017. http://griddiaryapp.com/en/.

[88] HAProxy: A reliable, high-performance TCP/HTTP load balancer, 2013. http://haproxy.1wt.eu/.

[89] Maurice Herlihy. Optimistic concurrency control for abstract data types. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1986.

[90] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 1990.

[91] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *eurosys*, 2016.

[92] Todd Hoff. Playfish's social gaming architecture - 50 million monthly users and growing. In *High Scalability Blog*. High Scalability, Sept 2010. highscalability.com/blog/2010/9/21/playfishs-social-gaming-architecture-50-million-monthly-user.html.

[93] Todd Hoff. How twitter stores 250 million tweets a day using mysql. In *High Scalability Blog*. High Scalability, December 2011. http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html.

[94] Todd Hoff. The architecture that Twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds. In *High Scalability Blog*. High Scalability, July 2013. http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html.

[95] Todd Hoff. Why amazon retail went to a service oriented architecture. In *High Scalability Blog*. High Scalability, July 2016. http://highscalability.com/blog/2016/7/13/why-amazon-retail-went-to-a-service-oriented-architecture.html.

[96] Matthew Humphries. Ellen DeGeneres crashes Twitter with Oscar selfie, 3 2014. http://www.geek.com/mobile/ellen-degeneres-crashes-twitter-with-an-oscars-selfie-1586464/.

[97] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[98] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, 2010.

[99] Apple iCloud, 2016. https://www.icloud.com/.

[100] Apache Incubator. Apache Usergrid. http://usergrid.apache.org/.

[101] Instagram. Wikipedia, 2017. https://en.wikipedia.org/wiki/Instagram.

[102] iWork. Wikipedia, 2017. https://en.wikipedia.org/wiki/IWork.

[103] Joab Jackson. Docker launches an enterprise edition, refines market strategy. The New Stack, March 2017. https://thenewstack.io/docker-launches-enterprise-edition-refines-market-strategy/.

[104] JavaCPP: The missing bridge between Java and native C++. github, Mar 2016. https://github.com/bytedeco/javacpp.

[105] Jetty web server. http://www.eclipse.org/jetty/.

[106] Roland Kuhn Jonas Boner, Dave Farley and Martin Thompson. The reactive manifesto, Sept 2014. http://www.reactivemanifesto.org/.

[107] Anthony D Joseph, Alan F de Lespinasse, Joshua A Tauber, David K Gifford, and M Frans Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1995.

[108] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1987.

[109] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1997.

[110] Ryan King. Announcing snowflake, June 2010. https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html.

[111] James J Kistler and Mahadev Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 1992.

[112] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1999.

[113] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[114] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of NetDB*, 2011.

[115] Hsiang-Tsung Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981.

[116] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 1992.

[117] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[118] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 1994.

[119] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 2001.

[120] Leslie Lamport. Generalized consensus and Paxos. Technical Report 2005-33, Microsoft Research, 2005.

[121] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, October 2006.

[122] Costin Leau. Spring Data Redis - Retwis-J, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[123] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2011.

[124] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1975.

[125] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 1989.

[126] B. F. Lieuwen, N. Gehani, and R. Arlein. The Ode active database: trigger semantics and implementation. In *Proceedings of International Conference on Data Engineering (ICDE)*, Feb 1996.

[127] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1999.

[128] Barbara Liskov and James Cowling. Viewstamped replication revisited, 2012.

[129] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1987.

[130] Jed Liu, Michael D. George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2009.

[131] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2011.

[132] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2011.

[133] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 2001.

[134] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2013.

[135] Alessandro Margara and Guido Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of ACM Conference on Distributed and Event-Based Systems (DEBS)*. ACM, 2014.

[136] Markets and Markets. Backend as a service (BaaS) market worth 28.10 billion USD by 2020. http://www.marketsandmarkets.com/PressReleases/baas.asp.

[137] martypdx. Firebase data consistency across multiple nodes. Stack Overflow, Apr 2015. http://stackoverflow.com/questions/29947898/firebase-data-consistency-across-multiple-nodes.

[138] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the USENIX International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[139] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[140] memcached, 2015. http://memcached.org/.

[141] Meteor, 2015. http://www.meteor.com.

[142] Microsoft Office Online. Wikipedia, 2017. https://en.wikipedia.org/wiki/Office_Online.

[143] Moblie HTML5, 2013. http://mobilehtml5.org.

[144] MobiRuby, 2013. http://mobiruby.org/.

[145] Mission Mode. New mobile apps revolutionize how organizations respond to crises and operations issues, Aug 2014. http://www.missionmode.com/new-mobile-apps-revolutionize-organizations-respond-crises-operations-issues/.

[146] MongoDB: A open-source document database, 2013. http://www.mongodb.org/.

[147] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2013.

[148] Mozilla. Kinto. http://kinto.readthedocs.org/en/latest/.

[149] Roi Mulia. Firebase - maintain/guarantee consistency. Stack Overflow, Jan 2016. http://stackoverflow.com/questions/34678083/firebase-maintain-guarantee-data-consistency.

[150] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2001.

[151] My CookBook Online. webpage, 2017. http://mycookbook-android.com/site/.

[152] MySpace. webpage, 2017. https://www.myspace.com/.

[153] MySQL, 2013. http://www.mysql.com/.

[154] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM Java Grande Conference*, 1999.

[155] Node.js, 2013. http://nodejs.org/.

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

[157] openio. openio.io: object storage grid for apps. http://openio.io/.

[158] Paprika Recipe Manager for iPad, iPhone, Mac, Android, and Windows. webpage, 2017. https://www.paprikaapp.com/.

[159] Parse, 2013. http://parse.com.

[160] Parse, 2015. http://www.parse.com.

[161] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V Madhyastha, and Cristian Ungureanu. Simba: tunable end-to-end data consistency for mobile apps. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2015.

[162] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 2000.

[163] webpage. https://playfab.com/.

[164] Wikipedia. https://en.wikipedia.org/wiki/Playfish.

[165] Robey Pointer, N Kallen, E Ceaser, and J Kalucki. Introducing flockdb, May 2010. https://blog.twitter.com/engineering/en_us/a/2010/introducing-flockdb.html.

[166] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[167] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[168] PostgreSQL, 2013. http://www.postgresql.org/.

[169] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraedsullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[170] React: A JavaScript library for building user interfaces. Github, 2016. https://facebook.github.io/react/.

[171] ReactiveX: An api for asynchronous programming with observable streams, 2016. http://reactivex.io/.

[172] Redis. Wait numslaves timeout. http://redis.io/commands/WAIT.

[173] Redis: Open source data structure server, 2013. http://redis.io/.

[174] Riak, 2015. http://basho.com/products/riak-kv/.

[175] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the ACM/IFIP/USENIX Middleware Conference*, 2001.

[176] Amazon S3, 2013. http://aws.amazon.com/s3/.

[177] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K Nurminen, Matti Kemppainen, and Pan Hui. SmartDiet: offloading popular apps to save energy. *ACM SIGCOMM Computer Communication Review*, 2012.

[178] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 2005.

[179] Salvatore Sanfilippo. WAIT: synchronous replication for Redis. http://antirez.com/news/66, December 2013.

[180] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[181] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, 2006.

[182] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the International Symposium on Self-Stabilizing Systems (SSS)*, 2011.

[183] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. In *Proceedings of the Workshop on the Management of Replicated Data*, 1990.

[184] Simplenote. webpage, 2017. https:/,/simplenote.com/.

[185] Snapchat. Wikipedia, 2017. https://en.wikipedia.org/wiki/Snapchat.

