ABSTRACT

Title of dissertation:	DECOUPLING CONSISTENCY DETERMINATION AND TRUST FROM THE UNDERLYING DISTRIBUTED DATA STORES
	Vasileios Lekakis, Doctor of Philosophy, 2018
Dissertation directed by:	Professor Pete Keleher Department of Computer Science

Building applications on cloud services is cost-effective and allows for rapid development and release cycles. However, relying on cloud services can severely limit applications' ability to control their own consistency policies, and their ability to control data visibility during replication.

To understand the tension between strong consistency and security guarantees on one hand and high availability, flexible replication, and performance on the other, it helps to consider two questions. First, is it possible for an application to achieve stricter consistency guarantees than what the cloud provider offers? If we solely rely on the provider service interface, the answer is no. However, if we allow the applications to determine the implementation and the execution of the consistency protocols, then we can achieve much more.

The second question is, can an application relay updates over untrusted replicas without revealing sensitive information while maintaining the desired consistency guarantees? Simply encrypting the data is not enough. Encryption does not eliminate information leakage that comes from the meta-data needed for the execution of any consistency protocol.

The alternative to encryption—allowing the flow of updates only through trusted replicas—leads to predefined communication patterns. This approach is prone to failures that can cause partitioning in the system. One way to answer "yes" to this question is to allow trust relationships, defined at the application level, to guide the synchronization protocol.

My goal in this thesis is to build systems that take advantage of the performance, scalability, and availability of the cloud storage services while, at the same time, bypassing the limitations imposed by cloud service providers' design choices. The key to achieving this is pushing application-specific decisions where they belong: the application.

I defend the following thesis statement: *By decoupling consistency determination and trust from the underlying distributed data store, it is possible to (1) support application-specific consistency guarantees; (2) allow for topology independent replication protocols that do not compromise application privacy.*

First I design and implement `Shell`, a system architecture for supporting strict consistency guarantees over eventually consistent data stores. `Shell` is a software layer designed to isolate consistency implementations and cloud-provider APIs from the application code. `Shell` consists of four internal modules and an *application store*, which together abstract consistency-related operations and encapsulate communication with the underlying storage layers. Apart from consistency protocols tailored to application needs, `Shell` provides application-aware conflict resolution without relying on generic heuristics such as the "last write wins." `Shell` does not require the application to maintain dependency-tracking information for the execution of the consistency protocols as other existing approaches do. I experimentally evaluate `Shell` over two different data-stores using real-application traces. I found that using `Shell` can reduce the inconsistent updates by 10%. I also measure and show the overheads that come from introducing the `Shell` layer.

Second, I design and implement *T.Rex*, a system for supporting topology-independent replication without the assumption of trust between all the participating replicas. *T.Rex* uses

role-based access control to enable flexible and secure sharing among users with widely varying collaboration types: both users and data items are assigned *roles*, and a user can access data only if it shares at least one role. Building on top of this abstraction, *T.Rex* includes several novel mechanisms: I introduce *role proofs* to prove role membership to others in the role without leaking information to those not in the role. Additionally, I introduce *role coherence* to prevent updates from leaking across roles. Finally, I use Bloom filters as *opaque digests* to enable querying of remote cache state without being able to enumerate it. I combine these mechanisms to develop a novel, cryptographically secure, and efficient anti-entropy protocol, *T.Rex*-Sync. I evaluate *T.Rex* on a local test-bed, and I show that it achieves security with modest computational and storage overheads.

DECOUPLING CONSISTENCY DETERMINATION AND TRUST FROM THE UNDERLYING DISTRIBUTED DATA STORES

by

Vasileios Lekakis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Dedication

To my wife, Jessica. Without her love and support, I certainly could not have completed this thesis.

Acknowledgments

I would like to thank my advisor Pete Keleher and my committee members Udaya Shankar, Yiannis Aloimonos, Mark Shayman, and Neil Spring.

I am grateful to all my friends from the lab: Yunus Basagalar, Matt Lentz, Ramakrishna Padmanabhan, Aaron Schulman, Adam Bender, Randy Baden, Greg Benjamin and Dave Levin.

I am also grateful for my Greek friends: Evripidis Paraskevas, Theo Rekatsinas, and Konstantinos Zampogiannis.

Finally, I am deeply grateful to my wife Jessica for her support and continuous encouragement. And my sweet dog Shaggy for keeping me company all these early mornings the last few years.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The proliferation of mobile devices has resulted in personalized computing ecosystems wherein users use whichever device suits their needs at the moment: a mobile phone on the move, a desktop computer at home, a tablet in front of the TV, and so on. Users expect that their data will be available and consistent across all of their devices. Moreover, as collaborative software like Google Docs [49] has become popular, users have grown to expect the data they share to be available and consistent, even as their collaborators make updates.

Replicated data stores are the canonical approach to achieving consistency across multiple devices. Generally speaking, they function as follows: when a user modifies a data object, the system updates a local version and records the change. This creates *entropy* in the system until other replicas know about this update. Replicas perform anti-entropy [3, 33] sessions either periodically or on-demand. In an anti-entropy session, replicas compare their version vectors, inform one another of whatever updates they are missing, and update their version vectors. Entropy decreases with each anti-entropy session, and as updates spread from user to user, entropy eventually approaches zero.

Replicated data stores fall into two major categories. The first category is Peer-to-Peer (P2P) systems [113] where every user's device is an autonomous replica in the system. To communicate, autonomous replicas can either follow a hierarchical communication pattern or use discovery [43] mechanisms to find "close" replicas int the network. This paradigm operates without a central trusted authority that coordinates replication.

Though P2P systems were very popular, they came with a series of limitations. These limitations include low availability (due to high node churn rates [110]) and unreliable [61, 69] routing techniques [30]. These shortcomings meant that P2P systems could not meet the ever-growing demand for always-available, highly durable, and low-cost storage solutions that accompanied the explosion of the Internet and the World Wide Web (WWW).

Centrally managed cloud-based storage systems—the second category of replicated data stores—have become more attractive. In this paradigm, a cloud provider maintains the storage infrastructure. Replication follows the master-slave paradigm, usually with only best-effort consistency guarantees. The customers interact with the storage layer through a service interface that the cloud provider owns and maintains.

Though there is a large variety [6–9, 12, 48] of available replicated storage services, the cornerstone of most applications today is NoSQL [26, 32] data stores. NoSQL data stores can support high throughput workloads while combining low latency with high scale. Furthermore, unlike the traditional relational databases, they do not impose any schema-constraints on the data they store.

Unlike the traditional full-ACID [53] stores, NoSQL data stores choose availability over strong consistency guarantees. Guaranteeing only eventual consistency allows them to process operations even when some of their internal replicas are unreachable, either because of hardware failures or, more commonly, because of network partitions. This preference for high availability means that applications frequently operate with inconsistent data.

To understand the tension between strong consistency and security guarantees on one hand, and highly flexible, performant, and available replication on the other, it helps to consider two questions. First, is it possible for an application to achieve stricter consistency guarantees than what the cloud provider offers? If we solely rely on the provider service interface, the answer is no. However, building a layer between applications and the cloud allows us to achieve much more.

The second question is, is it possible for an application to relay updates over untrusted replicas without revealing sensitive information, and while maintaining consistency guarantees? Encryption alone is not enough. Encrypting application data does not eliminate information leakage that comes from the meta-data needed for the execution of any consistency protocol. Another alternative — allowing the flow of updates only through trusted replicas — leads to predefined communication patterns. The problem with this approach is prone to failures that can cause partitioning in the system. One way to answer "yes" to this question is to allow trust relationships, defined at the application-level, to guide the synchronization protocol.

1.1 Challenges: Managing Consistency

It is challenging to offer consistency guarantees and a conflict resolution mechanism tailored to application needs when operating over a third-party data store. One obvious reason for this is that application developers have no control over the implementation of the storage layer. Other reasons for this are (1) that cloud providers offer API interfaces with limited expressiveness, and (2) that widely-used programming patterns for accessing the storage layer, such as Data Access Object (DAO), fail to account for eventually consistent data stores, which by definition do not support all of the ACID properties.

Cloud providers, fighting for market-share, create consistency abstractions that need to be broadly applicable. The results are low level API calls which, in consistency terms, translate to simple *READ* and *WRITE* operations. To make matters worse, application developers are forced to use these generic abstractions to build their own consistency solutions. There is no intuitive way to build a consistency protocol, like causal consistency, using a plain *READ* and *WRITE* API interface.

A second challenge is that applications commonly access their data through *data access objects* (DAO) [87] which provide abstract interfaces to databases or other persistence mechanisms. By utilizing the Single Responsibility Principle [98], the pattern separates the persistence layer from the application. The DAO pattern offers an object interface that stays the same despite possible changes in the persistence layer. This pattern also hides complexities in the underlying storage layer from the application. But while DAOs might protect the application from schema and implementation changes, they are meant to be used above storage layers that support all of the ACID properties. This is not a valid assumption for NoSQL data stores. When the assumptions for an abstraction no longer hold, the abstraction will leak complexity [64, 107]. Abstraction leakage exposes the complexity and limitations of the underlying implementation. When applications are backed from a NoSQL data store, eventual consistency seeps into the application code base.

The reality of complex distributed applications is that consistency-related assumptions are scattered throughout the code base. Software evolves rapidly. Different developers, primed to focus on the application functionality, add new features and bug fixes daily. Bugs, with consistency as a cause, manifest themselves as application level errors and thus are treated as such. When a production system is broken, it is difficult for developers to translate from application semantics to consistency semantics, which take place in an infrastructure that the cloud provider maintains and is not accessible to the developer.

The result is a system that is unwieldy to develop, difficult to maintain, and impossible to transition from one provider to another. The creation and maintenance of a system in this environment requires developers to manually translate application semantics to low-level consistency guarantees that are set by someone else—the provider. This task is error-

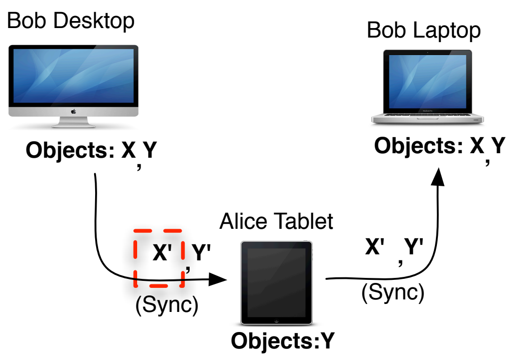

Figure 1.1: A personal sharing system might schedule the desktop's updates to be delivered to the laptop via the tablet. Both the desktop and the laptop access X and Y, while the tablet only accesses (*has rights for*) object Y. The data shown inside a box might be *leaked* because the tablet can see that the desktop just modified X. The tablet may also be able to see the contents of the update.

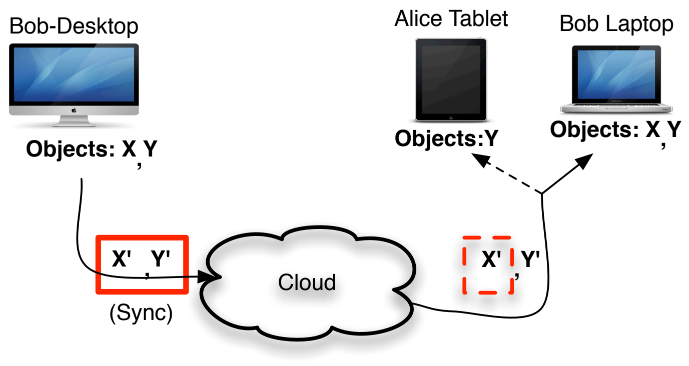

Figure 1.2: Cloud services add cloud replicas to the set of user replicas that might stage data for which they have no access rights. Data, and information about data, might leak to any of the user or cloud replicas.

prone and requires thorough knowledge of both the application and the infrastructure. As a result, many developers rely on adding redundancy to application code, essentially treating application-level symptoms rather than treating their deeper consistency causes. While these solutions are insightful when implemented, taking them out of context makes them hard to understand and even harder to maintain.

1.2 Challenges: Topology-Independent Replication Without Information Leakage

The explosion of mobile devices has created an "anywhere-anytime" mentality in the users; they expect data to follow them wherever they go and to whatever device they use. To achieve this goal, any device should have the ability to communicate with any other device in a P2P system. In the case of a cloud sync-service like Dropbox, devices should also be able to contact any of the provider's meta-data servers. The ability to communicate freely, without topology restrictions, makes a replication protocol topology-independent.

It is challenging to support topology-independent replication protocol without the assumption of uniform trust among the participating replicas. Topology-independent protocols [18, 113] allow replicas to push updates to any other replicas in the system, effectively using them as data relays. This approach is flexible and can handle varying connectivity while maximizing the use of scarce resources. However, replicas can temporarily host updates for objects in which they are not interested, and for which they have no rights to access.

Figure 1.1 shows an example of this process in an anti-entropy [113] protocol in pure P2P system. The figure shows a replication protocol that supports TI with three participating replicas: a desktop, a laptop, and a tablet. Both the desktop and the laptop store objects X and Y. The third device, a tablet, stores only object Y and is owned by a second user: Bob.

Such data placement exposes conventional replication and consistency protocols to data leakage. The system starts with a consistent set of states: both X and Y have the same values everywhere. The desktop updates X and Y, creating new versions X' and Y'. The figure shows the desktop sending new updates X' and Y' to the laptop, using Alice's tablet as a data relay. The issue is whether the tablet learns anything that it should not as it passes X' on to the laptop. If X' is not encrypted, Alice learns about its existence and contents. If it is encrypted, Alice could learn the meta-data of the file. Finally, if data and meta-data are both encrypted naively, Alice's tablet would have no way to determine where to send the update.

Cloud-based replication protocols also take advantage of TI in moving updates between cloud replicas as well as between cloud replicas and their clients. Figure 1.2 shows a similar example in which the Desktop again creates new updates X' and Y'. A cloud software agent running on the desktop, for example Dropbox [36], propagates the updates to a cloud server, which then updates Alice's other device, the laptop. The cloud is acting as a data relay, resulting in the updates potentially being pushed to *all* of Alice's client replicas and to an arbitrary number of servers of the cloud provider.

In both examples the replication protocol leaks information. In the case of the P2P scenario (Figure 1.1), the protocol leaks information to the tablet. In the cloud scenario (Figure 1.2), the user information leaks to the provider's cloud servers.

1.3 Thesis

Relying heavily on a plethora of cloud services is cost effective and allows for features to be developed and released quickly. At the same time, this reliance can severely limit the ability of the applications to define their own policies, get the consistency they want, and retain control of what data is visible during replication.

My goal in this thesis is to build systems that take advantage of the performance, scalability, and availability of the cloud storage services while, at the same time, bypassing the limitations imposed by cloud-service providers' design choices. The key to achieving this is pushing application-specific decisions where they belong [103]: the application.

Policies should be application-specified decisions. The rules of applying and consuming data updates—***a consistency protocol***—is one example of a policy. The cloud providers, through the highly available NoSQL data stores, provide best-effort consistency mechanisms. Reliable, high performing mechanisms need to be simple. In the case of NoSQL data stores, this simplicity comes from relaxing the consistency guarantees. This does not mean that an application has to settle for inconsistent data. Instead, application designs should take into account the nature of today's data stores and shift the consistency determination outside the core storage mechanism of the provider.

The trust relationships between replicas (user replicas or cloud servers) and data placement are another example of what should be an application-specified policy. The mechanism of data dissemination—***a replication protocol***—cannot determine these trust relationships. Existing P2P replication protocols achieve the desired topology-independence by assuming trust between every replica in the network. Today, this assumption has carried on to the cloud based synchronization services, compromising security. Once again, an application should not have to settle for this.

I defend the following thesis statement: *By decoupling consistency determination and trust from the underlying distributed data store, it is possible to (1) support application-specific consistency guarantees; (2) allow for topology-independent replication protocols that do not compromise application privacy.*

1.4 Contributions

I use the principle of separating the mechanism from policy [70, 123] to provide applications with stricter consistency guarantees and to allow for replication protocols that do not compromise users' privacy. The rest of this thesis is as follows:

Chapter 2: An introduction to existing consistency and replication protocols. I define and summarize various consistency protocols. I describe extensively the challenges that arise for accessing eventually-consistent data stores through today's programming patterns. I describe existing replication protocols and the challenges of supporting topology-independence. Additionally, I describe how related work addresses these challenges.

Chapter 3: `Shell`: An architecture for supporting strict consistency guarantees over eventually consistent data stores. I design and implement `Shell`. The `Shell` architecture consists of single-responsibility modules that together encapsulate consistency-related code. The `Shell` layer is located between the cloud provider and the application, executing the application-specified consistency implementation. In addition to consistency protocols tailored to application needs, `Shell` also provides application-aware conflict resolvers without relying on generic heuristics such as the "last write wins". `Shell` does not require the application to maintain dependency-tracking for the execution of the consistency protocols, as in existing approaches [14, 74]. I experimentally evaluate `Shell` over two different data-stores using real-application traces. I found that using `Shell` can reduce application-visible inconsistencies by up to 10%. I also show the overheads that come from introducing the `Shell` layer.

Chapter 4: *T.Rex*, A system for supporting topology-independent replication without the assumption of trust between all the participating replicas. *T.Rex* uses role-based access control [42] to enable flexible and secure sharing among users with widely varying collaboration types. Both users and data items are assigned *roles*, and users can access data only if they share at least one role with that data. Building on top of this abstraction, *T.Rex* includes several novel mechanisms: I introduce *role proofs* to prove role membership to others in the role without leaking information to those not in the role. Additionally, I introduce *role coherence* to prevent updates from leaking across roles. Finally, I use Bloom filters as *opaque digests* to enable querying of remote cache state without being able to enumerate it. I combine these mechanisms to develop a novel, cryptographically secure, and efficient anti-entropy protocol, *T.Rex*-Sync. I evaluate *T.Rex* on a local test-bed, and I show that it achieves security with modest computational and storage overheads.

Chapter 2

Background and Related Work

In this chapter, I provide background information about two common abstractions that we use to build modern distributed systems. I explain in detail the mechanics of Data Access Objects [87] and Topology Independence. I discuss their advantages and the assumptions on which they are based. I show how these abstractions lead to leakage when the assumptions they are based on no longer hold. By leakage, I refer to *both* exposure of complexity (non-obvious limitations of the underlying implementation) and to information leakage. Finally, I review research work related to my thesis and the two systems I built, `Shell` and *T.Rex*.

2.1 Consistency Primer

In this section, I provide short definitions of some common consistency protocols.

Strong Consistency [1]: After an update on object X completes, every subsequent access on X from every replica in the system will return the new value of X.

Eventual Consistency [113]: The storage layer guarantees that if no new updates are applied to an object X, *eventually* all *read* requests will return the last updated value of X.

Causal Consistency [4]: All causally-related operations, *reads*, *writes*, are executed in an order that reflects their *causality*. A case that establishes a causal relationship is a read on variable X followed by a write on variable Y because the read of X might influence the write of Y. Another case is a write on variable X followed by a read on on the same variable. Finally, two consecutive writes, even in different variables, are considered causally related. Some recent works in causal consistency [14, 74] introduce the concept of explicit causality. In the case of explicit causality the application developer provides the dependency graph of the operations that the application considers causally related.

Session guarantees [114]: A storage system might make consistency guarantees for the lifetime of a user- or application-defined *session*. Common models include *Read-Your-Writes*, *Monotonic Reads*, *Monotonic Writes*, and *Write follows Reads*. The *Read-Your-Writes* model specifies that after a replica updates an object X, it will never access an older value for X. The *Monotonic-Reads* model specifies that after a replica has read a value for object X, no later access will return older values of X. In *Monotonic-Writes*, a replica's writes are always serialized and applied in the same order everywhere. Finally, in the case of *Write follows Reads* write operations are propagated after reads on which they depend on.

Quorum protocols [67]: In this family of protocols there are three configuration parameters: N is the number of replicas in the system, W is the number of replicas that need to confirm a write operation, and R is the number of replicas to contact when an object is accessed for a read operation. Different values for the (N,W,R) variables provide different guarantees. First, when $W+R>N$, there is strong consistency because this configuration guarantees that any two quorums overlap, and therefore all prior decisions are seen. On the other hand, $W+R=N$ does not provide consistency guarantees because updates may not have propagated to the reader replica. Furthermore the $W<(N+1)/2$ configuration can lead to conflicting writes because the write quorums do not overlap. Finally, the $W+R\leq N$ configuration offers only eventual consistency.

2.2 The Rise of NoSQL

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability [119]. The result of this trade-off was the rise of NoSQL databases. NoSQL data stores favor high availability and low latency over consistency. In terms of consistency guarantees NoSQL databases support eventual consistency (see Section 2.1).

In contrast to relational databases, NoSQL data stores come without fixed schema. A programmer may have different schema on a per object basis. Dynamic schema on a per object basis allows applications to change without requiring any change to the underlying data store. In a relational database, a programmer would need to alter a table and also the application objects that are populated from this table. In the case of the dynamic schema, the application programmer worries about the application objects.

A NoSQL database can scale horizontally by adding more servers. On the other hand, relational databases scale for the most part vertically. Scaling vertically means that in order to scale a database server we would need to add more CPU cores or more memory. Relational databases can also scale vertically by sharding. With sharding, different database servers hold part of the data and programmer must maintain the routing logic that determines the mapping of data to a particular server shard. The major advantage of relational databases over NoSQL systems is the native support of transactions. However, it is difficult to support transactions over multiple shards. Moreover, the application developer is the one who must maintain this logic.

2.3 Data Access Objects (DAO)

The Data Access Object (`DAO`) design pattern abstracts details of the storage layer from the application. The `DAO` is often associated with JAVA Enterprise Edition, but it is applicable to most programming languages. For consistency purposes I will describe it using JAVA terms.

The canonical implementation of the pattern involves three classes. The first class is an interface that is exposed to the application, and which defines how the application interacts with the storage layer. The second class is JAVABEAN, which is the medium to transfer data between the application and the storage layer. A JAVABEAN class has a public constructor and *get/set* accessor methods. A JAVABEAN class must be serializable. The third class is the `DAO` implementation. This class implements the client interface by passing and accepting data through the JAVABEAN class.

Figure 2.1 illustrates the three basic parts of the `DAO` pattern for an application that stores NBA game events. The JAVABEAN is the `NBAEvent`. The `DAO` interface is the `NBAEventDAO`. `NBAEventDAO` defines how the application communicates with the storage layer. The third part, `DatabaseEventDAO`, is a relational database implementation of the `NBAEventDAO`.

The idea behind the `DAO` pattern is to allow for multiple implementations of the `NBAEventDAO`, for example, such as a file-based implementation and a NOSQL-based implementation. Although different implementations require storage-layer-specific domain knowledge to account for the peculiarities of the underlying system, the `DAO` layer's interface permits the application to access the storage layer without regard for storage-layer-specific domain information.

Applications, then, are responsible only for maintaining code that allows communication with the `DAO` layer. The interface between the application and the `DAO` layer will not

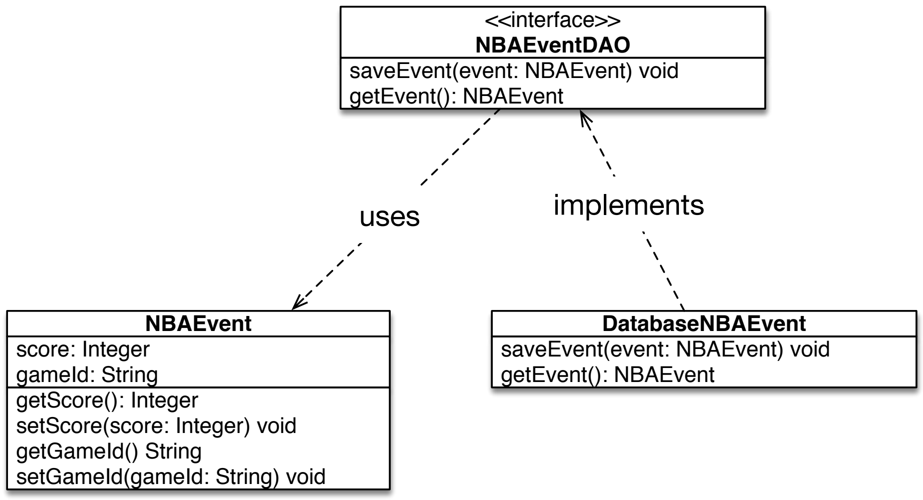

Figure 2.1: This is a high level class diagram of the implementation of the `DAO` pattern for an NBA event store. The figure illustrates the interface that the pattern exposes to the applications, `NBAEventDAO`. The data object that the pattern carries data, `NBAEvent` and one database implementation of the interface, `DatabaseEventDAO`.

change when the storage layer changes. Changing the storage layer does not affect the application itself. Similarly, changes in the application logic remain isolated in the application domain without the need to propagate them beyond the `DAO` layer. The result is that two important parts of an application's software stack, the business logic and the persistence layer, can change independently. Furthermore, the `DAO` layer allows the application to test its interaction with the storage layer through unit tests [58]. Figure 2.2 provides a bird's-eye view of the interaction between the `DAO`, persistence, and the application.

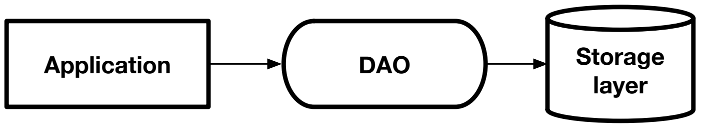

Figure 2.2: The `DAO` layer is located between the application and the storage layer. The `DAO` layer isolates the application logic from persistence domain and allows them to evolve independently providing them with a reference interface.

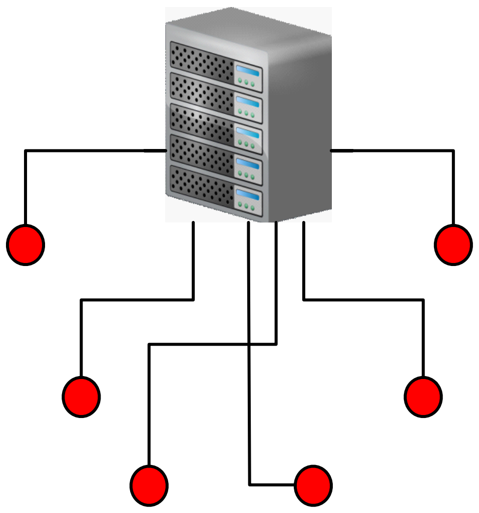

Figure 2.3: Representation of a traditional client-server model. “Thin” clients retrieve data from a server.

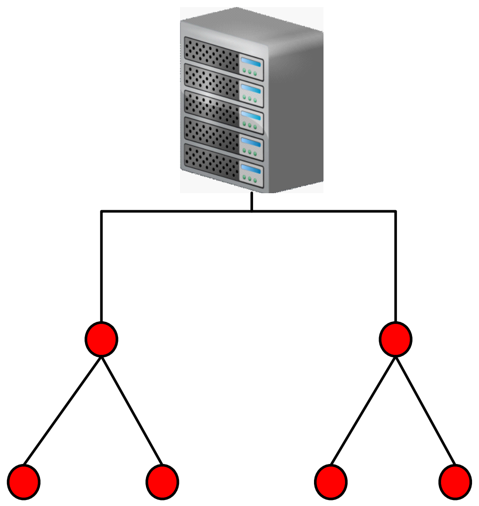

Figure 2.4: Hierarchical topology where only a few clients talk to the central server and act as caches for the rest of the replica in the hierarchy

2.4 Communication Patterns In Replication Protocols

The literature contains a vast number of papers focused on distributed storage systems [5, 26, 29, 71, 74]. Two important properties of every distributed storage system are their consistency and replication protocols. This section focuses on the replication aspect. More specifically, I will examine the mechanisms that a replication protocol uses to disseminate information to other replicas in the system. I divide the communication patterns used by replication protocols into four major categories, which I will analyze in the following paragraphs.

The most common pattern in network file systems (NFS) such as the AndrewFS [57], Coda [105], SpriteFS [82] and LBFS [81], and modern cloud replication applications, like Dropbox [36], is the *client-server* pattern. This pattern traditionally includes “thin” clients that interact with the users and a master back-end server that manages data. To replicate information between the clients, the information needs to travel first through the master

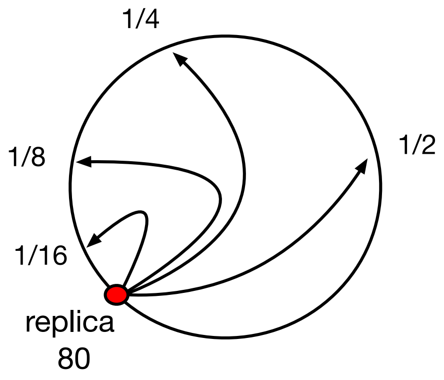

Figure 2.5: Representation of a DHT topology. These protocols involve sophisticated topology communication patterns between their participating replicas. In this image I show the finger table of node80. Finger tables are used to avoid linear look-ups.

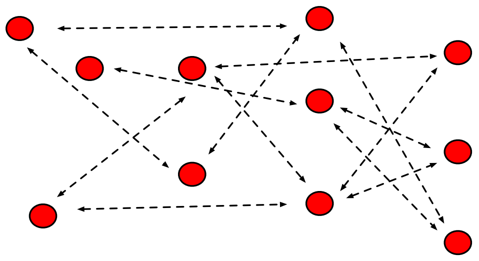

Figure 2.6: Depiction of a topology independent replication protocol where every replica in the system can communicate freely with any other replica in the system.

replica server. In modern applications, the master replica is often replaced by multiple cloud servers [67, 86]. Figure 2.3 illustrates the topology setup in client-server systems.

Another common communication pattern in the evolution of replication protocols is that of *hierarchical* [22] communication. This is the next logical step from the client-server model. In the client-server pattern, the master replica becomes a bottleneck and a single point of failure for large numbers of clients. The hierarchical model allows only a few replicas to communicate directly with the master replica, providing intermediate caching to the machines below them in the hierarchy. In practice this model of communication can allow network file systems to scale better, and was part of the file sharing revolution of the early 2000's. Kazaa [83], one of the popular file sharing networks of the period, used hierarchical caching for distributing the files to its users Figure 2.4 shows a hierarchical topology.

Distributed hash table (DHT) storage systems (Chord [108], CAN [95], PAST [100], Pastry [99], Tapestry [125], and CFS [31]) exemplify a third category of communication patterns. Servers are split across multiple replicas on a per data object or per data block basis. In most implementations, objects are replicated for high availability and data durability. DHTs use consistent hashing and map replica IDs onto a circular space. The object keys are mapped onto the same circular space with every key to be assigned to the "nearest" replica in the ID space. When a replica receives a query for an item with id_7 , it first checks whether it stores the item locally. If the replica does not store the item locally, it refers to the local routing table and forwards the query to a replica with an id closer to id_7. Figure 2.5 shows a DHT topology along a single replica (replica 80) and its finger table [108]. The k finger in the finger table point to the $1/2^{n-k}$ replica away around the ring space. DHTs use finger tables to avoid linear look-up times.

The last category consists of replication protocols that do not impose any constraint on the communication between the participating replicas. Replication protocols that allow this fluid form of communication are called *topology independent*. Protocols in this category often use *epidemic* [33] replication techniques to propagate data to other replicas in the system. The synchronization session between replicas is usually called *anti-entropy* because it reduces the total inconsistency (entropy) of the system. This family of protocols can be utilized in low bandwidth/energy environments where the communication with a master replica, in the cloud or elsewhere, would be very costly. Furthermore, the fully distributed nature of these protocols and their lack of fixed communication patterns makes them disaster-resilient [15]. The Figure 2.6 illustrates a topology independent replication protocol.

2.5 Synchronization Sessions

Replicas participating in replication protocols that provide topology independence synchronize with each other through anti-entropy sessions. In this setting, two replicas come together and exchange meta-data that describes updates seen by the replicas If at the end of this exchange one of the replicas is behind, the other transmits the missing data items.

The meta-data exchange and the status-check of the two replicas is mediated through version vectors [77]. At a high level, a version vector is a list of the following pairs: $ID_{replica}$, $Clock_{replica}$. The *ID* is a unique identifier for the replica. The *Clock* is usually a number representing the number of local updates that the replica has already performed. Each replica maintains a vector comprised of pairs. Each pair describes events seen by a corresponding replica.

Two replicas' version vectors are identical if the replicas have seen exactly the same set of updates from all system replicas. Version vectors are *ordered* if one dominates ($>=$ at all positions, $>$ in at least one) the other. Vectors are *concurrent* otherwise.

The synchronization session can be either unidirectional (*push* or *pull*-based) or bilateral. In unidirectional sessions only one of the replicas updates the other. Only the replica being updated ("pushed-to", or "pulling") will have a changed version vector at the end of the session. Version vectors of two replicas are identical after they have completed a bilateral anti-entropy session.

The Figure 2.7 shows a bilateral anti-entropy exchange between Replica_A and Replica_B. In the first message, Replica_A sends its version vector to Replica_B. Replica_B responds with its version vector and any data known to it but not to Replica_A. Finally, Replica_A returns any data not known to Replica_B. In practice, anti-entropy sessions are more concise and the data exchange happens asynchronously.

2.6 Abstraction Leakage

Abstraction in computer science and engineering is a way to manage complexity. Abstractions are simplifications of more complicated concepts and eliminate any complexity

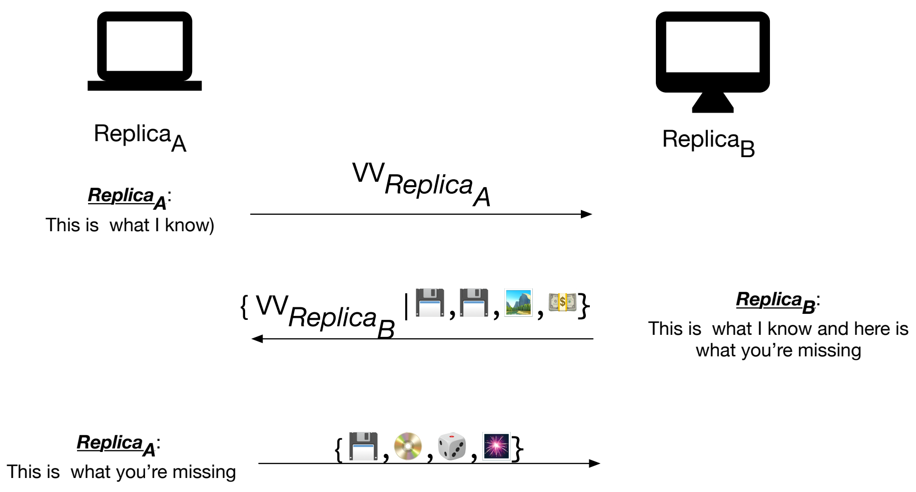

Figure 2.7: Bilateral anti-entropy session.

that is not relevant to the current context. For example, when an application uses the DAO pattern it can write and read data from an underlying data store without being exposed to the complexity of the implementation of these read and write operations. The same is true for an application that uses a replication protocol to keep its data in sync across multiple devices. All the complexity of the replication will remain encapsulated in the implementation of the synchronization protocol and will not reach the application.

Unfortunately, software abstractions are not really abstract. In reality, they are implemented by real code that runs on physical or virtual hardware. This code has properties that depend on their implementation or their infrastructure. For example, in the traditional DAO pattern setup, the data store is assumed to be a relational database. Relational databases support all the ACID [53] properties: Atomicity, consistency, isolation, and durability. Having ACID properties means that applications do not have to worry about the ordering of concurrent writes. This is a non-trivial assumption. Many replication protocols assume that every participating replica is known and trustworthy. This is another non-trivial assumption.