[186] Simple object access protocol. http://www.w3.org/TR/soap/.

[187] socketcluster.io. socketcluster.io: a scalable framework for realtime apps and microservices. http://socketcluster.io/#!/.

[188] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2011.

[189] Spotify: Music for everyone. webpage, 2017. https://www.spotify.com/.

[190] Sqlite home page, 2015. https://www.sqlite.org/.

[191] Square cash. https://cash.me/.

[192] Twitter's 'starling' released as open source, Jan 2008. https://blog.twitter.com/official/en_us/a/2008/twitters-starling-released-as-open-source.html.

[193] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[194] Michael Stonebraker and Joseph M Hellerstein. *Readings in Database Systems*. Morgan Kaufmann San Francisco, 1998.

[195] Jacob Strauss, Justin Mazzola Paluska, Chris Lesniewski-Laas, Bryan Ford, Robert Morris, and M Frans Kaashoek. Eyo: Device-transparent personal storage. In *Proceedings of USENIX Annual Technical Conference*, 2011.

[196] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[197] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with Project Voldemort. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[198] Andrew S Tanenbaum, Robbert Van Renesse, Hans Van Staveren, Gregory J Sharp, and Sape J Mullender. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 1990.

[199] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 1995.

[200] Things. webpage, 2017. https://culturedcode.com/things/.

[201] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[202] Global social gaming market to reach US$17.4 bn by 2019 propelled by rising popularity of fun games. Transparency Market Research Press Release, Sept 2015. http://www.transparencymarketresearch.com/pressrelease/social-gaming-market.htm.

[203] Trello. Wikipedia, 2017. https://en.wikipedia.org/wiki/Trello.

[204] Jean M. Twenge. Have smartphones destroyed a generation? The Atlantic, September 2017. https://www.theatlantic.com/magazine/archive/2017/09/has-the-smartphone-destroyed-a-generation/534198/.

[205] Twimight open-source Twitter client for Android, 2013. http://code.google.com/p/twimight/.

[206] Twitter. Twitter developer API, 2014. https://dev.twitter.com/overview/api.

[207] Twitter. private communication, 9 2015.

[208] Twitter. Wikipedia, 2017. https://en.wikipedia.org/wiki/Twitter.

[209] Robbert van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[210] Venmo. https://venmo.com/.

[211] Vine. Wikipedia, 2017. https://en.wikipedia.org/wiki/Vine_%28service%29.

[212] Voldemort: A distributed database, 2013. http://www.project-voldemort.com/voldemort/.

[213] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of USENIX Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[214] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 1992.

[215] William E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 1989.

[216] David A. Wheeler. SLOCCount, 2013. http://www.dwheeler.com/sloccount/.

[217] Words with friends. website, 2017. https://www.zynga.com/games/words-friends.

[218] Wunderlist. webpage, 2017. https://www.wunderlist.com/.

[219] Zen load balancer, 2013. http://www.zenloadbalancer.com/.

[220] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurhty, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2015.

[221] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[222] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2013.

A | Open-source Code

All code for this thesis is open-source and can be found at github.

- Sapphire: https://github.com/UWSysLab/Sapphire
- Diamond: https://github.com/UWSysLab/diamond
- TAPIR: https://github.com/UWSysLab/tapir

B | TLA+ Specification

B.1 Inconsistent Replication Specification

MODULE $IR\_consensus$

This is a TLA+ specification of the *Inconsistent* Replication algorithm. (And a mechanically-checked proof of its correctness using $TLAPS$)

EXTENDS $FiniteSets$, $Naturals$, TLC, $TLAPS$

Constants

Constant parameters: Replicas: the set of all replicas (Replica IDs)

Clients: the set of all clients (Client IDs)

$Quorums$: the set of all quorums $SuperQuorums$: the set of all super quorums Results: the set of all possible result types $OperationBody$: the set of all possible operation bodies

(with arguments, etc. - can be infinite)

S: shard id of the shard $Replicas$ constitute

f: maximum number of failures allowed (half of n)

Constants used to bound variables, for model checking (Nat is bounded) max_vc: maximum number of View-Changes allowed for each replicas max_req: maximum number of op requests performed by clients

CONSTANTS $Replicas$, $Clients$, $Quorums$, $SuperQuorums$, $Results$, $OpBody$,
$AppClientFail$, $AppReplicaFail$,
$SuccessfulInconsistentOp(\_)$, $SuccessfulConsensusOp(\_, \_)$,

$Merge(\_, \_)$,

$Sync(\_)$,

$ExecInconsistent(\_)$,

$ExecConsensus(\_)$,

$Decide(\_)$,

f,

S, $Shards$, $S =$ shard id

max_vc, max_req

ASSUME $IsFiniteSet(Replicas)$

ASSUME $QuorumAssumption \triangleq$

$\land\ Quorums \subseteq \text{SUBSET } Replicas$

$\land\ SuperQuorums \subseteq \text{SUBSET } Replicas$

$\land\ \forall\ Q_1,\ Q_2 \in Quorums : Q_1 \cap Q_2 \neq \{\}$

$\land\ \forall\ Q \in Quorums,\ R_1,\ R_2 \in SuperQuorums :$

$$Q \cap R_1 \cap R_2 \neq \{\}$$

ASSUME $FailuresAssumption \triangleq$

$\forall\ Q \in Quorums : Cardinality(Q) > f$

The possible states of a replica and the two types of operations currently defined by *IR*.

$ReplicaState \triangleq$ {"NORMAL", "FAILED", "RECOVERING", "VIEW-CHANGING"}

$ClientState \triangleq$ {"NORMAL", "FAILED"}

$OpType \triangleq$ {"Inconsistent", "Consensus"}

$OpStatus \triangleq$ {"TENTATIVE", "FINALIZED"}

Definition of operation space

$MessageId \triangleq [cid : Clients,\ msgid : Nat]$

$Operations \triangleq [type : OpType,\ body : OpBody]$

Message is defined to be the set of all possible messages

$TODO$: Assumptions

Assume unique message ids

Assume no more than f replica failures

We use shart to specify for what shard this message was

(we share the variables)

$$
\begin{aligned}
Message \triangleq\ & [type : \{\text{“REQUEST”}\}, \\
& id : MessageId, \\
& op : Operations] \\
\cup\ & [type : \{\text{“REPLY”}\}, \quad \text{reply no result} \\
& id : MessageId, \\
& v : Nat, \\
& src : Replicas] \\
\cup\ & \\
& [type : \{\text{“REPLY”}\}, \quad \text{reply with result} \\
& id : MessageId, \\
& v : Nat, \\
& res : Results, \\
& src : Replicas] \\
& v = \text{ view num.} \\
\cup\ & \\
& [type : \{\text{“START-VIEW-CHANGE”}\}, \\
& v : Nat, \\
& src : Replicas] \\
\cup\ & \\
& [type : \{\text{“DO-VIEW-CHANGE”}\},
\end{aligned}
$$

r : SUBSET ([$msgid$: $MessageId$,
op : $Operations$,
res : $Results$]
$\cup$ [$msgid$: $MessageId$,
op : $Operations$]),
v : Nat,
src : $Replicas$,
dst : $Replicas$]

$\cup$

[$type$: {"START-VIEW"},
v : Nat,
src : $Replicas$]

$\cup$

[$type$: {"START-VIEW-REPLY"},
v : Nat,
src : $Replicas$,
dst : $Replicas$]

$\cup$

[$type$: {"FINALIZE"}, finalize with no result
id : $MessageId$,
op : $Operations$,
res : $Results$]

$\cup$

[$type$: {"FINALIZE"}, finalize with result
id : $MessageId$,
op : $Operations$,
res : $Results$]