As Kiczales [64] and Spolsky [107] explain: all non-trivial abstractions leak complexity (non-obvious limitations of the underlying implementation). In this thesis I examine how this principle applies to data access, replication, and consistency in a distributed storage system. I show that the DAO pattern usage over NoSQL storage systems leads to leak of complexity. The `Shell` project examines the ramifications of using DAOs in cloud-powered applications. On the other hand, the *T.Rex* project shows that not all the replicas that participate in a replication protocol are equally trusted. In the case of *T.Rex*, the result of the trust assumption is information leakage. In `Shell`, the assumption of strong consistency leads to complexity leakage.

2.7 `Shell` Related Work

The massive scale of online applications led to the emergence of a new family of storage technologies that follow the BASE [92] principle. In contrast to ACID [53], BASE refers to data stores that are *basically available* to support the constant demand of always-on and five nines (99.999%) of availability. BASE systems rely on *soft-state*, as described by Clark [28], so they can be flexible and use *eventual consistency* [113]. Eventual consistency is the most common way to achieve high availability in environments characterized by the CAP theorem [23] without the use of special hardware [29]. The popularity of eventual consistency paradigm led to the development of a series of systems such as DynamoDB [32] and Big Table [26]. While these systems proliferated, traditional SQL databases have lost traction because they do not scale.

When a system does not support strong consistency, a programmer should not expect that a `READ` request issued to the DAO layer will return the latest value of data object. Programming *without* causality is extremely hard. This is the complexity leakage that sources from the DAO abstraction.

Many researchers, like Bailis [14] and Lloyd [74], have analyzed the question of how to support causal consistency in a cloud storage system. Causal consistency is important because is the strongest consistency scheme achievable [75] when operating in environments that face partitions and need high availability. These works [14, 38, 74] track the *READS* and *WRITES* of an application to create dependency graphs in the application layer that ship to the storage layer. Such storage systems can use these dependencies to provide the application with an enhanced form of causal consistency, called *causal-plus* [74].

Dependency tracking is a powerful tool. However, it relies on application developers to not only write and maintain software for tracking dependencies, but also to agree to send them to the cloud provider. The latter might not even be feasible, especially in bandwidth or energy constrained environments. This also forces an application to implement dependency tracking according to the specification of a single provider.

Several approaches rely on solutions that make assumptions on guarantees available from the cloud layer or its infrastructure. Li [71] differentiates operations into two groups, one that requires strong consistency and one that can tolerate *best-effort* consistency. Almeida [5] shows how to achieve Causal-plus by assuming that the provider relies on chain replication. Chain replication is achievable by a series of servers that are ordered in the form of a chain. Du [39] uses a periodic algorithm that relies on the physical clocks to track application dependencies. These approaches focus on specific providers' internal implementations, leaving applications, the users of the storage systems, out of the loop.

A layer-based approach somewhat similar to `Shell` appears in Bermbach [20] where application users have the illusion of causality with the system implementing session-consistency. This differs from `Shell` in that `Shell` explicitly supports causal and other consistency protocols. Zawirski [124] also proposes a scheme for supporting transaction-enabled causal-plus consistency by extending the storage layer to the actual devices of the users, which are considered partial replicas and act as caches. Both approaches [20, 124] offer a one-size-fits-all solution that deprives the application developer of the ability to select the desired consistency scheme. Moreover, in Swift [124] the researchers assume that it is possible to make the users devices an extension of the a storage layer, which is not always the case in reality.

2.8 *T.Rex* Related Work

In an era where almost everything comes with Internet connectivity and coffee makers can be used to mount large cyber-attacks [120], any assumption that security is not a priority because of a specific environment is fragile. Most of the existing replication protocols focus on data placement, coherence, and consistency, omitting discussion of security and privacy. Removing the assumption of implicit trust leads to information leakage where replicas, which might participate in the replication protocol even though not fully trusted, will gain access to all of the users' information.

As I will describe in Chapter 4 *T.Rex* is a system that supports partial replication, arbitrary consistency, and topology independence but without compromising application privacy. Except PRACTI [18] no other replication system provides all three properties. The list includes Bayou [90, 113] and Ficus [52], two of the first works to establish the anti-entropy synchronization sessions with the use of version-vectors. Systems like Anzere [97], Perspective [102], Eyo [109], Cimbiosys [94], PodBase [91], and UIA [43] use policies to abstract replication rules, data placement, and durability. In addition, systems like BluFS [85], EnsemBlue [89], and ZZFS [78] use high-level policies in similar environments to conserve energy. Moreover, PADS [19] and PRACTI [18], which are policy architectures and replication frameworks, can be used to build highly flexible replicated storage systems.

As demonstrated in the previous paragraph the "partial replication, arbitrary consistency, and topology independence" taxonomy is hard to achieve. Most of the work in this area concentrates on data placement, consistency, and coherence without seriously addressing security concerns. Furthermore, the assumption is that security issues like access control, authentication, and confidentiality are orthogonal to issues of data replication and consistency, and these issues could therefore be handled by mechanisms in higher levels. Another common assumption is that all participating devices have the same access control policies. In *T.Rex*, I show that a straightforward application of these assumptions results in information leakage.

One exception is the access control extension of Cimbiosys [121] that defines access rules through SecPal [17]. This approach, when used with Cimbiosys [94], may be problematic because policies are propagated as regular objects, and Cimbiosys supports eventual consistency. As a result, security changes may be lost because there is no convergence. However, though SecPal statements are signed, data is unencrypted, the system assumes a single trusted authority, and the underlying system relies on a tree-structured replica topology.

T.Rex uses fork consistency to avoid inter-role information leakage. It borrows ideas from a series of systems. The object coherence in *T.Rex* has similarities to Qufiles [118] and SUNDR [72]. Qufiles are a system abstraction that gives a server the ability to support different trans-coded versions of a file under one common umbrella. SUNDR proposed *fork consistency*, which allows attacks to be constrained to forking version histories, which can then be detected.

At the other end of the spectrum are replicated object systems that tolerate Byzantine faults, like PBFT [25], Farsite [2], and even Oceanstore [66]. These systems differ from *T.Rex* in their use (*T.Rex* is designed for sharing and collaboration, with roles defined by high-level predicates) and in their goals (*T.Rex* adds information leakage to access control and confidentiality).

2.9 Summary

In this chapter, I presented two major abstractions that I examine throughout this thesis: (1) the DAO pattern that helps applications access their data without knowing the details of the underlying storage layer and (2) the replication protocols that keep the application data in sync across different devices. I also examined instances where existing works make assumptions necessary to support their abstractions, and I examined the nature of these assumptions. Moreover, I showed that these assumptions lead to complexity and information leakage.

Chapter 3

Shell: Supporting strict consistency guarantees over eventually consistent data stores

In this chapter, I present `Shell`, a layered architecture that can provide applications with stronger consistency guarantees when operating over best-effort distributed data stores.

The contributions of this work can be summarized as follows:

- I show how to eliminate DAO-induced abstraction leakage by designing a layered system I call `Shell`. The architecture consists of single-responsibility modules that collectively encapsulate consistency-related artifacts of the underlying cloud implementation, isolating application code from any reliance on specific implementations.
- I show how to support protocols *stricter* than eventual consistency without requiring applications to explicitly track dependencies [14, 74], or having control over the provider's programmatic interfaces. I further show that `Shell`'s architecture allows applications to customize conflict resolution of concurrent writes; eliminating the need to rely on one-size-fits-all generic heuristics (e.g., "last write wins").
- I use a trace driven evaluation, four different data-center topology setups, and two different data stores to examine the performance and possible overheads `Shell` in-

troduces for applications operating over eventually consistent data stores. Our high-level findings are that (1) up to 9% of updates seen by the application would be inconsistent without a consistency enforcing layer like `Shell`, and (2) the number and character of those inconsistencies varies according to the underlying providers.

My key insight is that separating consistency enforcement from what cloud providers do extremely well (providing availability and data durability) allows applications to be written without consideration of the underlying consistency. The `Shell` layering architecture isolates the application code from both consistency implementations and the cloud communication libraries. Shell introduces well-defined boundaries between layers allows application code to be easier to develop and to maintain. Furthermore, pluggable consistency implementations can be modeled and developed independently from the rest of the application.

The rest of the chapter is organized as follows. In Sections 3.1, 3.2 and 3.3 I present the design, implementation, and experimental evaluation of `Shell`. Finally, I summarize with the lessons learned and conclude in Section 3.4.

3.1 Design

`Shell` is a software layer (Figure 3.1) designed to isolate consistency implementations and cloud-provider APIs from application code. I use the term *replica* to refer to an instantiation of a running application, together with a linked `Shell` layer and a binding to a cloud provider. Replicas interact solely through reads and writes to the underlying cloud provider.

An application can use the `Shell` abstraction with only two steps. First, the application chooses a consistency level by instantiating a consistency object. Second, all application objects that are to be shared across replicas are derived from `Shell`'s consistency-enabled

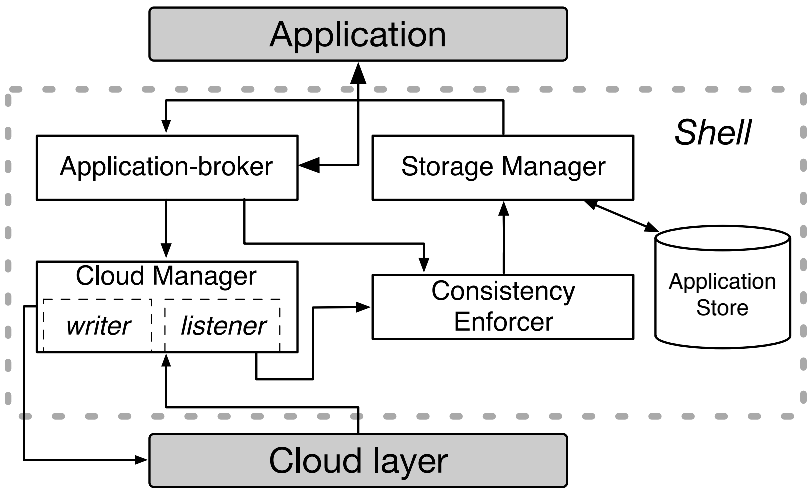

Figure 3.1: Shell consists of four software modules along with an application-store.

objects, discussed below. Optionally, applications can provide per-object *conflict resolvers*.

Shell consists of four internal modules and an *application store*, which together abstract consistency-related operations and encapsulate communication with the underlying storage layers. Figure 3.1 illustrates the four building blocks of Shell along with the application-store which in the current implementation is the in-memory data store REDIS [104]

This section describes the Shell programming abstraction that provides access to Shell guarantees and the overall Shell architecture. I also describe the life cycle of incoming updates: how they reach Shell from other replicas, and how applications interact with the Shell layer.

3.1.1 Shell objects

Shell introduces a consistency enabled object (SO) that applications must inherit to use the system. SOs contain a *version* object that I use to enforce consistency semantics when I send the object to another replica. Applications may provide conflict resolution code by implementing the *resolve(otherShellObject)* function that is part of the main object.

```java
public abstract class SO {
   private AppDomain domain;
   private ShellVersion version;
   public abstract SO resolve(SO other);
   public abstract Class identify();
   public abstract String guid();
}
```

Figure 3.2: The SO implemented Java. `Shell` operates in application-agnostic containers I call generic shell objects (SO). `Shell` isolates any application logic; it thus provides interfaces that application developers need to implement so the system knows how to read and write application data.

The SO contains application domain information that differentiates the objects of different applications from one another. Finally, the SO contains an *identify()* function where the application should provide a way to identify the type of object, so that the system may later serialize the object and send it over the network. Figure 3.2 provides an example of a JAVA based implementation of the SO.

Each `Shell` replica has a global unique identifier (GUID) and can serve multiple applications. Each application is identified by another GUID. The hash of the module and application GUID is called instance-id (IID).

Internally, `Shell` maintains a version vector [52, 88, 105, 113] for every application it supports. The vector is a list of IID, counter pairs. The counter signifies the number of updates `Shell` has seen for this particular application. `Shell` is able to differentiate between writes of different applications by leveraging the SO's domain information.

The *version* object, part of the SO, has two parts. The first part is the snapshot of the application's version vector the time an object reaches `Shell`. Other systems have used this technique, most notably systems like Cimbiosys [94] and WinFS [76] refer to this approach as *made-with-knowledge*. The second part is the *instance id*.

```
public interface AppStorageManager {
   void put(SO event);
   SO get(String key);
   List<SO> scan(String key);


   Future<SO> getAsync(String key);
   Future<List<SO>> scanAsync(String key);
}
```

Figure 3.3: An example of a Java-based implementation of `storage-manager`

3.1.2 Architecture

`Shell` consists of four major modules and the `application-store`. The `application-broker` module is the contact point between applications and `Shell`. The `consistency-enforcer` module is responsible for executing the consistency protocol of choice. The `storage-manager` module handles incoming updates once they become consistent. The `cloud-manager` module isolates all cloud-provider APIs from the application and encapsulates all accesses to and from the provider storage layer. Finally, the `application-store` serves outgoing application data and buffers incoming updates that do not yet meet application consistency criteria. In this section, I describe each in more detail.

The Application-Broker Module

The `application-broker` facilitates the communication between applications and `Shell`. The system provides three generic operations to applications: PUT, GET and SCAN. The module passes GET or SCAN requests on to the `application-store`. When receiving a PUT request, the module passes the request to both the `consistency-enforcer` and the `cloud-manager` modules, and sends an accept acknowledgment (ACK) back to the application. The current implementation's default is to treat PUT requests as non-blocking, so the ACK is sent immediately. However, applications can also configure PUT requests to be blocking.

The Consistency-Enforcer Module

The `consistency-enforcer` is responsible for applying the consistency protocol of choice on every update it receives. Updates reach the module by two paths: local updates that the `application-broker` propagates and network updates from other `Shell` replicas. `Shell` acts as a daemon that constantly evaluates updates. Each incoming update is evaluated for conformance to the chosen consistency protocol, resolving to either *drop*, *apply*, *defer*, or *conflict*.

A *drop* results when the consistency protocol detects an obsolete update. For example, if an application requiring *monotonic reads* has seen update i for object X, any update of X prior to i is obsolete, and should be dropped when detected. In this particular scenario the `consistency-enforcer` will *drop* any incoming update with a stale value.

An outcome of *defer* signifies that an update, while potentially useful, is being temporarily blocked by the consistency protocol. For example, when enforcing *causal consistency* the module will only apply updates conforming to causal order. In this case the `consistency-enforcer` temporarily stages incoming updates that do not comply with the consistency invariants until a later re-evaluation unblocks them. The arrival of a new update will cause the re-evaluation of staged updates.

The local version vector and the update's version vector might be in *conflict*, implying concurrent writes.

As mentioned in Section 3.1.1, the SO may include an application-supplied resolve method that is called by `Shell` to resolve the conflict. If no conflict implementation is provided, `Shell` will engage a last-write-wins strategy. I implement this heuristic for two

SOs by comparing their globally-unique creation timestamps.

The Storage-Manager Module

The `consistency-enforcer` operates on the meta-data carried by the SOs, treating application data as opaque byte streams. After the SO representing a new write becomes consistent, the `consistency-enforcer` will move it to the `application-store` and make it directly accessible by application operations. However, such a SO must first be translated into its application-specific format.

The `storage-manager` module guides this transformation, and also hides the application storage APIs from the rest of the architecture. Figure 3.3 shows a JAVA based example of a simple `storage-manager` interface providing both synchronous and asynchronous operations for reading and writing data to the application storage.

The existence of the `storage-manager` module highlights a basic principle behind the design of `Shell`: *Isolation between layers* (in this case, isolation between the application logic and `Shell`.) Other proposals [14] that use layering have the application track explicit causal relationships between applications objects. Having explicit application information, these systems can achieve tighter consistency that meets the application needs. While this approach may offer precise tracking of these relationships, it forces the application developers to maintain and update them every time they want to add a new feature or fix a software bug. However, the application space changes rapidly. Software products are developed by employing the *minimum viable product (MVP) [112]* approach where only a set of few critical features are developed to attract enough people to use it initially. This usually proves the product's concept and creates a feedback loop. By constantly adding features, the accidental complexity [24] of the application code remains high for longer periods of time. I believe that maintaining consistency information in the application level leads to errors, especially in software where products are developed in multiple iterations under the agile software principles [56].

The Cloud-Manager Module

The cloud-manager consists of two sub-modules, the update$_{writer}$ and update$_{listener}$. The update$_{listener}$ identifies incoming updates and passes them to the appropriate consistency-enforcer for evaluation. The update$_{writer}$ receives updates from the application-broker, encapsulates them in a form that the underlying provider software understands, and pushes them to the cloud layer. Updates pushed to the cloud are immutable objects [16], identified by GUIDS. Shell offloads the burden of replication and durability to the provider, which are strengths of cloud services [11, 12, 36, 44].

While the storage-manager isolates consistency from the application (above), the cloud-manager isolates the application from the cloud provider APIs (below), keeping all the providers' API calls in one place.

Having a central controller for talking with cloud services allows transparent mobility across multiple cloud providers. Existing research [21, 27, 122] has shown that employing multiple providers can be cost effective, more secure, and help avoid vendor lock-in.

Application storage

Shell includes an application-store to serve generic key-value methods such as PUT, GET, and SCAN. Shell also uses the application-store to temporarily buffer updates that do not meet the application consistency guarantees: Such as SOs that the consistency-enforcer marks as *defer* (the *defer buffer*). The defer buffer holds incoming updates until they meet application consistency guarantees (SOs marked as *defer* by the consistency-enforcer).