$\cup$ [$type$: {"CONFIRM"},

$v : Nat,$

$id : MessageId,$

$op : Operations,$

$res : Results,$

$src : Replicas]$

Variables and State Predicates

Variables:

1. State at each replica:

$rState$ = Denotes current replica state. Either:

- $NORMAL$ (processing operations)

- VIEW-CHANGING (participating in recovery)

$rRecord$ = Unordered set of operations and their results

$rViewNumber$ = current view number

2. State of communication medium: $sentMsg$ = sent (but not yet received) messages

3. State at client:

$cCurrentOperation$ = crt operation requested by the client

$cMmessageCounter$ = the message I must use for the next operation

VARIABLES $rState,\ rRecord,\ rViewNumber,\ rViewReplies,\ sentMsg,\ cCrtOp,$
$cCrtOpToFinalize,\ cMsgCounter,\ cCrtOpReplies,\ cCrtOpConfirms,$
$cState,\ aSuccessful,\ gViewChangesNo$

Defining these tuples makes it easier to express which varibles remain unchanged

$rVars \triangleq \langle rState,\ rRecord,\ rViewNumber,\ rViewReplies \rangle$ Replica variables.

$cVars \triangleq \langle cCrtOp,$ current operation at a client
$cCrtOpToFinalize,$
$cCrtOpReplies,$ current operation replies
$cCrtOpConfirms,$

$cMsgCounter$,

$cState\rangle$ Client variables.

$aVars \triangleq \langle aSuccessful\rangle$ Application variables

$oVars \triangleq \langle sentMsg,\ gViewChangesNo\rangle$ Other variables.

$vars \triangleq \langle rVars,\ cVars,\ oVars\rangle$ All variables.

$TypeOK \triangleq$

$\land\ rState[S] \in [Replicas \to ReplicaState]$

$\land\ rRecord[S] \in [Replicas \to \text{SUBSET}\ ([msgid : MessageId,$
$op : Operations,$
$res : Results,$
$status : OpStatus]$
$\cup\ [msgid : MessageId,$
$op : Operations,$
$status : OpStatus])]$

$\land\ rViewNumber[S] \in [Replicas \to Nat]$

$\land\ rViewReplies[S] \in [Replicas \to \text{SUBSET}\ [type : \{\text{“do-view-change”},$
“start-view-reply”$\},$
$viewNumber : Nat,$
$r : \text{SUBSET}\ ([msgid : MessageId,$
$op : Operations,$
$res : Results,$
$status : OpStatus]$
$\cup\ [msgid : MessageId,$
$op : Operations,$
$status : OpStatus]),$
$src : Replicas]]$

$\land\ sentMsg[S] \in \text{SUBSET}\ Message$

$\land\ cCrtOp[S] \in [Clients \to Operations \cup \{\langle\rangle\}]$

$\land\ cCrtOpToFinalize \in [Clients \to Operations \cup \{\langle\rangle\}]$

$\land\ cCrtOpReplies[S]\ \in [Clients \to \textsc{subset}\ ([viewNumber : Nat,$
$res : Results,$
$src : Replicas]$
$\cup\ [viewNumber : Nat,$
$src : Replicas])]$

$\land\ cCrtOpConfirms[S] \in [Clients \to \textsc{subset}\ [viewNumber : Nat,$
$res : Results,$
$src : Replicas]]$

$\land\ cMsgCounter[S] \in [Clients \to Nat]$

$\land\ cState \in [Clients \to ClientState]$

$\land\ aSuccessful \in \textsc{subset}\ ([mid : MessageId,$
$op : Operations,$
$res : Results]$
$\cup [mid : MessageId,$
$op : Operations])$

$\land\ gViewChangesNo[S] \in Nat$

$Init \triangleq$

$\land\ rState = [r \in Replicas \mapsto \text{“NORMAL”}]$

$\land\ rRecord = [r \in Replicas \mapsto \{\}]$

$\land\ rViewNumber = [r \in Replicas \mapsto 0]$

$\land\ rViewReplies = [r \in Replicas \mapsto \{\}]$

$\land\ sentMsg = \{\}$

$\land\ cCrtOp = [c \in Clients \mapsto \langle\rangle]$

$\land\ cCrtOpToFinalize = [c \in Clients \mapsto \langle\rangle]$

$\land\ cCrtOpReplies = [c \in Clients \mapsto \{\}]$

$\land\ cCrtOpConfirms = [c \in Clients \mapsto \{\}]$

$\land\ cMsgCounter = [c \in Clients \mapsto 0]$

$\land\ cState = [c \in Clients \mapsto \text{“NORMAL”}]$

$\land\ aSuccessful = \{\}$

$\land\ gViewChangesNo = 0$

Actions

$Send(m) \triangleq sentMsg' = [sentMsg \text{ EXCEPT } ![S] = @ \cup \{m\}]$

Client Actions

Note: CHOOSE does not introduce nondeterminism (the same value is chosen each time)

Client sends a request

$ClientRequest(c, op) \triangleq$

$\land\ cCrtOp[S][c] = \langle\rangle$ the client is not waiting for a result of another operation

$\land\ cCrtOpToFinalize[S][c] = \langle\rangle$

$\land\ cMsgCounter' = [cMsgCounter \text{ EXCEPT } ![S][c] = @ + 1]$

$\land\ cCrtOp' = [cCrtOp \text{ EXCEPT } ![S][c] = op]$

$\land\ Send([type \mapsto \text{“REQUEST”},$
$id \mapsto [cid \mapsto c, msgid \mapsto cMsgCounter[S][c] + 1],$
$op \mapsto op])$

$\land\ \text{UNCHANGED } \langle rVars, aVars, cCrtOpReplies, cCrtOpToFinalize,$
$cCrtOpConfirms, cState, gViewChangesNo\rangle$

$\land\ cMsgCounter[S][c] < max\_req$ BOUND the number of requests a client can make

Client received a reply

$ClientReceiveReply(c) \triangleq$

$\exists\, msg \in sentMsg[S]:$

$\land\ msg.type =$ "REPLY"

$\land\ cCrtOp[S][c] \neq \langle\rangle$

$\land\ msg.id = [cid \mapsto c,\ msgid \mapsto cMsgCounter[S][c]]$ reply to c's request for crt op

$TODO$: if already reply from src, keep the most recent one (biggest view Number)

$\land\ Assert(Cardinality(cCrtOpReplies[c]) < 10,$ "$cCrtOpReplies$ $cardinality$ $bound$")

$\land\ \lor\ \land\ cCrtOp[S][c].type =$ "Inconsistent"

$\land\ cCrtOpReplies' = [cCrtOpReplies$ EXCEPT $![S][c] = @ \cup$

$\{[viewNumber \mapsto msg.v,$

$src \mapsto msg.src]\}]$

$\lor\ \land\ cCrtOp[S][c].type =$ "Consensus"