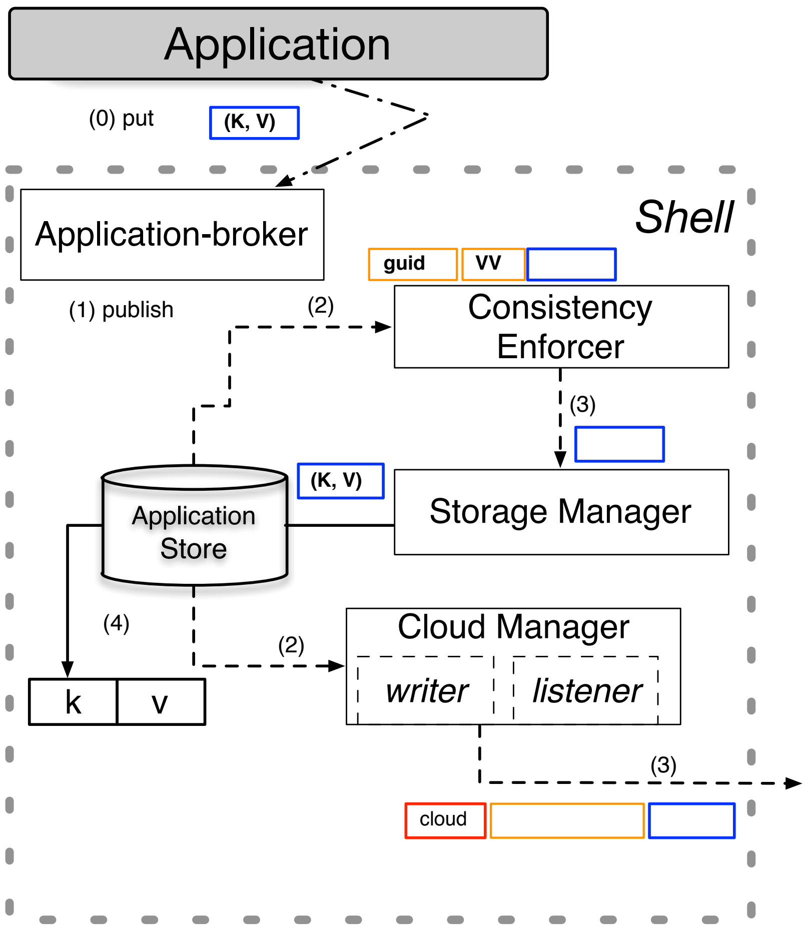

Figure 3.4: An overview of the PUT flow in Shell. (0): the application issues a PUT request. (1): The application-broker assigns a version (GUID, version vector and in (2) propagates the SO to consistency-enforcer and the cloud-manager. During (3) the cloud-manager pushes the update in the cloud and the consistency-enforcer evaluates. (4): The update is marked as *apply* and storage-manager stores it in the local application storage.

A Shell-enabled architecture also uses the providers' storage infrastructure as its sole channel for communication among replicas. After a SO has been communicated to and marked *consistent* at all replicas, it becomes *obsolete* and can be garbage-collected.

In this implementation of Shell, I use REDIS [104] to implement the application-store. REDIS lives in-memory, and is fast and lightweight. I maintain two different key spaces to differentiate between consistent and *defer*-marked data. Moreover, REDIS offers message brokering capabilities that I use to transfer messages between the different Shell mod-

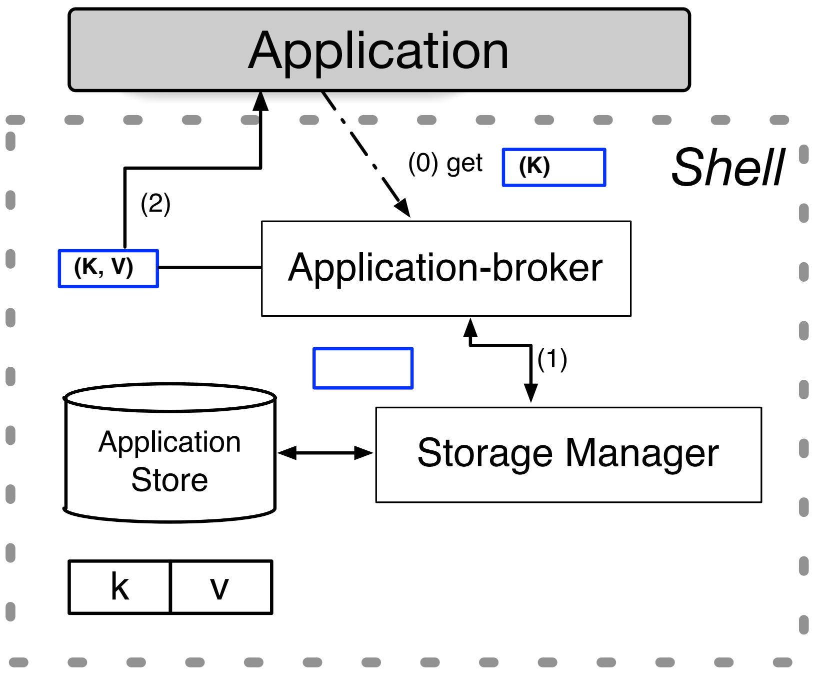

Figure 3.5: An overview of the GET flow in `Shell`. (0): the application issues a GET request. With the help of `storage-manager` the `application-broker` retrieves the data from the `application-store`, at (1). Finally, the `application-broker` serves the data to the application in (2).

ules. For example in Figure 3.4 during step (2) REDIS propagates the messages from the `application-broker` to `consistency-enforcer` and `cloud-manager`, through publish-subscribe messaging.

PUT, GET, and Incoming Updates

PUT: Figure 3.4 shows the flow of a PUT request. I have divided the flow into five steps. During step (0) the application issues a PUT request to the `application-broker`. At this stage, the application object is already encapsulated in an SO, but without an assigned version. Upon receiving a request, the `application-broker` (1) assigns it a version and publishes the SO to the `consistency-enforcer` and `cloud-manager`. The `consistency-enforcer` then (2) evaluates the update and encapsulates it in the format

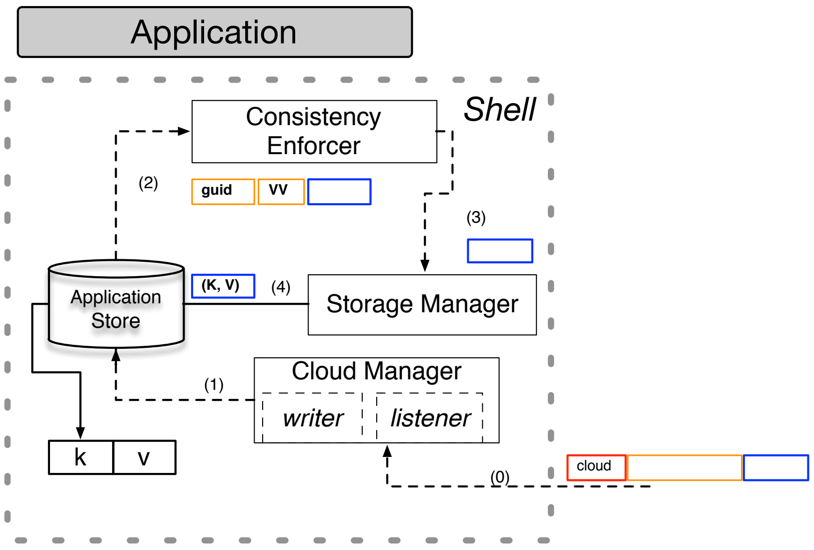

Figure 3.6: An overview of the reception of a cloud update. step (0): The `cloud-manager` monitors the provider storage for incoming updates and receives one. During step (1), the `cloud-manager` extracts the SO from the cloud update and pushes it to `consistency-enforcer`. At step (2) the `consistency-enforcer` evaluates the update. If the result is *apply* uses the `storage-manager` (step (3)) which translates the update to application object and saves it (step (4)) in the application storage.

of the underlying cloud provider. Next, the `cloud-manager` (3) pushes the update to the cloud and the `consistency-enforcer` after evaluating the update as *apply* and (4) using the `storage-manager` to save it in the application storage.

The figure also illustrates how the data is encapsulated. A `application-broker` receives the data in a form of an application object, which inherits the SO, and attaches meta-data. The `consistency-enforcer` operates only on meta-data. The `cloud-manager` encapsulates the data even further by adding appropriate cloud semantics around the SO; for example by translating the SO to an S3Object or to a DynamoDB record.

GET: Figure 3.5 shows the flow of a GET request, which consists of three steps. First (0) the `application-broker` accepts the request. The `storage-manager` then (1) reads the data from the application data store and (2) serves the data back to the application.

INCOMING UPDATE: Figure 3.6 shows an example flow of an incoming update from the cloud to another `Shell`-enabled replica. The Update$_{Listener}$, described in Section 3.1.2, constantly monitors the provider store for incoming updates. In this example the Update$_{Listener}$ discovers a new update.

The Update$_{Listener}$ (0) transforms the update from a cloud-specific form to a SO, and then (1) passes the update to `consistency-enforcer`. At step (2) the update is evaluated against consistency constraints. In this example the update has been marked as *apply* by the `consistency-enforcer` module, but other outcomes (*drop*, *defer*, *conflict*) are possible as well. If the result of the evaluation is an *apply*, the `consistency-enforcer` will update the local version vectors. Finally, the SO is (3) translated to an application object with the help of the `storage-manager` and (4) saved into the consistent application data store.

3.2 Implementation

`Shell` is implemented in 3.2K lines of JAVA source code. The communication interface between the `application-broker` and its clients is based on ZeroMQ [60]. It uses REDIS [104] for the *defer* buffer and application store. In our implementation REDIS also acts a message broker between `Shell`'s internal modules by utilizing its built-in publish-subscribe capabilities.

I developed two different `cloud-manager` implementations. The first uses publish-subscribe to propagate updates. Every replica subscribes to the update stream of every other replica in the system. In this setup the Update$_{Writer}$ module is the publisher, and the Update$_{Listener}$ module is the subscriber. The Update$_{Listener}$ of every replica subscribes to the update topic stream of every other replica in the system. Upon the reception of a new update from the `application-broker`, the Update$_{Writer}$ will publish the update to its subscribers. For the implementation of the publish-subscribe, I used the ZeroMQ implementation [59] of the pattern, which maps cleanly to the pragmatic general multicast (PGM) protocol [50].

In the second setup I operate `Shell` over Dropbox [36] with the `cloud-manager` to be responsible for all the communications *from* and *to* Dropbox. In this case the Update$_{Writer}$ is a Dropbox client that uploads every update to the Dropbox server. On the other hand, the Update$_{Listener}$ uses long polling to query Dropbox for new updates. In the implementation I used publicly available Dropbox APIs and I did not alter the default rate limiting policy that is in place.

I have implemented three consistency protocols: causal consistency (CC) [4], monotonic reads (MR) [114] and eventual consistency (EV). In eventual consistency I only rely on what the `cloud-manager` gives us, having `Shell` writing and reading updates as they come.

Why two different storage layers: Applications that operate over a cloud distributed replicated data store are bound to experience inconsistencies. With the ZeroMQ implementation I show that application-visible inconsistencies occur even when an application completely controls the communication layer. This is mainly due to propagation delays in the network layer, but can also be caused by heavy replica loads. I use the Dropbox back-end to show the effects of not having control over the communication layer, and of the application being susceptible to the providers' design choices and implementation. Thus, for Dropbox I expect inconsistencies due to both propagation delays in the network layer as well as API and service implementations.

3.2.1 Dealing With Unreliable `cloud-manager`

Dropbox was the source of the most problems throughout the implementation. Although Dropbox supports particularly low read/write rates, it was consistently missing updates while on the wire. To overcome this, I implemented a robust retry mechanism based on the failsafe [54] library.

Dealing with the `cloud-manager` failures highlights the benefits of having a layered approach like `Shell` in place. `Shell` abstracts the `GET`, and `PUT` for the application. This lets the developer to focus solely on application code rather than dealing with the `cloud-manager` peculiarities. In addition, the application code, I later developed (Section 3.3) to evaluate `Shell`, remained free from any Dropbox and failsafe related APIs.

3.3 Evaluation

The traditional way of accessing, using a DAO, a non-ACID storage layer from the application code leads to inconsistencies. In this section, I experimentally show this phenomenon by using a trace driven approach. I use an example application (Section 3.3.2) to replay a data-set (Section 3.3.1) that exhibits causal relationships between its events. Initially, I rely on eventual consistency or the build-in consistency guaranties of the storage layer, and I measure the inconsistencies that surface in the application.

Using the results of the eventual consistency as baseline, I show that *if* I enable `Shell`'s layered approach, I can minimize inconsistencies. Furthermore, I allow the application to use a consistency protocol of choice like monotonic reads or causal consistency.

I repeat our experiment for two different back-end layers (ZeroMQ, Dropbox). I use four different network topology setups by assigning the replicas in different data-centers. Apart from measuring the consistency our driver application receives, I also examine how often it has access to fresh updates by introduction the metric of `update-visibility` (Section 3.3.5).

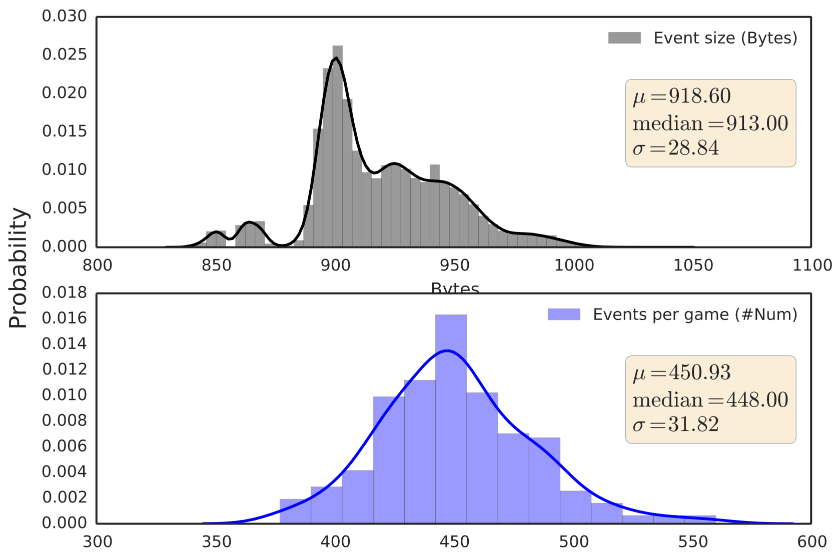

Figure 3.7: A summary of the NBA data-set I use to evaluate `Shell`. Every trace contains on average 451 causally related events. Every event is on average 919 bytes. The graph shows the median and the standard deviation σ for both the number of events and the event size.

3.3.1 Dataset

The data-set contains play-by-play NBA game traces [13] from all the regular season games of the last 10 years which accounts to 8550 game traces. The events of every trace have an integer identifier $id \in (1, max(game\_trace))$. An event carries information about what takes place in the game and also carries information about the score and the time remaining in the game. Figure 3.7 shows that each game trace contains on average 450.9 events and that every event on average has size of 918.6 bytes. All the events of a trace are causally related which means that if the events were executed in a single machine the event with id_1 *happens before* the id_2. The events of a single game trace form a causality chain with length to be equal $max(id_{trace})$. This data-set offers substantially longer event chains compared to existing studies [14] where the average length was at most 25 events long. The event id is artificial, I have attached it to the events so I have a ground truth for our experiments. I believe that longer event chains provide a benchmark that focuses on the cost of consistency-related operations.

Why this is a representative data-set: In environments where availability is a priority and partitions are unavoidable existing studies [75] have shown that causal consistency is the strongest scheme achievable. However, other consistency schemes [114, 119] which provide weaker guarantees might be appropriate depending the application needs. I choose a data-set rich in causal relationships, and I run Shell's evaluation over multiple consistency protocols (causal consistency, monotonic reads, eventual consistency) to observe whether or not `Shell` can satisfy the causal guarantees and what is the ordering to achieve this.

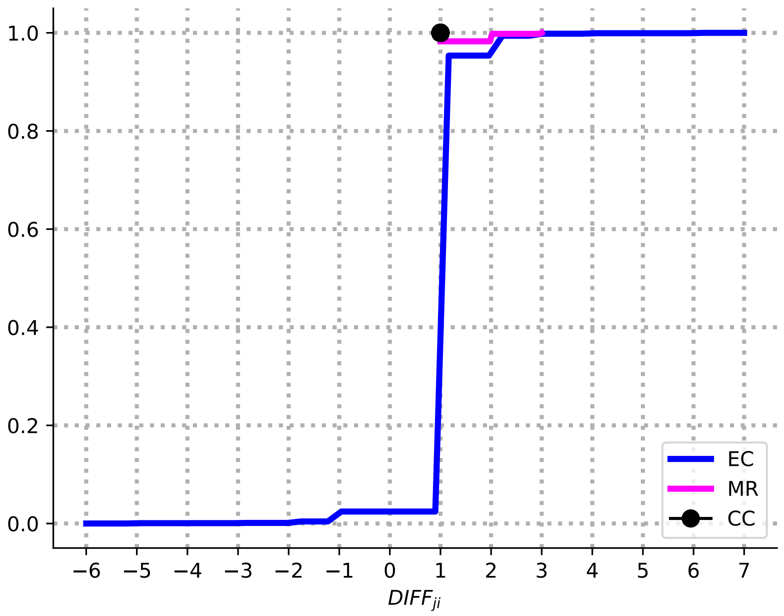

Figure 3.8: Dropbox Backend: DIFF$_{ji}$ metric for the *East-West* data center topology with LANSync *disabled*. `Shell` maintains the causality chain (DIFF$_{ji}$ =1) across every replica. For monotonic reads, `Shell` applies updates where DIFF$_{ji} \geq 1$, which means that some are future updates. Finally in eventual consistency I observe both future and stale events

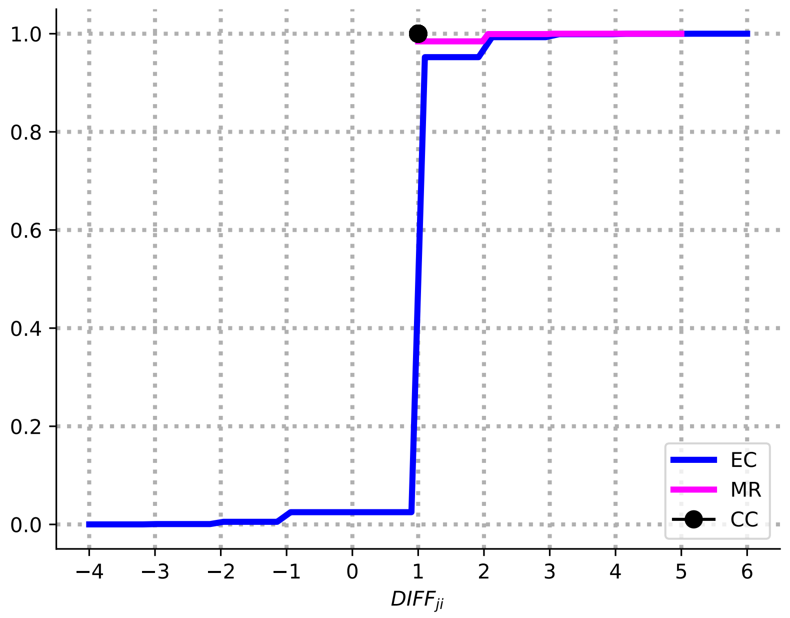

Figure 3.9: ZeroMQ Backend: DIFF_{ji} metric for the *East-West* data center topology. I observe similar behavior with Figure 3.8

3.3.2 Application: NBA Broadcast

Our evaluation is trace driven. For every experiment with the ZeroMQ back-end I replay the events from 30 NBA games, while with the Dropbox back-end I replay the events of 10 games (the ZeroMQ backend is much faster than the Dropbox back-end). I use a multiple writers/readers approach that make concurrent updates more likely.