$\land\ cCrtOpReplies' = [cCrtOpReplies$ EXCEPT $![S][c] = @ \cup$

$\{[viewNumber \mapsto msg.v,$

$res \mapsto msg.res,$

$src \mapsto msg.src]\}]$

$\land$ UNCHANGED $\langle cCrtOp,\ cCrtOpToFinalize,\ cCrtOpConfirms,$

$cMsgCounter,\ cState,\ rVars,\ aVars,\ oVars\rangle$

"Helper" formulas

$\_\_matchingViewNumbers(Q,\ c) \triangleq$

a (super)quorum of replies with matching view numbers

$\land\ \forall\, r \in Q:$

$\land\ \exists\, reply \in cCrtOpReplies[S][c]:\ reply.src = r$

$\land\ \forall\, p \in Q: \exists\, rr,\ pr \in cCrtOpReplies[S][c]:$

$\land\ rr.src = r$

$\land\ pr.src = p$

$$\land\ rr.viewNumber = pr.viewNumber$$

$\_\_matchingViewNumbersAndResults(Q,\ c) \triangleq$

a (super)quorum of replies with matching view numbers

and results

$\land \forall\ r \in Q :$

$\land \exists\ reply \in cCrtOpReplies[S][c]: reply.src = r$

$\land \forall\ p \in Q : \exists\ rr,\ pr \in cCrtOpReplies[S][c] :$

$\land\ rr.src = r$

$\land\ pr.src = p$

$\land\ rr.viewNumber = pr.viewNumber$

$\land\ rr.res = pr.res$

IR Client received enough responses to decide

what to do with the operation

$ClientDecideOp(c) \triangleq$

$\land\ cCrtOp[S][c] \neq \langle\rangle$

I. The *IR* Client got a simple quorum of replies

$\land\ \lor \exists\ Q \in Quorums :$

$\land \forall\ r \in Q :$

$\exists\ reply \in cCrtOpReplies[S][c] : reply.src = r$

$\land\ \lor\ \land\ cCrtOp[S][c].type =$ "Inconsistent"

$\land\ \_\_matchingViewNumbers(Q,\ c)$

$\land\ aSuccessful' = aSuccessful \cup$

$\{[mid \mapsto [cid \mapsto c,$

$msgid \mapsto cMsgCounter[S][c]],$

$op \mapsto cCrtOp[S][c]]\}$

$\land\ SuccessfulInconsistentOp(cCrtOp[S][c])$

$\land\ Send([type \mapsto \text{“FINALIZE”},$
$\qquad id \mapsto [cid \mapsto c,\ msgid \mapsto cMsgCounter[S][c]],$
$\qquad op \mapsto cCrtOp[S][c]])$
$\land$ UNCHANGED $\langle cCrtOpToFinalize \rangle$

$\lor\ \land\ cCrtOp[S][c].type = \text{“Consensus”}$
$\land$ LET $res \triangleq$ IF $\_\_matchingViewNumbersAndResults(Q,\ c)$
THEN
CHOOSE $result \in$
$\{res \in Results :$
$\exists\, reply \in cCrtOpReplies[S][c] :$
$\land\ reply.src \in Q$
$\land\ reply.res = res\}$: TRUE
ELSE
$Decide(cCrtOpReplies[S][c])$
IN
$\land\ Send([type \mapsto \text{“FINALIZE”},$
$\qquad id \mapsto [cid \mapsto c,\ msgid \mapsto cMsgCounter[S][c]],$
$\qquad op \mapsto cCrtOp[S][c],$
$\qquad res \mapsto res])$
$\land\ cCrtOpToFinalize' = [cCrtOp$ EXCEPT $![S][c] = cCrtOp[S][c]]$
$\land$ UNCHANGED $\langle aSuccessful \rangle$

II. The *IR* Client got super quorum of responses

$\lor \exists\, SQ \in SuperQuorums :$
$\land \forall\, r \in SQ :$
$\exists\, reply \in cCrtOpReplies[S][c] : reply.src = r$
$\land\ cCrtOp[S][c].type = \text{“Consensus”}$ only care if consensus *op*
$\land\ \_\_matchingViewNumbersAndResults(SQ,\ c)$

$\land$ LET $res \triangleq$ CHOOSE $result \in$

$\{res \in Results :$

$\exists\, reply \in cCrtOpReplies[S][c] :$

$\land\ reply.src \in SQ$

$\land\ reply.res = res\}$: TRUE

IN

$\land\ Send([type \mapsto$ "FINALIZE",

$id \mapsto [cid \mapsto c,\ msgid \mapsto cMsgCounter[S][c]],$

$op \mapsto cCrtOp[S][c],$

$res \mapsto res])$

$\land\ aSuccessful' = aSuccessful \cup$

$\{[mid \mapsto [cid \mapsto c,$

$msgid \mapsto cMsgCounter[S][c]],$

$op \mapsto cCrtOp[S][c],$

$res \mapsto res]\}$

$\land\ SuccessfulConsensusOp(cCrtOp[S][c],\ res)$

$\land$ UNCHANGED $\langle cCrtOpToFinalize \rangle$

$\land\ cCrtOp' = [cCrtOp$ EXCEPT $![S][c] = \langle\rangle]$

$\land\ cCrtOpReplies' = [cCrtOpReplies$ EXCEPT $![S][c] = \{\}]$

$\land$ UNCHANGED $\langle cMsgCounter,\ cState,\ cCrtOpConfirms,\ rVars,\ gViewChangesNo \rangle$

Client received a confirm

$ClientReceiveConfirm(c) \triangleq$

$\exists\, msg \in sentMsg[S] :$

$\land\ msg.type =$ "CONFIRM"

$\land\ cCrtOpToFinalize[S][c] \neq \langle\rangle$

$\land\ msg.id = [cid \mapsto c,\ msgid \mapsto cMsgCounter[S][c]]$ reply to c's request for crt op

$\land\ cCrtOpConfirms' = [cCrtOpConfirms$ EXCEPT $![S][c] = @ \cup$

$$\{[viewNumber \mapsto msg.v,$$
$$res \mapsto msg.res,$$
$$src \mapsto msg.src]\}]$$

$\wedge$ UNCHANGED $\langle cCrtOp, cCrtOpReplies, cCrtOpToFinalize, cMsgCounter, cState, rVars, aVars, oVars\rangle$

An operation is finalized by a client and result returned to the application

$ClientFinalizedOp(c) \triangleq$

$\wedge\ cCrtOpToFinalize[S][c] \neq \langle\rangle$

$\wedge\ \exists\, Q \in Quorums :$

IR client received a quorum of responses

$\wedge\ \forall\, r \in Q :$

$\exists\, reply \in cCrtOpConfirms[S][c] : reply.src = r$

$\wedge$ LET

take the result in the biggest view number

$reply \triangleq$ CHOOSE $reply \in cCrtOpConfirms[S][c] :$
$\neg\exists\, rep \in cCrtOpConfirms[S][c] :$
$rep.viewNumber > reply.viewNumber$

IN

$\wedge\ aSuccessful' = aSuccessful \cup$
$\{[mid \mapsto [cid \mapsto c,$
$msgid \mapsto cMsgCounter[S][c]],$
$op \mapsto cCrtOpToFinalize[S][c],$
$res \mapsto reply.res]\}$

$\wedge\ SuccessfulConsensusOp(cCrtOp[S][c], reply.res)$ respond to app

$\wedge\ cCrtOpToFinalize' = [cCrtOpToFinalize$ EXCEPT $![S][c] = \langle\rangle]$

$\wedge\ cCrtOpConfirms' = [cCrtOpConfirms$ EXCEPT $![S][c] = \{\}]$

$\wedge$ UNCHANGED $\langle rVars, cCrtOp, cCrtOpReplies, cMsgCounter, cState, oVars\rangle$

Client fails and looses all data

$ClientFail(c) \triangleq$

$\land\ cState' = [cState \text{ EXCEPT } ![S][c] = \text{“FAILED”}]$

$\land\ cMsgCounter' = [cMsgCounter \text{ EXCEPT } ![S][c] = 0]$

$\land\ cCrtOp' = [cCrtOp \text{ EXCEPT } ![S][c] = \langle\rangle]$

$\land\ cCrtOpReplies' = [cCrtOpReplies \text{ EXCEPT } ![S][c] = \{\}]$

$\land\ AppClientFail$

$\land\ \text{UNCHANGED } \langle rVars,\ aVars,\ oVars\rangle$

Client recovers

$ClientRecover(c) \triangleq \text{FALSE}$

Replica Actions

Replica sends a reply

$ReplicaReceiveRequest(r) \triangleq$

$\exists\, msg \in sentMsg[S] :$

$\land\ msg.type = \text{“REQUEST”}$

$\land\ \neg\exists\, rec \in rRecord[S][r] : rec.msgid = msg.id$

not alredy replied for this *op*

$\land\ \lor\ \land\ msg.op.type = \text{“Inconsistent”}$

$\land\ Send([type \mapsto \text{“REPLY”},$

$id \mapsto msg.id,$

$v \mapsto rViewNumber[S][r],$

$src \mapsto r])$

$\land\ rRecord' = [rRecord \text{ EXCEPT } ![S][r] = @ \cup \{[msgid \mapsto msg.id,$

$op \mapsto msg.op,$

$status \mapsto \text{“TENTATIVE”}]\}]$

$\lor \land msg.op.type = \text{“Consensus”}$

$\land \textsc{let}\ res \triangleq ExecConsensus(msg.op)$

$\textsc{in}$

$\land Send([type \mapsto \text{“REPLY”},$
$id \mapsto msg.id,$
$v \mapsto rViewNumber[S][r],$
$res \mapsto res,$
$src \mapsto r])$

$\land rRecord' = [rRecord\ \textsc{except}\ ![S][r] = @ \cup \{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$
$res \mapsto res,$
$status \mapsto \text{“TENTATIVE”}]\}]$

$\land \textsc{unchanged}\ \langle rState, rViewNumber, rViewReplies, cVars, aVars, gViewChangesNo\rangle$

Replica receive a message from an *IR* Client to finalize an *op*

For inconsistent oprations the replica sends $< CONFIRM >$ and

executes the operation.

TODO: Write this more compact

$ReplicaReceiveFinalize(r) \triangleq$

$\exists\, msg \in sentMsg[S] :$

$\land msg.type = \text{“FINALIZE”}$

$\land \lor \land msg.op.type = \text{“Inconsistent”}$

$\land Send([type \mapsto \text{“CONFIRM”},$
$v \mapsto rViewNumber[S][r],$
$id \mapsto msg.id,$
$op \mapsto msg.op,$
$src \mapsto r])$

$\land \lor \exists\, rec \in rRecord[S][r] :$

$\land\ rec.msgid = msg.id$

$\land\ rec.op = msg.op$ Replica knows of this op

$\land$ IF $rec.status \neq$ "FINALIZED"

THEN $ExecInconsistent(msg.op)$

ELSE TRUE

$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = (@ \setminus \{rec\}) \cup$
$\{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$
$status \mapsto$ "FINALIZED"$]\}]$

$\lor\ \land \neg\exists\, rec \in rRecord[S][r]:$ Replica didn't hear of this op

$\land\ rec.msgid = msg.id$

$\land\ rec.op = msg.op$

$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = @ \cup$
$\{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$
$status \mapsto$ "FINALIZED"$]\}]$

$\land\ ExecInconsistent(msg.op)$

$\lor\ \land\ msg.op.type =$ "Consensus"

$\land\ \lor\ \land \exists\, rec \in rRecord[S][r]:$

$\land\ rec.msgid = msg.id$

$\land\ rec.op = msg.op$ Replica knows of this op

$\land\ \lor\ \land\ rec.status =$ "TENTATIVE" Operation tentative

$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = (@ \setminus \{rec\}) \cup$
$\{[msgid \mapsto msg.id,$
$op \mapsto msg.op,$
$res \mapsto msg.res,$
$status \mapsto$ "FINALIZED"$]\}]$

$\land\ Send([type \mapsto$ "CONFIRM",

$$
\begin{array}{l}
\qquad\qquad\quad v \;\mapsto rViewNumber[S][r], \\
\qquad\qquad\quad id \;\mapsto msg.id, \\
\qquad\qquad\quad op \;\mapsto msg.op, \\
\qquad\qquad\quad res \;\mapsto msg.res, \\
\qquad\qquad\quad src \;\mapsto r])
\end{array}
$$

$\wedge$ IF $rec.res \neq msg.res$

THEN $UpdateConsensus(msg.op, msg.res)$

ELSE TRUE

$\vee \wedge rec.status =$ "FINALIZED" Operation already finalized (view change happened in the meantime)

$\wedge Send([type \mapsto$ "CONFIRM",

$$
\begin{array}{l}
\qquad\quad v \;\mapsto rViewNumber[S][r], \\
\qquad\quad id \;\mapsto msg.id, \\
\qquad\quad op \;\mapsto msg.op, \\
\qquad\quad res \;\mapsto rec.res, \\
\qquad\quad src \;\mapsto r])
\end{array}
$$

$\wedge$ UNCHANGED $\langle rRecord \rangle$

$\vee \wedge \neg\exists\, rec \in rRecord[S][r]:$ Replica didn't hear of this op

$\wedge rec.msgid = msg.id$

$\wedge rec.op = msg.op$

$\wedge rRecord' = [rRecord$ EXCEPT $![S][r] = @ \cup$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\quad \{[msgid \mapsto msg.id, \\
\qquad\qquad\qquad\qquad\qquad\qquad op \mapsto msg.op, \\
\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto msg.res, \\
\qquad\qquad\qquad\qquad\qquad\qquad status \mapsto \text{"FINALIZED"}]\}]
\end{array}
$$

$\wedge Send([type \mapsto$ "CONFIRM",

$$
\begin{array}{l}
\qquad\quad v \;\mapsto rViewNumber[S][r], \\
\qquad\quad id \;\mapsto msg.id, \\
\qquad\quad op \;\mapsto msg.op,
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad res \;\mapsto msg.res,\\
&\qquad\qquad src \;\mapsto r])\\
&\quad \land \mathit{ExecuteAndUpdateConsensus}(msg.op,\ msg.res)\\
&\land \textsc{unchanged}\ \langle rState,\ rViewNumber,\ rViewReplies,\ cVars,\ aVars,\ gViewChangesNo\rangle
\end{aligned}
$$

A replica starts the view change procedure

supports concurrent view changes (id by src)

$$
\begin{aligned}
&ReplicaStartViewChange(r) \triangleq\\
&\land Send([type \mapsto \text{“START-VIEW-CHANGE”},\\
&\qquad v \;\mapsto rViewNumber[r],\\
&\qquad src \;\mapsto r])\\
&\land rState' = [rState \ \textsc{except}\ ![r] = \text{“RECOVERING”}]\\
&\land \textsc{unchanged}\ \langle rViewNumber,\ rViewReplies,\ rRecord,\ cVars,\ aVars\rangle\\
&\land gViewChangesNo < max\_vc \quad \text{BOUND on number of view changes}\\
&\land gViewChangesNo' = gViewChangesNo + 1
\end{aligned}
$$

A replica received a message to start view change

$$
\begin{aligned}
&ReplicaReceiveStartViewChange(r) \triangleq\\
&\land \exists\, msg \in sentMsg[S]:\\
&\quad \land msg.type = \text{“START-VIEW-CHANGE”}\\
&\quad \land \textsc{let}\ v\_new \triangleq\\
&\qquad\qquad \textsc{if}\ msg.v > rViewNumber[r]\ \textsc{then}\ msg.v\\
&\qquad\qquad \textsc{else}\ \ rViewNumber[S][r]\\
&\qquad \textsc{in}\\
&\qquad \land \neg\exists\, m \in sentMsg[S]: \quad \text{not already sent (just to bound the model checker)}\\
&\qquad\quad \land m.type = \text{“DO-VIEW-CHANGE”}\\
&\qquad\quad \land m.v \geq msg.v\\
&\qquad\quad \land m.dst = msg.src\\
&\qquad\quad \land m.src = r
\end{aligned}
$$