In the broadcasting application, replicas communicate their state through two shared objects: `lastEvent`, which holds the most recent application-visible event, and `eventList`, which holds all locally-seen events. The `eventList` is a log of *apply* events, ordered by the times at which the `consistency-enforcer` marked them as *apply*.

I divide the events of the games traces to all participating replicas and I broadcast games sequentially. The replica that has in its possession the event with id_0 starts the game by submitting a `PUT` request to `Shell`. Then the event is replicated to every other replica in the system. Upon the reception of an event with id_i, a replica that in its local trace has the id_{i+1}, will broadcast the next event. If a replica has last seen an event with id_{x+1} but receives an event with id_{x+4} it will search its local trace for the event of id_{x+5}. If the replica has the event id_{x+5}, it will broadcast it next without waiting to receive the events with id_{x+2} and id_{x+3}. The replicas do a read every 100 ms; this pause denotes the end of a round for the broadcast application.

To capture how well `Shell` maintained the *happens before* relationship of the events of every trace, I introduce a new consistency metric I call DIFF_{ji}. I define DIFF_{ji} as $id_j - id_i, \forall(i, j) \in \{0, len_{eventList} | j - i = 1\}$. When $\text{DIFF}_{ji} = 1$ between two events in the *eventList*, it means that they were consecutive in the original trace. A $\text{DIFF}_{ji} > 1$, means that the replica instead of receiving the event with id_{i+1} received an update further in the future. This can also happen when two conflicting updates take place in parallel. Finally, when $\text{DIFF}_{ji} < 0$, it means that the replica received a stale update.

3.3.3 Setup

For the experiments, `Shell` system includes 12 replicas. I distribute them in four different data-center topology setups. These setups are: *Single* where all the replicas are in a single data-center. *West*: It divides the replicas in two data centers, in Oregon and California. *East-West*: It places 3 replicas in California, Oregon, Ohio, and Virginia. Finally, *Global* places each one of the 12 replicas in 12 different data-centers around the world; it includes locations in Dublin, Frankfurt, Sydney, Seoul, Tokyo, Sao Paulo, Singapore, and Mumbai along with the location in the *East-West* topology. For the topology setups where I have more than one replica in the same data-center, I place all the replicas in the same availability zone [10].

I used `m4.2xlarge` EC2 instances. In the case of the Dropbox implementation I repeat the experiments with the LANSync [37] feature both enabled and disabled.

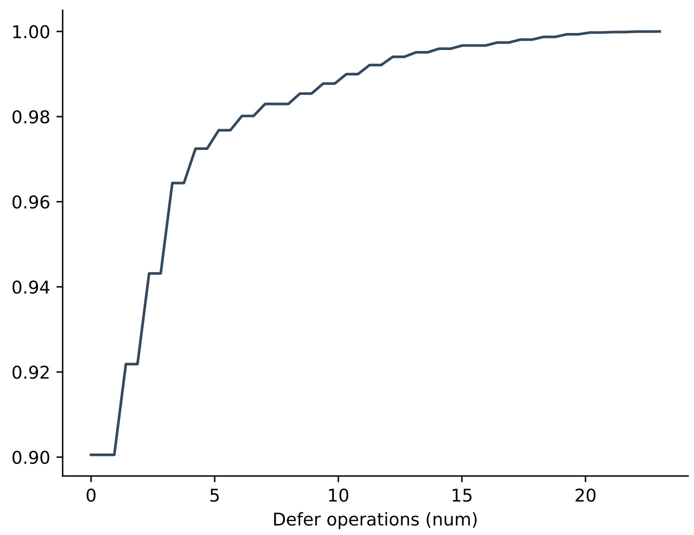

Figure 3.10: Dropbox: CDF of the *defer* operations throughout the causal consistency experiments for every participating replica. In the case of the Dropbox back-end I observe, that 98% of the updates are not marked as *defer* and can be applied as soon as they arrive in `Shell`, following causal consistency. However, the rest 2% might experience up to 14 *defer* operations before `Shell` is ready to *apply* it.

3.3.4 Consistency

In this section, I measure the inconsistencies our application experiences when operating above an eventually consistent data store. I also show how enabling the `Shell` layer along with the SOs help to achieve the desired consistency. The results I present are for the *East-West* data-center topology. In the case of the Dropbox back-end I show the results for the LANSync feature disabled. Figures 3.8 and 3.9 show the CDF of the DIFF_{ji} metric for all the consistency protocols for both back-ends. The X-axis holds the values of the DIFF_{ji} metric. In the case of eventual consistency, since this is the baseline, `Shell` will not affect the value of the DIFF_{ji} metric at all. In eventual consistency when the `Shell` layer receives an update it marks it as *apply* and passes it to the `storage-manager`.

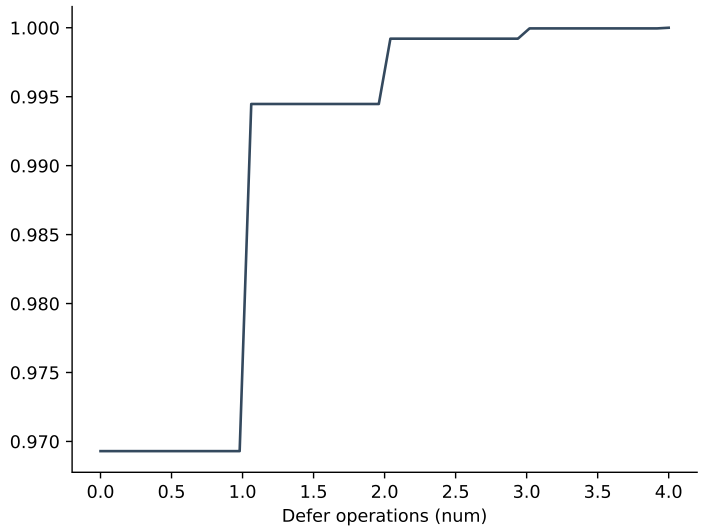

Figure 3.11: ZeroMQ: The figure shows the CDF of the *defer* operations throughout the causal consistency experiments across all the participating replicas. The 97% of updates are applied without any *defer* operation but throughout the experiment an update might experience up to 4 *defer* operations before it gets to be applied in a causal order. The difference between the two back-ends (Figure 3.10) comes from the fact that the custom ZeroMQ back-end updates propagate in the entire topology faster than they do in the case of the Dropbox back-end.

In the case of the Dropbox back-end, Figure 3.8, for eventual consistency the application experiences both future ($\text{DIFF}_{ji} > 1$) and stale ($\text{DIFF}_{ji} < 1$) events. For eventual consistency, 7% of the total messages are not ordered, and of them 34.5% consist of stale updates. In this setup the most stale update that reached the application had DIFF_{ji} =-6 and the update that was furthest in the future had DIFF_{ji} =7. In the case of the ZeroMQ, Figure 3.9, back-end I observe similar behavior. The DIFF_{ji} metric takes values $\in [-4,6]$, 7.5% of the updates are reaching the replicas not ordered, and of those 35% consist of stale updates.

In monotonic reads, by definition, I cannot observe stale updates at least throughout a session; in the experiment a single game is a session. As expected, I observe events only with $\text{DIFF}_{ji} \geq 1$. By definition in monotonic reads the DIFF_{ji} metric takes values $\in [1,3]$ and

$[1,5]$ for Dropbox and ZeroMQ. Future updates ($\text{DIFF}_{ji} > 1$) consist of 1.8% of the total updates. Finally, Causal consistency, respects 100% of the chains and all the messages maintain the DIFF_{ji} =1 invariant for both back-ends.

In the rest of our topology setups I observe the following for the DIFF_{ji} metric: In the case of the Dropbox back-end I observe: In eventual consistency, the $\text{DIFF}_{ji} \neq 1$ events range $\in$ [5.7, 9]% of the total updates, and from those on average 34% are stale. For monotonic reads the percentage of updates with $\text{DIFF}_{ji} > 1$ ranges $\in$ [1.7, 3.4]%. In all setups, causal consistency gave 100% of events with DIFF_{ji} = 1. In this setup every update that leaves from a replica first goes to a Dropbox server which stores the update and then propagates it to the rest of the replicas in the topology. The main variations are due to the network distance between the Dropbox and the replicas in our topology.

On the other hand, for the ZeroMQ back-end I observe different behavior. In the *Single* data center topology, I observe no inconsistencies for all three protocols. More specifically for the eventual consistency, 99.998% of the updates come with DIFF_{ji} = 1. However, this changes dramatically when the replicas are in different data centers. For example in the *Global* topology setup, I observe that the percentage of updates with DIFF_{ji} = 1 falls to 90.07%. Monotonic reads follows similar behavior but less extreme. In this case, I observe the variation to be between 99.999% to 97.4%. As in the case of the Dropbox, the ZeroMQ back-end does not affect causal consistency which maintain the DIFF_{ji} = 1 invariant for 100% of the updates.

Figures 3.10 and 3.11 show the CDF of *defer* operations that the `consistency-enforcer` took throughout the causal consistency experiments. For the Dropbox back-end, on average every replica has to *defer* 84 updates accounting for 1.77% of the total update population. For every update that is marked as defer, `Shell` does on average 3.3 (*median = 2.0*) more *defer* operations to satisfy causal consistency for a given update. For the ZeroMQ back-end, per replica I observed 352 updates to be marked as *defer*, but with a standard deviation

	Single LANSync	Single	*West* LANSync	*West*	*EastWest* LANSync	*EastWest*	*Global* LANSync	*Global*
X_D: Total *defer* Dropbox	μ_{X_D}=229.16 σ_{X_D}=18.51 Md_{X_D}=223.5 7.9% of total	μ_{X_D}=289.16 σ_{X_D}=19.12 Md_{X_D}=285 9.8% of total	μ_{X_D}=309.83 σ_{X_D}=13.92 Md_{X_D}=312 10.5% of total	μ_{X_D}=449.41 σ_{X_D}=33.12 Md_{X_D}=450.5 14.5% of total	μ_{X_D}=290.25 σ_{X_D}=11.6 Md_{X_D}=293 9.6% of total	μ_{X_D}=299.3 σ_{X_D}=19.8 Md_{X_D}=296.5 9.9% of total	μ_{X_D}=267.9 σ_{X_D}=23.3 Md_{X_D}=275.0 9.0% of total	μ_{X_D}=279.5 σ_{X_D}=32.4 Md_{X_D}=277.5 9.4% of total
Y_D: Single *defer* Dropbox	μ_{Y_D}=3.9 σ_{Y_D}=3.6 Md_{Y_D}=3 max_{Y_D}=20	μ_{Y_D}=3.9 σ_{Y_D}=3.6 Md_{Y_D}=3 max_{Y_D}=23	μ_{Y_D}=3.9 σ_{Y_D}=3.8 Md_{Y_D}=3 max_{Y_D}=23	μ_{Y_D}=6.4 σ_{Y_D}=7.5 Md_{Y_D}=3 max_{Y_D}=36	μ_{Y_D}=4 σ_{Y_D}=3.7 Md_{Y_D}=3 max_{Y_D}=23	μ_{Y_D}=4.2 σ_{Y_D}=4 Md_{Y_D}=3 max_{Y_D}=23	μ_{Y_D}=3.3 σ_{Y_D}=2.7 Md_{Y_D}=3 max_{Y_D}=18	μ_{Y_D}=4.2 σ_{Y_D}=4 Md_{Y_D}=3 max_{Y_D}=23
X_Z: Total *defer* ZeroMQ		μ_{X_Z}=0 σ_{X_Z}=0 Md_{X_Z}=0 0% of total		μ_{X_Z}=1.08 σ_{X_Z}=1.6 Md_{X_Z}= 0.5 0.007% of total		μ_{X_Z}=429.4 σ_{X_Z}=429.7 Md_{X_Z}=255 3% of total		μ_{X_Z}=476.41 σ_{X_Z}=415.30 Md_{X_Z}=480.5 3.36% of total
Y_Z: Single *defer* ZeroMQ		μ_{Y_Z}=0 σ_{Y_Z}=0 Md_{Y_Z}=0 max_{Y_Z}=0		μ_{Y_Z}=1 σ_{Y_Z}=0 Md_{Y_Z}=1. max_{Y_Z}=1		μ_{Y_Z}=1.21 σ_{Y_Z}=0.48 Md_{Y_Z}=1 max_{Y_Z}=4		μ_{Y_Z}=1.14 σ_{Y_Z}=0.6 Md_{Y_Z}=1 max_{Y_Z}=9

Table 3.1: I define X as the variable of the total number of updates that marked *defer* throughout the experiment. I define Y as the variable of the number of *defer* for a single update. I show the mean μ, standard deviation σ and median Md for both X,Y. I also show the percentage of updates that marked as *defer* from `consistency-enforcer` for the X variable and the max number of *defer* operations per update for the Y variable. Finally, the subscripts $_{D,Z}$ represent Dropbox and ZeroMQ back-end.

of 339.8 and median of 227. Additionally, the total number of updates that experienced a *defer* operation amounts to 2.6% of the updates.

Table 3.1 shows a summary for the *defer* updates that are delayed due to consistency operations needed for all four topology setups of our evaluation. For the Dropbox back-end I see that enabling the LANSync feature leads to less *defer* operations. This is true even in the case of the Global setups where no replicas are co-located in a data-center. For the ZeroMQ back-end, I see very different behavior when I introduce higher network distance between the replicas. For both Single and West setups the number of *defer* operations is close to zero and significantly than the other two setups (Global, East-West). This is expected because the main reason for the inconsistencies that I observe come from the propagation delays between the replicas and not because of the implementation of a third party service such as Dropbox.

The take-away from this experiment is that different back-ends, network topologies, and environments can dramatically affect the number of consistency anomalies seen by an application with the default (weak) eventual consistency. However, a `Shell`-enabled architecture give applications the ability to select stronger consistencies, which are maintained regardless of the environment.

This is important, because in a web service industry in which services are graded by the number of nines to the right of the decimal point, 10% or even 3% of problematic updates is substantial. Even relatively rare events are not rare at scale [55].

3.3.5 Update Visibility, Not Bandwidth

`Shell` decouples consistency from durability and replication through an indirection layer. This layer can add delays from both consistency interlocks, and from the computational overhead of processing incoming updates. Figures 3.10 and 3.11 and Table 3.1 I show a quantitative breakdown of this overhead. But this is not enough to understand the effect of this indirection on the actual application.

I define *update-visibility* to measure the effect of consistency-required interventions. I define `update-visibility` as $T_{apply} - T_{Rx}$, where T_{apply} is the time `Shell` applies the incoming update to application-visible storage, and T_{Rx} is the time when `Shell` first receives the update from the `cloud-manager`. Figure 3.12 shows, in log-scale, the `update-visibility` metric for each consistency protocol and back-end. For eventual consistency, the `update-visibility` is minimal with the 100% of the updates to experience delays less than 2 ms. This is consistent with the internal implementation of eventual consistency where `Shell` allows every update to reach the application as soon as it receives it. I observe similar behavior for monotonic reads, where the 98% of the updates reach the application within 1.4 ms. However, a few updates experience noticeable

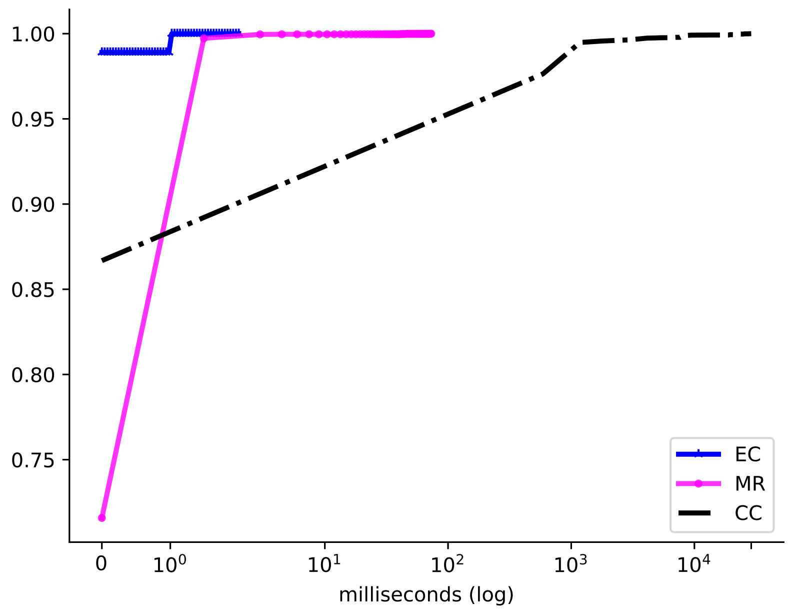

Figure 3.12: Dropbox: CDF (log-scale) of the `update-visibility` metric for eventual consistency, monotonic reads, and causal consistency.

`update-visibility` that can reach up to 70 ms. The reason for this extra delay is the additional wait-time needed to satisfy the consistency protocol.

In causal consistency, 87.5% of the updates experience `update-visibility` of less than 10 ms and 97% less than 600 ms. However, the rest of the graph shows that I have a heavy tail, and some updates can experience delays that reach up to 30 seconds. These measurements are tightly coupled with the behavior of the underlying storage layer, in this case Dropbox. I do not expect such high delays as these for other storage layers that can deliver updates faster and have higher throttling limits. But I do expect the same trend.

Indeed, the same trend appears in the case of the ZeroMQ back-end. Figure 3.13 shows the `update-visibility` metric for the ZeroMQ back-end in log-scale. For eventual consistency, 99.7% become visible within 1 ms. For monotonic reads, more than the 95% of the updates is visible within 3 ms, but a few updates experience higher visibility latency

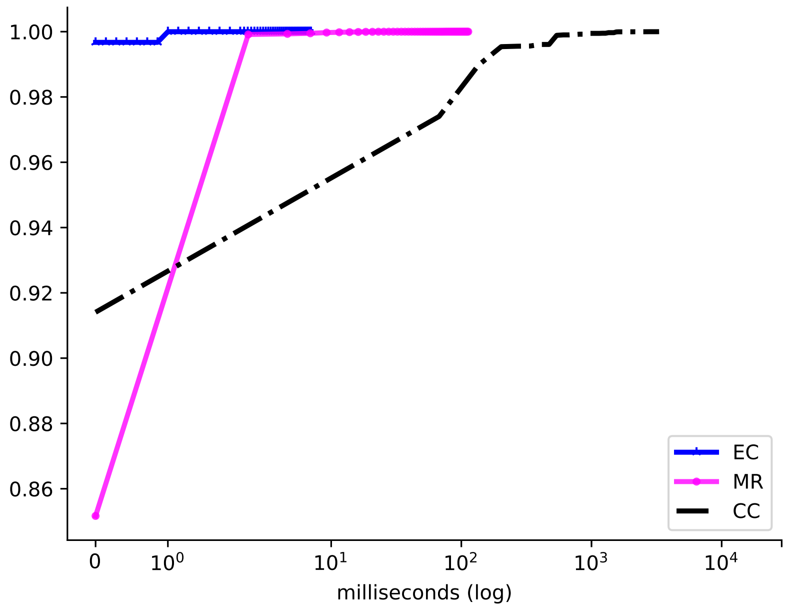

Figure 3.13: ZeroMQ: CDF (log-scale) of the `update-visibility` metric for eventual consistency, monotonic reads, and causal consistency.

that reaches up to 110 ms. Finally, for causal consistency, over 98% of the updates experience `update-visibility` latency less than 100 ms. However, for updates that `Shell` has to temporarily stage until it receives all the causally preceding updates, I see higher latency that can reach 3 sec. In the case of ZeroMQ implementation, I am not limited by a provider's throttling policies, and the highest `update-visibility` latency is one order of magnitude lower than in the case of Dropbox.

While `update-visibility` is a good metric to show how `Shell` affects an application, the operations bandwidth it can achieve is not a good metric. In the current `Shell` implementation, everything is asynchronous. `Shell` accepts every `PUT` operation, but does not apply it immediately. Every `GET` operation returns immediately as well, returning only data that is consistent. The only thing a bandwidth plot would show is the performance of the application storage, which in `Shell` is differentiated from the `cloud-manager`. In

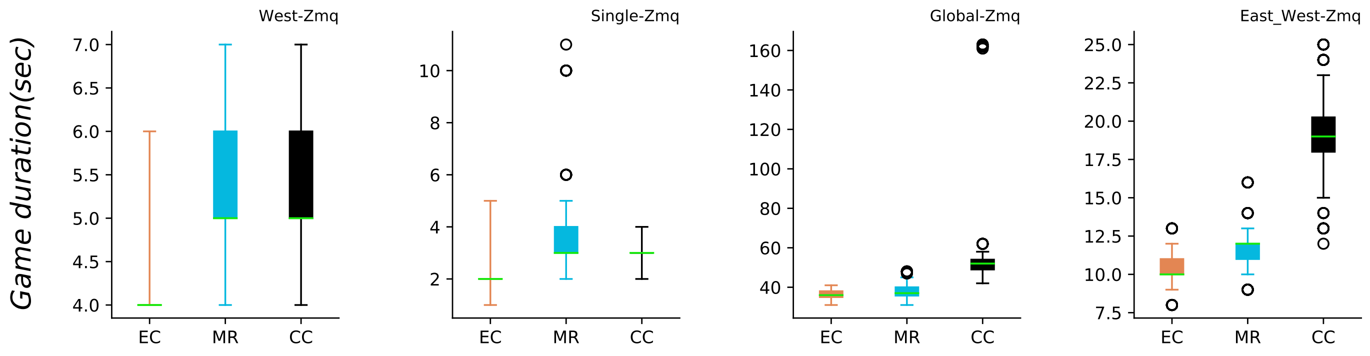

Figure 3.14: Box plots for the duration of replaying 30 game traces under different data center topology setups for the ZeroMQ back end.

the current implementation, the application storage is REDIS, which supports very high throughput [96]. For completeness, `Shell` can push only up to 8 updates per minute to Dropbox without receiving throttles. On the other hand, a single threaded `cloud-manager` for ZeroMQ achieved up to 100 requests per second.

3.3.6 Application Experience

In the previous section I analyzed how `Shell`, the storage layer, and the consistency protocol affect a single update. In this section, I examine the experience of the application as a whole. More specifically, how much time on average takes for an application to see all the events of a game.

Before the experiment I expected to observe the following: $Duration_{EC} < Duration_{MR}$ $< Duration_{CC}$ for both storage layers. The duration over the ZeroMQ verified our expectations. Figure 3.14 shows the box plots of the game duration. The X-axis shows different consistency protocols and the Y-axis shows game duration (seconds). Over eventual consistency consistently I see the best performance (shorter time to complete a trace). For the topology setups that introduce some network distance between the replicas (Global, East-West), monotonic reads outperforms causal consistency. This is expected because in causal consistency updates need to be in certain order before become visible in the application. The difference between eventual consistency and monotonic reads is not significant, and I believe that comes from the fact that replicas have to carry more meta-data (Figure 3.16) and some extra overhead to examine the version vectors before I apply the update. For all the setups most games complete within 80 seconds, but a few games in the Global setup reach 160 sec.

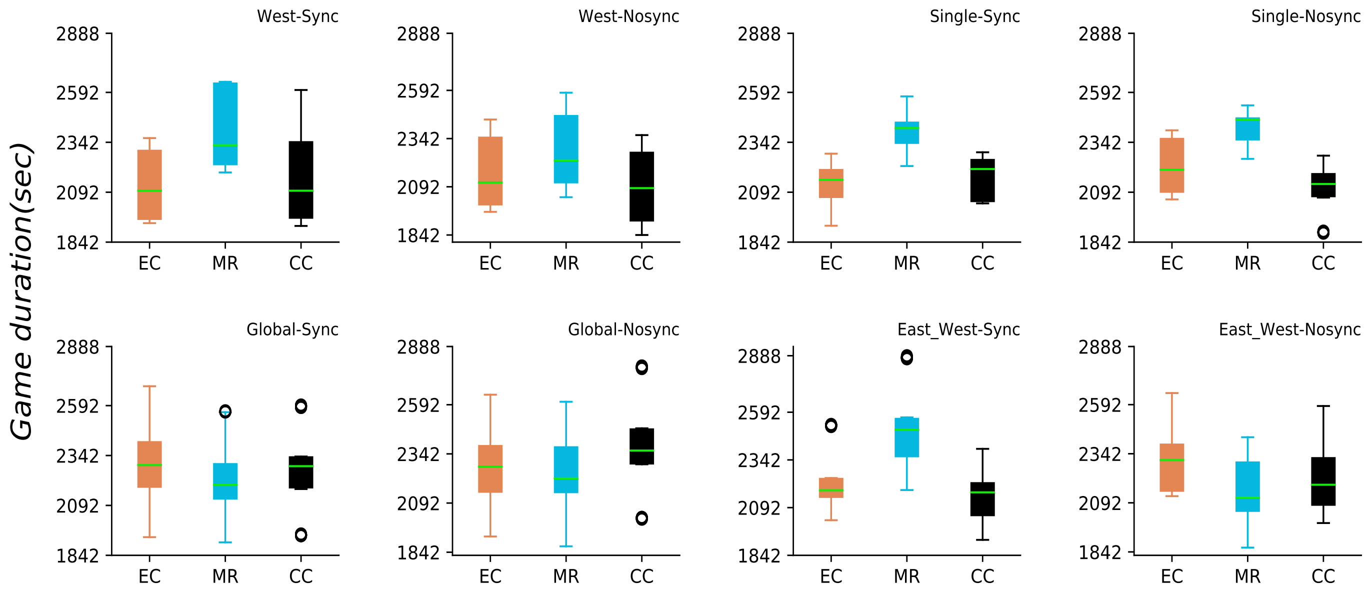

Figure 3.15: Box plots for the duration of replaying 10 game traces under different data center topology setups and storage layer configurations for the Dropbox back end. I show the game duration for each one of the four topology setups for both LANSync enabled, in the graph shown as LanSync, and disabled.

Surprisingly in the case of the Dropbox, I did not observe the expected behavior. Figure 3.15 shows the duration I observe over all the topology setups over Dropbox. At the top of every plot I show the topology along with the LanSync configuration. Plots with title *Sync*, show the duration when the LanSync is enabled. No protocol outperformed all consistently. More surprisingly, I see that for the Global topology, eventual consistency is as slow the other two protocols. In other cases, the higher delays come from monotonic reads, again not expected, especially when all the replicas are located in the same data-center and

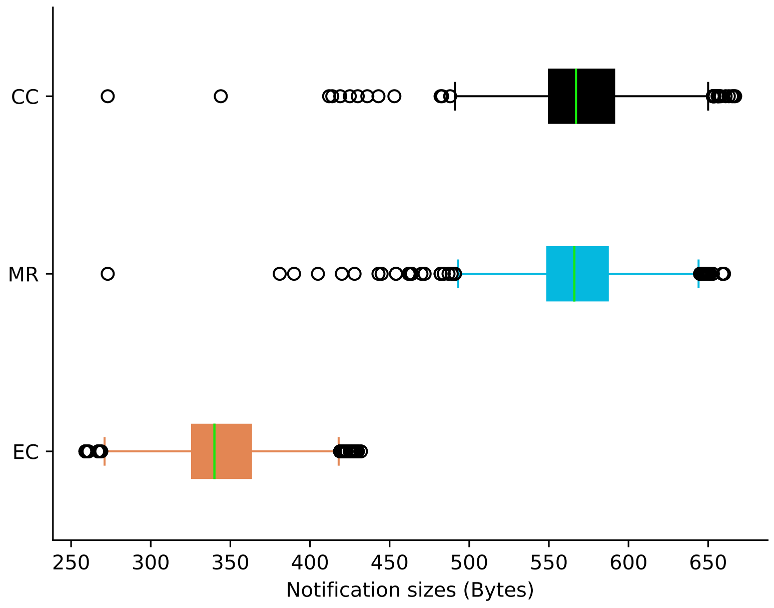

Figure 3.16: Box plots showing how the update sizes change for different protocols. In eventual consistency `Shell` sends only the data but in the other two protocols includes the necessary version vectors as well.

in the same availability zone (Single-setup).

I can only speculate why this is the observed behavior, because I do not have access to the internals of the Dropbox implementation, as I do in the case of ZeroMQ. First, the duration metric for Dropbox is two orders of magnitude slower than ZeroMQ. The reason for the is the very low write-rate that Dropbox allows before it starts throttling. The throttling mechanism returns a back-off delay that an application has to respect before any retry. Moreover, in contrast with ZeroMQ, the Dropbox communication pattern is different. The updates for the most part travel to the Dropbox server and then are propagated in the rest of the replicas. One would expect that the LANSync feature would accelerate the trace replay, especially in the Single-setup.But having all the replicas in the same availability zone is not the same as having all the replicas in the same LAN.

The Dropbox back-end shows what most cloud based applications face today. Applications do not have access to the internals of the providers' implementations and thus, have to face unexpected behavior similar to what I observed in this experiment. When it comes to data consistency, unexpected behavior usually leads to inconsistent data. The `Shell` project advocates that *defensive designs* can *isolate unexpected behavior* and make it easier for the application to satisfy its semantics. This is what `Shell` offers. An application that uses `Shell` over Dropbox would get better experience than an application that is not. But the point of this experiment is not to compare the two admittedly different purposed data stores, nor to show that for Dropbox I found an anomaly where causal consistency can complete a traced faster than eventual consistency. I do not expect to see in the general case of high-throughput data-stores.Rather I show that when placing an application above the storage layer of a third party provider, the application *inherits characteristics of the provider's implementation* which might not be always in favor of the application needs.

For completeness, Figure 3.16 shows the size of updates that `Shell` exchanges between replicas. The updates include both data and meta-data. The meta-data is necessary for the implementations of various consistency protocols. For eventual consistency, since this is our baseline and is not needed, I do not include any meta-data; I only pass around `Shell` SOs without populating the `Shell` version. On the other hand, in the other two protocols I include version vectors which leads to a significant increase of the update size. The increase accounts to almost 60% when compared to eventual consistency. Given that our application messages are relatively small, this overhead is significant. Notification overheads remain the same for both back-ends since the messages I send to every replica do not change format.

3.4 Summary

In this chapter I presented `Shell`, an architecture that provides an abstraction of cloud services. `Shell` translates abstract method calls to service-specific calls, allowing application code to remain unchanged while running on different underlying cloud services.

Further, `Shell` is able to provide strong consistency guarantees customized to the application needs, even if the underlying provider supports only weak eventually-consistent guarantees. Strong consistency is guaranteed by regulating the flow of updates reaching the application from the cloud storage service.

The ability to separate application code from provider implementations allows them to evolve independently. This freedom allows application developers to focus solely on feature development while keeping the application free from code dealing with specific provider APIs or issues. Moreover, `Shell` also isolates the consistency protocol implementations, which no longer have to evolve as quickly as the application code.

Our experimental evaluation relies on two very different communication layers. Dropbox, used as a communication layer, provides only low, throttled throughput, and no access to layer internals. However, it does provide durability guarantees based on its use as a file-sharing service.

The ZeroMQ library is not a data-sharing service at all, only a communication channel. As such it provides no durability or consistency guarantees. Messages, once delivered, are forgotten. However, our ZeroMQ-based layer on top of the communication library has high throughput and low latency, and differs from the Dropbox layer only in implementation of the `cloud-manager` module, not in any consistency-related code.

In both cases I evaluated system performance across three consistency protocols, using eventual consistency as the baseline. Both back-ends delivered misordered updates to the application in their native, eventually-consistent state. System performance differed under the two layers. The ZeroMQ layer was faster, and the Dropbox layer had inconsistent performance, especially across different consistencies.

Despite these differences, `Shell` is always able to provide causal guarantees without requiring the application to participate in dependency tracking. `Shell` cannot provide cross-provider performance guarantees. However, the separation of application from provider-specific code allows the underlying service to be transparently replaced with another service, potentially one that provides performance that is better tuned to a specific application's needs.

The basic principle behind this work is that of isolation between distinct layers. Other proposals [14] that use layering require applications to explicitly provide causally-preceding write identifiers. While this approach may give precise information, it forces the application developers to intermingle consistency-related and application-specific code, while the latter rapidly evolves. This approach can lead to errors.

Cloud-supported applications are now the norm. Current approaches mean that choices made by service-providers inevitably influence the evolution of applications built on top of their services. `Shell`, by contrast, allows applications to develop independently from constraints imposed by any specific provider, allowing mobility across consistency levels, and across providers.

Chapter 4

Topology independence and Information Leakage in Replicated Storage Systems

A user may collaborate and share data with many other users and across many devices. Replicated storage systems have traditionally treated all sharing equally and assumed that security can be layered on top. On the contrary, I show that security needs to be integrated into the consistency and replication mechanisms to prevent information leakage.

The traditional assumption in such systems is that all interacting users trust one another. In the case of Topology Independent replication systems, this assumption can severely limit performance: if users only perform anti-entropy sessions with those they trust, then updates can only propagate as quickly as trusted users encounter one another. It would be far more efficient to use untrusted users to "ferry" updates between trusted ones. In the case of cloud-based replication systems, users are required to trust the third-party cloud provider.

In this chapter, I present *T.Rex*, a replicated data system that achieves various forms of consistency *without requiring all interacting users to trust one another*. At first glance, it may appear that simply encrypting the data is enough. While this prevents any information leakage from the data itself, it does not prevent leakage from the *meta-data* that users share during anti-entropy sessions. As I discuss, this meta-data can reveal user trust relationships, behaviors, and update frequency.

T.Rex uses role-based access control [42] to enable flexible and secure sharing among users with widely varying collaboration types: both users and data items are assigned *roles*, and a user can access data only if they share at least one role. Building on top of this abstraction, *T.Rex* includes several novel mechanisms: I introduce *role proofs* to prove role membership to others in the role without leaking information to those not in the role. I introduce *role coherence* to prevent updates from leaking across roles. Finally, I use Bloom filters as *opaque digests* to enable querying of remote cache state without being able to enumerate it. I believe these to be generally applicable beyond replicated data systems.