$\land\ Send([type \mapsto$ "DO-VIEW-CHANGE",
$v \mapsto v\_new + 1,$
$r \mapsto rRecord[r],$
$src \mapsto r,$
$dst \mapsto msg.src])$
$\land\ rViewNumber' = [rViewNumber$ EXCEPT $![S][r] = v\_new + 1]$
$\land\ rState' = [rState$ EXCEPT $![S][r] =$ "VIEW-CHANGING"$]$
$\land$ UNCHANGED $\langle cVars, rRecord, rViewReplies, aVars, gViewChangesNo\rangle$

Replica received DO-VIEW-CHANGE message

$ReplicaReceiveDoViewChange(r) \triangleq$
$\land \exists\, msg \in sentMsg[S]:$
$\land\ msg.type =$ "DO-VIEW-CHANGE"
$\land\ msg.dst = r$
$\land\ msg.v > rViewNumber[r]$
$\land\ rViewReplies' = [rViewReplies$ EXCEPT $![r] = @ \cup$
$\{[type \mapsto$ "do-view-change",
$viewNumber \mapsto msg.v,$
$r \mapsto msg.r,$
$src \mapsto msg.src]\}]$
$\land$ UNCHANGED $\langle cVars, rViewNumber, rRecord, rState, aVars, oVars\rangle$

A replica received enough view change replies to start processing in the new view

$ReplicaDecideNewView(r) \triangleq$
$\land \exists\, Q \in Quorums:$
$\land \forall\, rep \in Q : \exists\, reply \in rViewReplies[r]:\ \land\ reply.src = rep$
$\land\ reply.type =$ "do-view-change"

received at least a quorum of replies

$\land$ LET $recoveredConensusOps\_a \triangleq$

```
        any consensus operation found in at least a majority of a Quorum
        {x ∈ UNION {y.r : y ∈ {z ∈ rViewReplies[S][r] : z.src ∈ Q}} :
          ∧ x[2].type = "Consensus"
          ∧ ∃ P ∈ SuperQuorums :
              ∀ rep ∈ Q ∩ P       :
                ∃ reply ∈ rViewReplies[r] :
                  ∧ reply.src = rep
                  ∧ x ∈ reply.r}  same op, same result

      recoveredConensusOps_b ≜   TODO: what result? from the app?
        the rest of consensus ops found in at least one record (discard the result)
        {⟨z[1], z[2]⟩ :
          z ∈ {x ∈ UNION {y.r : y ∈ {z ∈ rViewReplies[S][r] : z.src ∈ Q}} :
            ∧ x[2].type = "Consensus"
            ∧ ¬x ∈ recoveredConensusOps_a}}

      recoveredInconsistentOps_c ≜
        any inconsistent operation found in any received record (discard the result)
        {⟨z[1], z[2]⟩ :
          z ∈ {x ∈ UNION {y.r : y ∈ {z ∈ rViewReplies[S][r] : z.src ∈ Q}} :
            x[2].type = "Inconsistent"}}
    IN
      ∧ AppRecoverOpsResults(recoveredConensusOps_a)
      ∧ AppRecoverOps(recoveredConensusOps_b)
      ∧ AppRecoverOps(recoveredInconsistentOps_c)
      ∧ rRecord' = [rRecord EXCEPT ![S][r] = @ ∪ recoveredConensusOps_a
                                               ∪ recoveredConensusOps_b
                                               ∪ recoveredInconsistentOps_c]
  ∧ LET v_new ≜
```

max view number received

CHOOSE $v \in \{x.viewNumber : x \in rViewReplies[r]\}$:

$\forall\, y \in rViewReplies[r]$:

$y.viewNumber \leq v$

IN

$\land\, Send([type \mapsto$ "START-VIEW",

$v \mapsto v\_new,$

$src \mapsto r])$

$\land\, rViewNumber' = [rViewNumber$ EXCEPT $![r] = v\_new]$

$\land\, rViewReplies' = [rViewReplies$ EXCEPT $![r] = \{\}]$

$\land$ UNCHANGED $\langle rState, cVars, aVars, gViewChangesNo\rangle$

A replica receives a start view message

$ReplicaReceiveStartView(r) \triangleq$

$\land\, \exists\, msg \in sentMsg$:

$\land\, msg.type =$ "START-VIEW"

$\land\, msg.v \geq rViewNumber[r]$

$\land\, msg.src \neq r$ don't reply to myself

$\land\, Send([type \mapsto$ "START-VIEW-REPLY",

$v \mapsto msg.v,$

$src \mapsto r,$

$dst \mapsto msg.src])$

$\land\, rViewNumber' = [rViewNumber$ EXCEPT $![r] = msg.v]$

$\land\, rState' = [rState$ EXCEPT $![r] =$ "NORMAL"$]$

$\land$ UNCHANGED $\langle rRecord, rViewReplies, cVars, aVars, gViewChangesNo\rangle$

$ReplicaReceiveStartViewReply(r) \triangleq$

$\land\, \exists\, msg \in sentMsg$:

$\land\, msg.type =$ "START-VIEW-REPLY"

$\land\ msg.dst = r$

$\land\ msg.v > rViewNumber[r]$ receive only if bigger than the last view I was in

$\land\ rViewReplies' = [rViewReplies$ EXCEPT $![S][r] = @ \cup$

$\{[type \mapsto$ "start-view-reply",

$viewNumber \mapsto msg.v,$

$r \mapsto \{\},$

$src \mapsto msg.src]\}]$

$\land$ UNCHANGED $\langle rRecord, rState, rViewNumber, cVars, aVars, oVars\rangle$

$ReplicaRecover(r) \triangleq$ we received enough START-VIEW-$REPLY$ messages

$\exists\ Q \in Quorums :$

$\land\ r \in Q$

$\land\ \forall\ p \in Q : \lor\ p = r$

$\lor\ \land\ p \neq r$

$\land\ \exists\ reply \in rViewReplies[S][r] : \land\ reply.src = p$

$\land\ reply.type =$ "start-view-reply"

$\land\ rViewReplies' = [rViewReplies$ EXCEPT $![S][r] = \{\}]$

$\land\ rState' = [rState$ EXCEPT $![r] =$ "NORMAL"$]$

$\land$ UNCHANGED $\langle rRecord, rViewNumber, cVars, aVars, oVars\rangle$

$ReplicaResumeViewChange(r) \triangleq$ $TODO$: On timeout

FALSE

A replica fails and looses everything

$ReplicaFail(r) \triangleq$ $TODO$: check cardinality

$\land\ rState' = [rState$ EXCEPT $![S][r] =$ "FAILED"$]$

$\land\ rRecord' = [rRecord$ EXCEPT $![S][r] = \{\}]$

$\land\ rViewNumber' = [rViewNumber$ EXCEPT $![r] = 0]$ * $TODO$: check what happens if we loose the view number

$\land\ rViewReplies' = [rViewReplies$ EXCEPT $![S][r] = \{\}]$

$\land$ UNCHANGED $\langle rViewNumber, cVars, aVars, oVars\rangle$

$\land\ Cardinality(\{re \in Replicas :$

We assume less than f replicas are allowed to fail

$\lor\ rState[S][re] = \text{“FAILED”}$

$\lor\ rState[S][re] = \text{“RECOVERING”}\}) < f$

High-Level Actions

$ClientAction(c) \triangleq$

$\lor\ \land\ cState[c] = \text{“NORMAL”}$

$\land\ \ \lor\ ClientRequest(c)$ * some client tries to replicate commit an operation