I combine these mechanisms to develop a novel, cryptographically secure, and efficient anti-entropy protocol. To demonstrate its general-purpose use, I have built several different consistency schemes on top of it, including eventual consistency [113], PRAM [73], causal consistency [4], and causal+ consistency [74]. I have implemented *T.Rex* and these consistency schemes, and with an evaluation on a local testbed, I demonstrate that it achieves security with modest computational and storage overheads.

The rest of this chapter is organized as follows. Section 4.1 describes our model and security goals. Section 4.2 describes *T.Rex*'s overall design. Section 4.3 describes the implementation, and how this implementation meets security challenges. Section 4.4 covers the experimental evaluation of *T.Rex*, Section 4.5 describes application of our approach to cloud services, and conclude in Section 4.6.

4.1 System model

T.Rex is a storage management system designed to manage user and application data on personal or cloud servers and mobile devices. Every participating device or server acts as a replica that eagerly replicates all or part of the total data collection.

T.Rex sharing is defined and constrained through interaction between possibly overlapping *roles*. A role consists of a unique name, a secret key, and the *role predicate*. The role predicate is defined over per-file meta-information called *labels*. A predicate for the collection of tax documents might be `filetype=pdf && context=taxes`, for example. A role defines data collections, via the predicates, and access groups, via the group of devices that participate in the role.

Replicas periodically push data (created either locally or received from other replicas) to other replicas through one-way *anti-entropy sessions* [113]. The choice of destination could be random, or determined by availability, stability, or available bandwidth. Data is eventually relayed from where it is created to all interested replicas through a succession of such anti-entropy sessions.

The system as a whole contains many devices and users. We distinguish between *T.Rex replicas*, those running the protocol, and *cloud replicas*, which replicate cloud data under the control of cloud service providers. We assume that devices each play only a small number of roles ($<$10), and are single-user, with the usual convention of a multi-user machine being treated as multiple virtual devices. Each device hosts a single replica, so we use the two terms interchangeably, depending on context.

4.1.1 Security goals

We assume Byzantine failures [68]. Replicas may maliciously and arbitrarily deviate from correct protocol execution in an attempt to subvert the protocol. By contrast, software running in cloud services is often assumed to be "honest-but-curious" [46]. Such replicas would follow the protocol honestly but might analyze the protocol messages to infer information about users and their data. However, given the multitude of reports of cloud services inadvertently exposing data [35, 62, 63, 79, 84, 101, 111], together with recent disclosures of national intelligence agencies compelling data disclosure, there seems little reason to differentiate the cloud failure model from that of the standard *T.Rex* Byzantine model.

Our security goals are to provide data confidentiality, data integrity, and to prevent information (including metadata) leakage. More specifically, we attempt to prevent the following types of information flow:

- *replica or user roles* - Other than roles common to both, replica r_i should not be able to learn which roles r_j plays.
- *data written in object updates* - Any data written in the context of role r_i should not be visible in plaintext to a replica that does not play role r_i.
- *role activity* - For roles that it does not play, a replica should not be able to identify the roles of encrypted data. As a simple example, `Alice` may wish to avoid dark corridors if she detects heightened activity in the `Down-With-Alice` role, even if the precise nature of that activity is unknown.

4.2 T.Rex design

This section provides a high-level overview of the *T.Rex* system. A *T.Rex* "group" consists of a set of devices, possibly owned by many users, each with a *T.Rex* replica. Replicas interact in the context of a set of roles defined by a single, distinguished *role master*, though this responsibility can be delegated. The role master defines roles and their keys, maps rights and capabilities onto roles, and maps roles onto users or devices. The role master also generates and disseminates certificates for the public keys of devices and users. Interaction between groups (different role masters) is possible,but they share in the context of a single group. If `Alice` in group `Wonderland` wants to share data with `Han`

`Solo` in group `Star Wars`, either `Alice` must join `Star Wars` or `Han Solo` must join `Wonderland`.

The role master is also responsible for key revocation. We follow an approach similar to SPORC [41]. A replica is removed from an existing role by creating a new *update-role-key* message signed by the role master. This message contains a new role key and a timestamp, encrypted with the public keys of the replicas that remain in the role. Replicas removed from the role can still see old updates but not those created after the update-role-key message.

4.2.1 Replication: State, Updates and Policies

Replicas interact through a one-way push-style synchronization protocol [33], also known as an anti-entropy. To be accurate most of the times anti-entropy refers to pull-style protocols but we use this term here as well. A *source* replica initiates the synchronization with an arbitrary *target* replica, pushing data seen by the source and not by the target.

Each replica has a global unique ID named *RID*. Each replica also has a *clock*, which is a counter that is incremented each time a local object update occurs. Every object in the replica has a version consisting of the triple `<OID,RID,clock>` where `OID` is a unique ID for the object. Each object also has a set of key-value attributes defining the object's labels. Replicas also have version vectors, which describe the latest updates seen from the rest of the systems replicas.

A data item consists of two parts: meta-data (label, size, permissions) and the actual data of the item. Each object stored by a replica must have labels that satisfy at least one of the replica's role predicates.

The time between two anti-entropy sessions is called the *update period*, During the update period, a replica merely records object updates created locally. Updates include object creations, deletions, and modifications, and object label definitions and deletions.

At the end of an update period, a replica gathers the actions created in the previous period and creates two types of per-role entities: *kernels*, which contain meta-information describing the actions of the previous period, and *shards*, which include the same meta-information as well as any actual update data. The kernel contains a unique GUID, a role ID, connection information about the replica, and the version vectors of the replica at the beginning and the end of the current period.

Update periods are always non-overlapping, as consistency protocols are otherwise quite complex and can duplicate information flow (Section 4.2.2). We enforce this invariant by ending the current periods when data is requested by another replica or when meta-data arrives from another replica in an anti-entropy session.

A replica's *update policy* determines whether it will send kernels or shards when communicating with other replicas. These choices are analogous to choosing an invalidate or update policy in a shared memory multiprocessor. Currently, *T.Rex* supports three policies. Under the *kernel* policy replicas exchange only kernels (meta-data). A replica locally invalidates copies of objects specified by incoming kernels, requiring the corresponding data to be demand-fetched before they can be used again. The *shard* policy sends both where data and meta-data together. Finally, *kernel-shard-based* during which the source replica of the update sends shards, data and meta-data for the roles that shares with the target replica and only kernels (meta-data) for the rest.

4.2.2 Ramifications of consistency in *T.Rex*

T.Rex implements several different consistency protocols, including eventual consistency [113], PRAM [73], causal consistency [4], and causal+ consistency [74]. The choice of consistency protocol is not strictly relevant to this work. More relevant is whether these consistency protocols are supported at the object or role level.

Users might creating overlapping roles. This choice is necessary, as roles are defined by high-level user-defined predicates. Ensuring that logical predicates are non-overlapping is difficult, and may not always be possible.

Consider a replica, r_i, that plays two roles: $role_x$ and $role_y$, and suppose that the intersection of $role_x$ and $role_y$ is object o_a. If r_i receives an update for object o_a in the context of $role_x$, a straightforward implementation would immediately apply it to the local copy of o_a, assuming consistency requirements are met. However, if r_i now reads o_a in the context of $role_y$, it sees the altered value. Worse, its copy of o_a is now different from copies on replicas that only play $role_y$, and *the distinct versions will never converge*. This violates the base consistency model supported by nearly all mobile storage systems, eventual consistency, which requires that all copies of the same object *eventually* converge.

One way to support eventual consistency is to transfer the update from one role to another. However, this transfer raises several questions. First, might two replicas playing both $role_x$ and $role_y$ each transfer the same update? How are the two resulting updates to be recognized as the same?

Second, a model like causal consistency demands that if update u_1 is read at a replica before u_2 is created, u_1 *happens-before* u_2, and must be applied before u_2 anywhere u_2 is applied. If updates are transferred from one role to another, the transferred updates must obey the same consistency constraints. However, this effectively implies that two updates of $role_x$ might be ordered only through updates that were created in an entirely different role. The correctness criteria in such situations is not clear. Further, additional ordering constraints can also affect performance.

A more damning critique is that moving updates among roles can be seen as a leak of information from one role to another. Users should have the option of allowing updates to leak across roles, but the system should be able to enforce the stronger semantics.

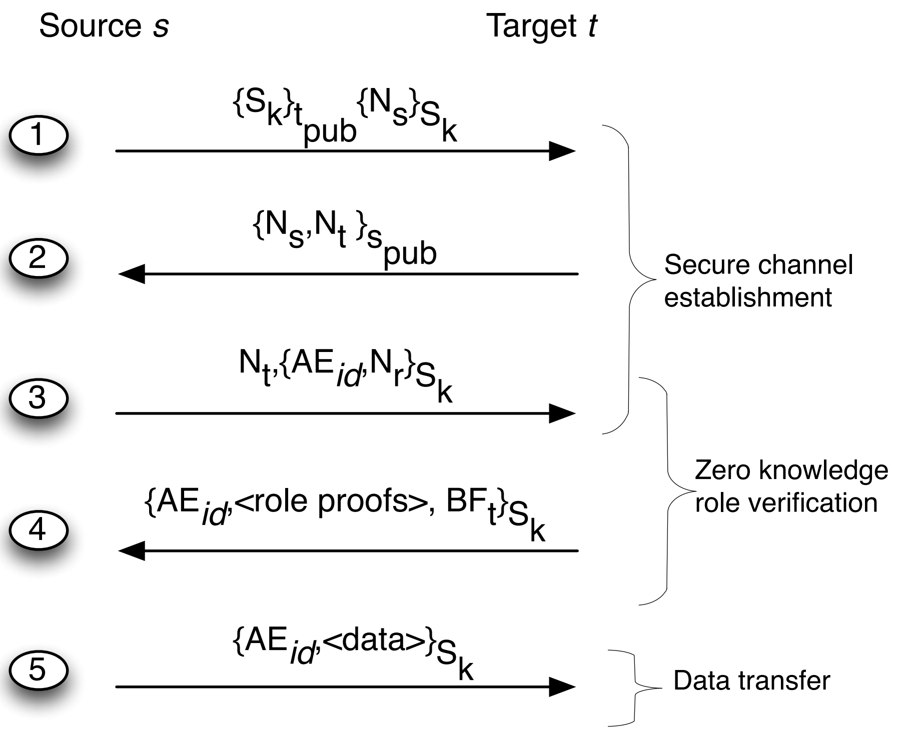
Source s
Target t
1
$\{S_k\}_{t_{pub}}\{N_s\}_{S_k}$
2
$\{N_s, N_t\}_{s_{pub}}$
3
$N_t, \{AE_{id}, N_r\}_{S_k}$
4
$\{AE_{id}, \text{<role proofs>}, BF_t\}_{S_k}$
5
$\{AE_{id}, \text{<data>}\}_{S_k}$
Secure channel establishment
Zero knowledge role verification
Data transfer

Figure 4.1: *T.Rex*-Sync

For all of these reasons, we take a second approach: splitting any object that is updated in the context of more than one role into two physical representations, effectively forking the object's version history.

4.3 *T.Rex* implementation

This section describes protocol and mechanism detail of our implementation. *T.Rex*'s protocols differ from that of "personal" sharing systems [43, 89, 91, 94, 102] in the use of cryptographic primitives, extra handshakes for cryptographic challenges, and in guiding consistency transfers through *opaque digests*.

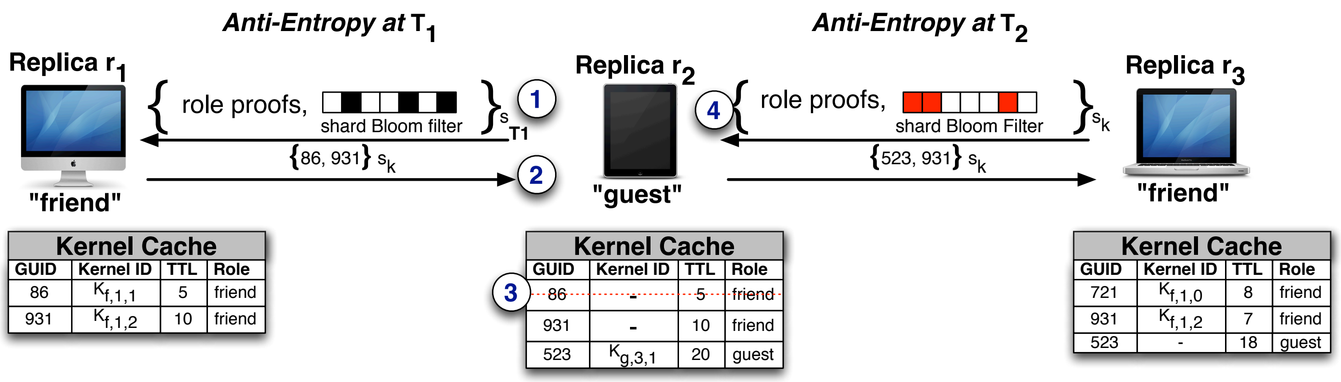

Figure 4.2: **Anti-entropy:** Replicas r_1 and r_3 communicating through r_2, which is of another role. $K_{r,x,y}$ refers to the kernel in role r for period y of replica x. For r_1 to push data to r_2, it first sends a request to r_2 (not shown). (1) r_2 responds with proofs describing roles played locally, and a Bloom filter describing locally cached kernels. (2) r_1 responds with new encrypted kernels not represented in the Bloom filter. Only the GUIDs (86, 931) of $K_{f,1,1}$ and $K_{f,1,2}$ are visible to r_2, the rest is encrypted. (3) After this exchange, r_2 locally makes a decision to drop kernel 86 because the TTL expires, and (4) r_2 initiates anti-entropy with r_3. The replication update then repeats between r_2 and r_3. r_3 is able to decrypt kernel $K_{f,1,2}$ but not apply it, as it is missing $K_{f,1,1}$ (GUID 86) (see Section 4.2.2).

4.3.1 Replication and information leakage

A conventional replication protocol handshake consists of the target replica t initiating a session with a request for the target's version vector. The source replica compares its version vector with the target's vector to determine the set of updates that should be sent to the target.

T.Rex's handshake is designed to make this determination without allowing either the source or target learn information data held by the other, except for roles in common. The first three messages of the handshake shown in Figure 4.1 are used by the source and target to authenticate each other. However, the third message also contains the role challenge and session ID (AE_{id}), and the fourth allows the target to respond with role proofs. The challenge is just a random nonce. A proof is an HMAC of the nonce and the *role key* (rk_i is the role key for role i): a per-role key shared by all devices that play that role. Aside from

HMAC proofs, the key is also used to encrypt data and updates for that role[1]. A device that plays multiple roles will return a proof for each role. The source compares the target's proofs against locally created proofs to identify roles in common.

The challenge and proofs allow the target to prove it plays specific roles, but only to other replicas that also play those roles. A source replica does not learn of any target role that it does not also share. This is a fundamental difference between *T.Rex* and existing anti-entropy protocols, where targets completely summarize their state for arbitrary sources in the very first message.

The fourth message also contains the target's Bloom filter. We term these Bloom filters *opaque digests* to point out that the Bloom filters have security value. The Bloom filter is constructed with a cryptographic hash, and values in the filter are kernel GUIDs chosen uniformly at random from a large (128-bit) space. The source gains no information about any kernels present at the target unless it has those kernels as well. This Bloom filter summarizes IDs of kernels known to the target, much like a summary cache [40]. The source queries the Bloom filter for kernel IDs cached locally, sending to the target any not contained in the filter.

Bloom filters are used to obfuscate data cached by the target. Kernels are encrypted to prevent information leaking to other roles. In more detail:

> *An encrypted kernel reveals neither the role in which it was created, nor the local clock value when it was created, nor the replica at which it was created.*

This *update anonymity* means that a target only learns explicit information about kernels, or updates, of roles shared with the source. Replica i does not know which kernels it has seen from replica j, as kernels created by j for roles not shared by i will be encrypted.

[1]Our implementation currently uses an AES key as a role key, but in the future we may use a public key, with hybrid encryption for confidentiality.

A target can only describe its state by enumerating all kernels seen locally, or by using some sort of summary data structure. An enumeration is not bandwidth-efficient, but more importantly would reveal explicit information about the target. A listing of kernels cached at the target reveals its *target policy* (how it chooses partners for anti-entropy), and *caching policy* (which kernels to cache for forwarding to other replicas in subsequent anti-entropy sessions). A replica's local caching policy will often prefer kernels of roles it shares. Therefore, a snapshot of cached kernels at replica i, when compared with a snapshot of cached kernels at replica j, could reveal commonalities. These commonalities, in turn, could show that either the target or cache policies of the two replicas are similar, giving good evidence that the replicas share at least one role.

The final step of the protocol is data transfer. The source identifies roles in common with the target by comparing role proofs. Call this set of common roles R_{com}. Kernel $k_{r,i,j}$ is the j^{th} kernel created by replica i for role r. All kernels $k_{r,i,j}$, such that $k_{r,i,j} \notin BF_t$ and $r \in R_{com}$, can be sent to the target without extra encryption by the role key rk_i, since both source and target share role $role_i$. The data will still be encrypted with the session key before it is sent out over the network. Kernels for roles not common to both the source and target are doubly encrypted: first with the role key, and again with the session key.

Kernels not common to both the source and target still might be in a role played by the target. The target therefore must try to decrypt the header of each unknown kernel with each of it's keys. This is not a large overhead given our assumption that devices play only a small number of roles.

Figure 4.2 shows the outline of two complete anti-entropy sessions in *T.Rex*. Replica r_1 initiates a session with r_2, with which it shares no roles. Two kernels are sent to r_2, and at some future point in time r_2 makes a local decision to drop kernel 86 from it's cache. Hence, kernel 86 never makes it to r_3. Replica r_3 will discover this omission when it later learns of subsequent updates in the same role, or by the same device.