$\lor\ ClientReceiveReply(c)$ some client receives a reply from a replica

$\lor\ ClientReceiveConfirm(c)$ some client receives a confirm from a replica

$\lor\ ClientFail(c)$ * some client fails

$\lor\ ClientDecideOp(c)$ an operation is successful at some client

$\lor\ ClientFinalizedOp(c)$ * an operation was finalized at some client

$\lor\ \land\ cState[c] = \text{“FAILED”}$

$\land\ \lor\ ClientRecover(c)$

$ReplicaAction(r) \triangleq$

$\lor\ \land\ rState[S][r] = \text{“NORMAL”}$

$\land\ \lor\ ReplicaReceiveRequest(r)$ some replica sends a reply to a $REQUEST\ msg$

$\lor\ ReplicaReceiveFinalize(r)$

$\lor\ ReplicaReceiveStartViewChange(r)$

$\lor\ ReplicaReceiveStartView(r)$

$\lor\ ReplicaFail(r)$ * some replica fails

$\lor\ \land\ rState[S][r] = \text{“FAILED”}$

$\land\ \lor\ ReplicaStartViewChange(r)$ * some replica starts to recover

$\lor\ \land\ rState[r] = \text{“RECOVERING”}$ * just to make it clear

$\land \lor ReplicaReceiveDoViewChange(r)$

$\lor ReplicaDecideNewView(r)$

$\lor ReplicaReceiveStartViewReply(r)$

$\lor ReplicaRecover(r)$

$\lor \land rState[S][r] = \text{“VIEW-CHANGING”}$

$\land \lor ReplicaReceiveStartViewChange(r)$

$\lor ReplicaReceiveStartView(r)$

$\lor ReplicaResumeViewChange(r)$ * some timeout expired and view change not finished

$\lor ReplicaFail(r)$

$Next \triangleq$

$\lor \exists\, c \in Clients : ClientAction(c)$

$\lor \exists\, r \in Replicas : ReplicaAction(r)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$FaultTolerance \triangleq$

$\land \forall\, successfulOp \in aSuccessful,\ Q \in Quorums :$

$(\forall\, r \in Q : rState[S][r] = \text{“NORMAL”} \lor rState[S][r] = \text{“VIEW-CHANGING”})$

$\Longrightarrow (\exists\, p \in Q : \exists\, rec \in rRecord[S][p] :$

$\land successfulOp.msgid = rec.msgid$

$\land successfulOp.op = rec.op)$ Not necessarily same result

$\land \forall\, finalizedOp \in aSuccessful,\ Q \in Quorums :$

$(\forall\, r \in Q : rState[r] = \text{“NORMAL”} \lor rState[r] = \text{“VIEW-CHANGING”})$

$\Longrightarrow (\exists\, P \in SuperQuorums :$

$\forall\, p \in Q \cap P :$

$\exists\, rec \in rRecord[p] :$

$finalizedOp = rec)$

$Inv \triangleq TypeOK \land FaultTolerance$

B.2 TAPIR Specification

MODULE $TAPIR$

This is a TLA+ specification of the $TAPIR$ algorithm.

EXTENDS $FiniteSets$, $Naturals$, TLC, $TLAPS$

$Max(S) \triangleq$ IF $S = \{\}$ THEN 0 ELSE CHOOSE $i \in S : \forall\, j \in S : j \leq i$

$TAPIR$ constants:

1. *Shards*: function from shard id to set of replica ids in the shard
2. Transactions: set of all possible transactions
3. *nr_shards*: number of *shards*

CONSTANTS $Shards$, $Transactions$, $NrShards$

Note: assume unique number ids for replicas

IR constants & variables (description in the IR module)

CONSTANTS $Clients$, $Quorums$, $SuperQuorums$,
max_vc, max_req, f

VARIABLES $rState$, $rRecord$, $rViewNumber$, $rViewReplies$, $sentMsg$, $cCrtOp$,
$cCrtOpToFinalize$, $cMsgCounter$, $cCrtOpReplies$, $cCrtOpConfirms$,
$cState$, $aSuccessful$, $gViewChangesNo$

$irReplicaVars \triangleq \langle rState, rRecord, rViewNumber, rViewReplies \rangle$

$irClientVars \triangleq \langle cCrtOp,$ current operation at a client
$cCrtOpReplies,$ current operation replies
$cMsgCounter,$
$cState,$
$cCrtOpToFinalize,$

$cCrtOpConfirms\rangle$ Client variables.

$irAppVars \triangleq \langle aSuccessful\rangle$ Application variables

$irOtherVars \triangleq \langle sentMsg, gViewChangesNo\rangle$ Other variables.

TAPIR Variables/State: 1. State at each replica:

$rPrepareTxns$ = List of txns this replica is prepared to commit

$rTxnsLog$ = *Log* of committed and aborted txns in ts order $rStore$ = *Versioned* store $rBkpTable$ = Table of txns for which this replica

is the bkp coordinator

2. State of communication medium: $sentMsg$ = sent (and duplicate) messages

3. State at client: $cCrtTxn$ = *crt txn* requested by the client

TAPIR variables & data structures

VARIABLES $rPreparedTxns$, $rStore$, $rTxnsLogAborted$, $rTxnsLogCommited$,
$rClock$, $cCrtTxn$, $cClock$

$tapirReplicaVars \triangleq \langle rPreparedTxns, rStore, rTxnsLogAborted, rTxnsLogCommited,$
$rClock\rangle$

$tapirClientVars \triangleq \langle cCrtTxn, cClock\rangle$

$StoreEntry \triangleq [vs : Nat, val : Nat]$ vs = version

$Store \triangleq [key : Nat,$
$entries : \text{SUBSET } StoreEntry,$
$latestVs : Nat,$
$latestVal : Nat]$

$TransactionTs \triangleq [cid : Clients, clock : Nat]$ Timestamp

$ReadSet \triangleq [key : Nat, val : Nat, vs : Nat]$

$WriteSet \triangleq [key : Nat, val : Nat]$

$Transaction \triangleq [rSet : \text{SUBSET } ReadSet,$
$wSet : \text{SUBSET } WriteSet,$
$shards : \text{SUBSET } Nat]$

$TypeOK \triangleq$

$\land \; rStore \in [\textsc{union}\ \{Shards[i] : i \in 1\,..\,NrShards\} \to \textsc{subset}\ Store]$

$\land \; rPreparedTxns \in [\textsc{union}\ \{Shards[i] : i \in 1\,..\,NrShards\} \to \textsc{subset}\ Transaction]$

$\land \; rTxnsLogAborted \in [\textsc{union}\ \{Shards[i] : i \in 1\,..\,NrShards\} \to \textsc{subset}\ Transaction]$

$\land \; rTxnsLogCommited \in [\textsc{union}\ \{Shards[i] : i \in 1\,..\,NrShards\} \to \textsc{subset}\ Transaction]$

$TAPIRResults \triangleq \{$"Prepare-OK", "Retry", "Prepare-Abstain", "Abort"$\}$

$TAPIROpType \triangleq \{$"Prepare", "ABORT", "COMMIT"$\}$

$TAPIROpBody \triangleq [opType : TAPIROpType,\ txn : Transaction]$

$TAPIRClientFail \triangleq$ TRUE state we lose at the app level

$TAPIRReplicaFail \triangleq$ TRUE state we lose at the app level

TAPIR implementation of *IR* interface

$TAPIRExecInconsistent(op) \triangleq$ TRUE

$TAPIRExecConsensus(op) \triangleq$ IF $op.type =$ "Consensus" THEN "Prepare-OK" ELSE "Abort"

$TAPIRDecide(results) \triangleq$ TRUE

$TAPIRMerge(d,\ u) \triangleq$ TRUE

$TAPIRSync(records) \triangleq$ TRUE

$TAPIRSuccessfulInconsistentOp(op) \triangleq$ TRUE

$TAPIRSuccessfulConsensusOp(op,\ res) \triangleq$ TRUE

Initialize for all *shards*

$InitIR \triangleq$

$\land\ rState = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto$ "NORMAL"$]]$

$\land\ rRecord = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto \{\}]]$

$\land\ rViewNumber = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto 0]]$

$\land\ rViewReplies = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto \{\}]]$

$\land\ sentMsg = [s \in 1\,..\,NrShards \mapsto \{\}]$

$\land\ cCrtOp = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \langle\rangle]]$

$\land\ cCrtOpToFinalize = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \langle\rangle]]$

$\land\ cMsgCounter = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto 0]]$

$\land\ cCrtOpReplies = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \{\}]]$

$\land\ cCrtOpConfirms = [s \in 1\,..\,NrShards \mapsto [c \in Clients \mapsto \{\}]]$

$\land\ cState = [c \in Clients \mapsto \text{“NORMAL”}]$

$\land\ aSuccessful = \{\}$

$\land\ gViewChangesNo = [s \in 1\,..\,NrShards \mapsto 0]$

IR instance per shard *TODO*: modify replica also

$IR(s) \triangleq \textsc{instance}\ IR\_consensus\ \textsc{with}\ AppClientFail \leftarrow TAPIRClientFail,$
$AppReplicaFail \leftarrow TAPIRReplicaFail,$
$OpBody \leftarrow TAPIROpBody,$
$ExecInconsistent \leftarrow TAPIRExecInconsistent,$
$ExecConsensus \leftarrow TAPIRExecConsensus,$
$Merge \leftarrow TAPIRMerge,$
$Sync \leftarrow TAPIRSync,$
$SuccessfulInconsistentOp \leftarrow TAPIRSuccessfulInconsistentOp,$
$SuccessfulConsensusOp \leftarrow TAPIRSuccessfulConsensusOp,$
$Decide \leftarrow TAPIRDecide,$
$Results \leftarrow TAPIRResults,$
$Replicas \leftarrow Shards[s],$
$Quorums \leftarrow Quorums[s],$
$SuperQuorums \leftarrow SuperQuorums[s],$
$S \leftarrow s$

TAPIR messages

$Message \triangleq$

$[type : \{\text{“READ”}\},$
$\ key : Nat,$
$\ dst \ : \textsc{union}\ Shards]$

$\cup$

$[type : \{\text{“READ-REPLY”}\},$
$\ key : Nat,$
$\ val \ : Nat,$
$\ vs : Nat,$ version
$\ dst : Clients]$

$\cup$

$[type : \{\text{“READ-VERSION”}\},$
$\ key : Nat,$
$\ vs : Nat,$
$\ dst : \textsc{union}\ Shards]$

$\cup$

$[type : \{\text{“READ-VERSION-REPLY”}\},$
$\ key : Nat,$
$\ vs : Nat,$
$\ dst : Clients]$

$$
\begin{aligned}
InitTAPIR \triangleq\ & \land\ cCrtTxn = [c \in Clients \mapsto \langle\rangle] \\
& \land\ cClock = [c \in Clients \mapsto 0] \\
& \land\ rPreparedTxns = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto \{\}]] \\
& \land\ rStore = [r \in \textsc{union}\ \{Shards[i] : i \in 1\,..\,NrShards\} \mapsto \{\}] \\
& \land\ rTxnsLogAborted = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto \{\}]] \\
& \land\ rClock = [s \in 1\,..\,NrShards \mapsto [r \in Shards[s] \mapsto 0]]
\end{aligned}
$$

$$Init \triangleq InitIR \land InitTAPIR$$

Tapir replica actions

$TAPIRReplicaReceiveRead(r) \triangleq$ TRUE

$TAPIRReplicaAction(r) \triangleq$

$\lor \land rState[r] =$ "NORMAL"

$\land \lor TAPIRReplicaReceiveRead(r)$

Tapir client actions

$TAPIRClientExecuteTxn(c) \triangleq$

first, resolve all reads (read from any replica and get the vs)

then send prepares in all shard involved by seting the $cCrtOp$ in the

respective IR shard instance

$TODO$: for now just simulate this, pick a transaction from

transaction pool, get some versions from the replica

stores

$\land cCrtTxn[c] = \langle\rangle$

$\land \exists\, t \in Transactions :$

LET $rSet \triangleq \{rse \in ReadSet :$

$\land \exists\, trse \in t.rSet : rse = trse$

$\land$ LET

$r \triangleq Max(\{r \in Shards[(rse.key \% NrShards) + 1] :$

$\exists\, se \in rStore[r] : rse.key = se.key\})$

IN

$\land r \neq 0$

$\land \exists\, se \in rStore[r] :$

$$
\begin{aligned}
&\wedge\ rse.key = se.key \\
&\wedge\ rse.val = se.latestVal \\
&\wedge\ rse.vs = se.latestVs \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
shards \triangleq\ &\{s \in 1\,..\,NrShards : \\
&\quad \vee \exists\, trse \in t.rSet : s = (trse.key\%NrShards) + 1 \\
&\quad \vee \exists\, twse \in t.wSet : s = (twse.key\%NrShards) + 1\}
\end{aligned}
$$

IN

$\wedge\ Cardinality(rSet) = Cardinality(t.rSet)$ found all the reads

$\wedge\ cCrtTxn' = [cCrtTxn \text{ EXCEPT } ![c] = [rSet \mapsto rSet,$
$wSet \mapsto t.wSet,$
$shards \mapsto shards]]$

$\wedge$ UNCHANGED $\langle irReplicaVars,\ irClientVars,\ irOtherVars,\ irAppVars,$
$tapirReplicaVars,\ cClock\rangle$

$TAPIRClientPrepareTxn(c) \triangleq$

$\wedge\ cCrtTxn[c] \neq \langle\rangle$

$\wedge\ \exists\, s \in cCrtTxn[c].shards :$ prepare in shard s

- ok if already prepared

$\wedge\ IR(s)!ClientRequest(c, [type \mapsto$ "Consensus",
$body \mapsto \langle$"Prepare", $cCrtTxn\rangle])$

$\wedge$ UNCHANGED $\langle irReplicaVars,\ irAppVars,$
$cCrtOpReplies,$
$cCrtOpConfirms,$
$cCrtOpToFinalize,$
$gViewChangesNo,$
$cState,\ tapirClientVars,\ tapirReplicaVars\rangle$

$TAPIRClientAction(c) \triangleq$

$\lor \land cState[c] = \text{“NORMAL”}$

$\land \lor TAPIRClientExecuteTxn(c)$ for now just simulate this

(don't send explicit $READ$ messages)

$\lor TAPIRClientPrepareTxn(c)$

$\lor 2PC(c)$

High-Level Actions

$Next \triangleq$

$\land \lor \exists\, c \in Clients : TAPIRClientAction(c)$

$\lor \land \exists\, s \in 1 \,..\, NrShards : IR(s)!Next$

$\land \text{UNCHANGED } \langle tapirClientVars,\ tapirReplicaVars \rangle$

$Inv \triangleq Cardinality(aSuccessful) < 2$