4.3.2 Role Coherence and object forking

We term the object-forking approach *role coherence*. Objects are initially created tagged with all roles in which they are visible. An object updated in the context of one role is forked into a version history for that role, and a version history for all other roles that can see the object. At the limit, an object visible in *n* roles is forked into *n* logically distinct objects, each of which may continue to be updated. Inside a role, all of the consistency schemes work as before. This approach is similar to that of Qufiles [118], in which a single file can have multiple physical instantiations, transcoded for different bandwidth requirements.

Role coherence requires accesses to be associated with a single role. Updates from remote replicas are carried in role-specific containers (shards), and therefore have explicit role associations. An early version of *T.Rex* handled local reads and writes by passing the role context through the POSIX file system interface as a modifier on the base filename, as in "`filename@role`". We currently handle local reads and writes by defining an interface that allows applications to specify a current role, and use this role to implicitly tag object reads and writes. Labeling could also be done automatically [106] or by using the hints of a provenance tracking subsystem [80].

Object forking creates overhead in both metadata and object data. A straightforward implementation of the underlying object-forking storage mechanism would fork a new copy of an object with each incoming update, causing storage overheads to increase rapidly with the number of roles.

Instead, we represent objects as *recipes*, which are ordered lists of block hashes [45, 115]. An object's data consists of the concatenation of the blocks from the recipe, in order. Instead of replicating the whole object to create a new version, we only update the recipe (metadata) of the object to include the new blocks, and save only modified blocks to disk.

4.3.3 Attacks

In this section we describe how *T.Rex* defends against a variety of attacks.

Passive eavesdropping: Anti-entropy sessions use public keys to establish secure channels. The secure channel is established before replicas reveal any information about replica identity or state.

Impersonation: Session establishment is secure against replay attacks because of fresh nonce challenges. An attacker could replay the first message from an earlier exchange. However, replaying the third message would fail because the target's challenge nonce in message two will change.

Role Leakage: In this family of attacks a malicious replica tries to find role commonality among two other replicas. Trudy could initiate anti-entropy sessions with both Alice and Bob, using the same role challenge, N_r, with both. Both would return vectors of role proofs (message 4 in Figure 4.1), one for each role they play. Trudy could determine how many roles the two play in common by comparing the proof vectors.

T.Rex prevents this attack by tying the challenge to the current session. N_r is defined as being equal to $N_s \oplus N_t$.

Activity Inference: We define activity inference as a replica learning about updates in roles that it does not share. Assume Trudy and Alice participate in an anti-entropy session, but do not share any roles. The protocol should allow Trudy to update Alice without Alice revealing anything about her state, or what she knows about other replicas. This is a *fundamental* difference between *T.Rex* and other anti-entropy protocols, which require the target of an update to summarize its state to the source. *T.Rex* lets Alice summarize state through an opaque digest, revealing nothing about updates unknown to Trudy.

Drop out-of-role updates: Trudy could discard all updates that do not belong to her roles. This will not affect either liveness or correctness, but it will impact performance.

4.3.4 Freshness

We use Bloom filters to compare replicas' cache contents without revealing unnecessary information. Bloom filters may be efficiently queried without allowing enumerations, but they can grow large if they contain many objects, or are required to have low false positive rates. Our Bloom filters are currently structured to have false positive rates of less than or equal to 0.5%.

Constantly adding newly-seen kernels to a Bloom filter with a constant false positive rate would result in ever-increasing filter sizes. We bound this growth by defining a *freshness* (f) interval. The target is constrained to include in his Bloom filters all cached kernels not older than the freshness interval, and kernels older than the freshness interval are dropped from the cache once they have been applied locally.

Freshness allows us to bound Bloom filter growth, but admits the possibility that kernels may not survive long enough to be propagated everywhere. *T.Rex*'s consistency module eventually detects these events and requests the shards directly from the creator, which is guaranteed to hold onto locally-created shards for some tunable, but long, period of time.

For example, a replica with f=10 is interested in updates that have been created (or received, since replicas can act relays) from the source during its last 10 anti-entropy periods. If a target is tuned to initiate one anti-entropy session each minute, then f=10 means that the replica is interested for updates that are at most 10 minutes old. In other words, freshness is a metric of time, defined as the number of anti-entropy sessions since an update was created, or received. Systems in highly-connected environments might use a low value of freshness, since high connectivity allows them to quickly learn of newly created updates. On the other hand, devices and systems with low connectivity might use higher values of f. By default, freshness is a per-device attribute, but *T.Rex* also supports per-role f values.

4.3.5 Prototype

T.Rex's primary interface is the POSIX file system interface, using FUSE to bind our user-level servers to the Linux kernel's VFS interface. The *T.Rex* prototype consists of approximately 24,000 lines of C code compiled with gcc-4.6.3. The prototype uses Google protocol buffers 2.5 [117] to serialize messages exchanged between replicas, and the ZMQ-2.2.0 [60] networking library to communicate. We use libTomcrypt-1.17 [34] to implement all cryptographic operations, with 160-bit SHA-1 hashes, 256-bit AES for payload encryption, and 2048-bit RSA keys. Metadata is stored in sqlite3. We also use several data structures provided by the UTHash library [116].

T.Rex is divided into a set of high-level, communicating modules. The *net* module handles all network communications, and implements *T.Rex-Sync* logic. The *consistency* module checks consistency-related prerequisites of incoming kernels and applies or blocks them. The *sk-factory* module produces and manages the storage of shards and kernels, both locally and remotely created. Finally, the *FS* module provides a file system interface for the system by plugging into the kernel's virtual file system (VFS) layer.

In more detail, an incoming kernel is passed by the *net* module to *sk-factory*. If the kernel matches a local role, *sk-factory* will decrypt it, verify it, and pass it to the *consistency* module. Kernels that do not match any local role are sent to disk. The *consistency* module checks whether the kernel can be applied without violating consistency invariants. If so, each non-stale (overwritten by a logically later update) update contained in the kernel is applied to the local object store. If not, it is stored on a *pending* queue until later kernels unblock it, or supersede it. Updates are generally applied by replaying appropriate file system operations from the *FS* module. Finally, the state of the replica is updated in the database.

Outgoing kernels are created by *sk-factory* at the start of an anti-entropy session (Section 4.3.1). The *net* module, implementing *T.Rex*-Sync, determines whether each kernel should be pushed to the target.

4.4 Performance evaluation

Our goal in building *T.Rex* is to explore the potential for eliminating information leaks in replicated storage systems. Sections 4.2 and 4.3 discuss new functionality in our replication and consistency protocols; this section quantifies the cost of supporting that functionality.

We evaluate three categories of overhead. First, CPU overheads arise from the extra cryptographic operations used for authentication and confidentiality. Second, our protocols include extra authentication messages, and potentially send duplicated data because of update anonymity. Finally, the new functionality adds space overhead both because of new data structures, and because of duplicated data across roles.

We drive our evaluation through a data-set (Figure 4.3) modeled on a large collection of images from the online picture-sharing site *500px* [51]. There are 18,315 files, with mean size of 358.1 KBytes and median size of 335.1 KBytes.

4.4.1 CPU costs

The goal of our first experiment is to compare *T.Rex*-Sync with existing synchronization protocols. Existing protocols roughly follow the same approach. The replicas exchange version vectors to inform each other of updates they have seen. Next, the source uses the target's version vector to determine which locally seen updates should be sent to the target. Our first experiment compares *T.Rex*-Sync with a stripped down version of *T.Rex*-Sync called `trad-sync`, which models the conventional anti-entropy approach.

We use two replicas, residing on different machines and communicating over a 100-

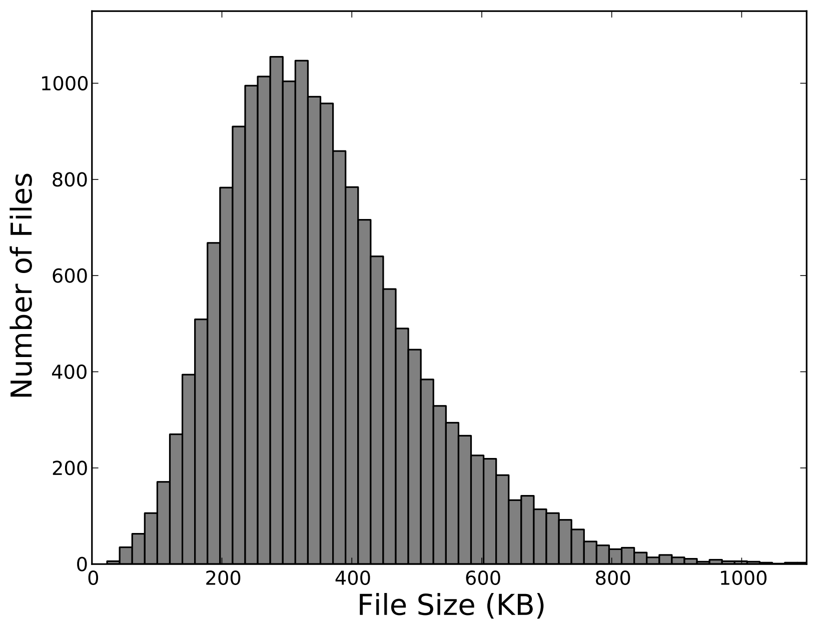

Figure 4.3: Data-set modeled on online image sharing site. There are 18,315 files, with mean size of 358.1 KBytes and median size of 335.1 KBytes.

Mbit local area network (LAN). During each 60-second period, the source selects 10,000 files from the initial dataset and performs random data and metadata updates. The source then initiates an anti-entropy with the target. The total duration of a single run of this experiment is five hours. We vary the number of roles, R, that the two replicas play, where $R \in [1,32]$. For each R we performed three runs, each using a different distribution of updates in update periods. The distributions we used are *uniform*, *zipf* with $s = 2$, and *Poisson* with $\lambda = 50$. We present results only for the *uniform* case, as the other results were qualitatively similar. Both machines run Linux Ubuntu 12.04 with an Intel i5-7502.67 GHz for the source and an Intel Core 2 Duo-E84003.00GHz for the target.

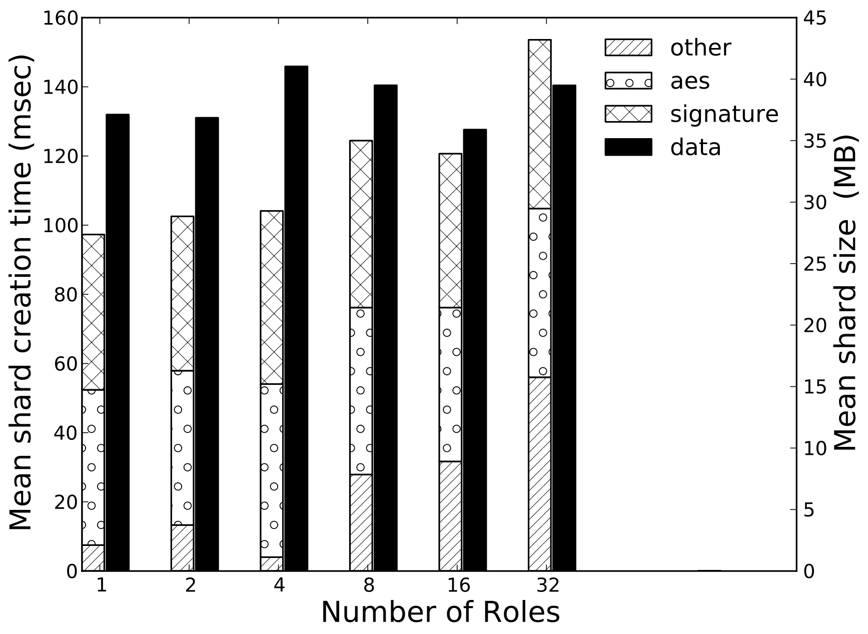

Figure 4.4: Shard creation CPU costs, and shard sizes.

Shard creation

Figure 4.4 shows the average data exchanged over the course of the 300 anti-entropy sessions, with varying number of roles. The average shard is 40MB in size, and requires approximately 100ms to be created for runs where the replicas participate in 4 or fewer roles, or 120ms or more when R is higher. The extra cost is caused by larger R values increasing the percentage of updates that must be role-encrypted. A creation time of 100ms to 140ms is large, but is amortized across 10,000 distinct updates, 4k bytes each.

As shown by the left bars, roughly comparable portions of shard creation time are spent on public-key encryption (the initial handshake, plus signature generation and verification for shards), and AES encryption (all outgoing and incoming data). The remaining time, marked ‘other’ on the figure, includes disk accesses to store shard data and metadata, serialization of messages to and from protocol buffers, and miscellaneous copying. This latter

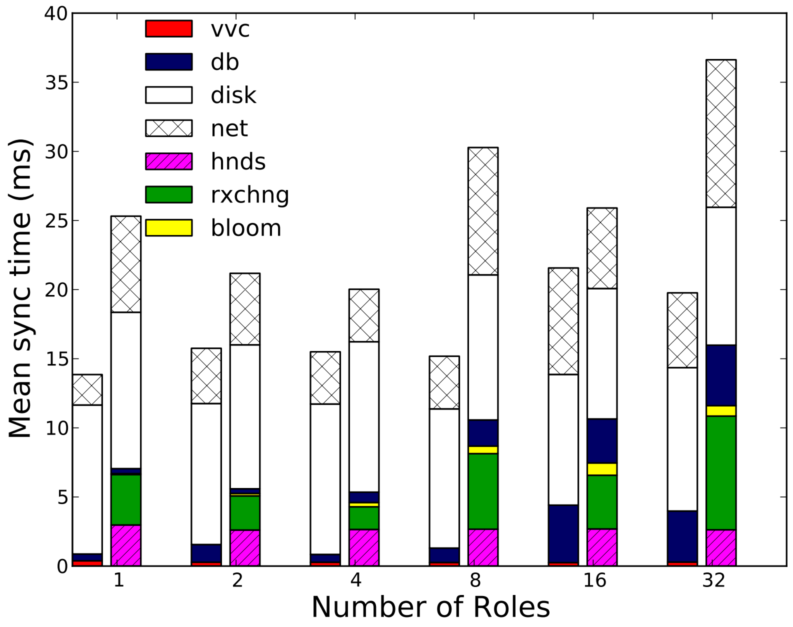

Figure 4.5: **Overhead breakdown** Traditional synchronization protocol (`trad-sync`) on the left, *T.Rex*-Sync on the right.

category increases linearly as a function of *R*. The public-key overhead is constant.

Anti-entropy

Figure 4.5 shows the average time needed to complete a single anti-entropy session for *trad-sync* (the bar on the left) and *T.Rex-Sync* (the bar on right). *Version vector check* ("vvc") is the time spent by the source in checking for shards needed by the target, as indicated by the target's version vector. This cost is associated only with *trad-sync* since *T.Rex* uses Bloom filters to determine this information. *Database* ("db") is time spent by the source in loading all the metadata needed to construct shards. *Disk* is the overhead of loading the shard's actual data from the disk. *Network* ("net") is the aggregate time-on-the-wire network costs throughout an anti-entropy session, *minus* the network latency of sending the shard data. The actual sending of data can happen asynchronously (if using

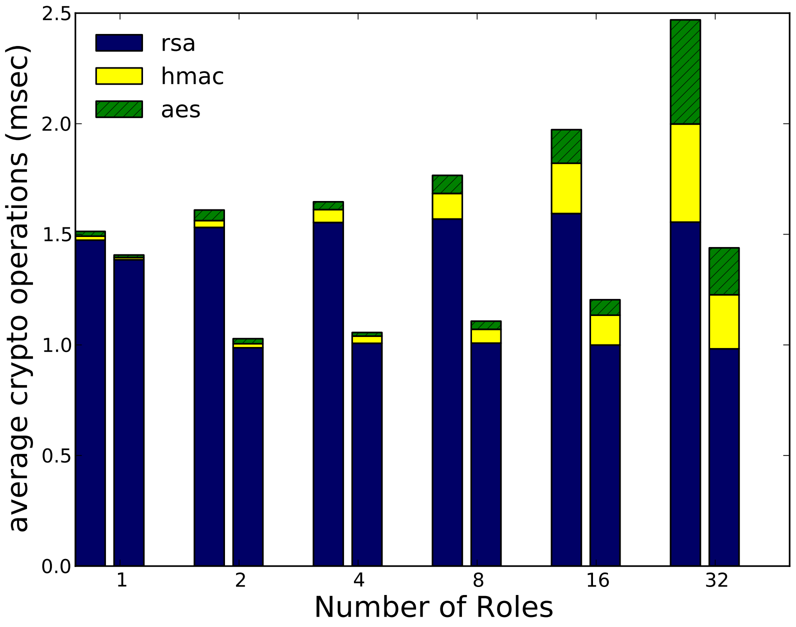

Figure 4.6: **Cryptographic overhead:** Costs of cryptographic operations. Bars on the left are for the source, and on the right for the target.

kernels rather than shards), and is therefore not strictly relevant here. Note, however, that the cost of sending 40MB of updates dwarfs all of these costs, even that of the public-key crypto. *Handshake* ("hnds") is the time to create the first two messages of the handshake in Figure 4.1. This cost is dominated by the creation and verification of the public-key challenges, and only exists in *T.Rex*-Sync. *Role exchange* ("rxchng") is the duration of the role-exchange procedure, as described in Section 4.3.1. As in the case of the handshake, the value of *rxchang* represents the time spent on local computations at the replicas, rather than time on the wire. Tasks include role proof creation and verification, database operations to retrieve role keys, and message preparation costs. The cost of the extra RPCs is shown in *net*. *Bloom-Check* ("bloom") is the time needed to query the target's Bloom filter.

As Figure 4.5 indicates, the largest overheads incurred by either *T.Rex*-Sync or *trad-sync* are from disk accesses. Our current implementation saves all file data, including new updates, on disk. This performance could be improved by writing to the disk asynchronously, and possibly through use of a cache in DRAM. However, as much of the data is merely being relayed among multiple replicas, a cache might have little locality to exploit.

Handshake cost is insensitive to the number of roles, as the number of public-key cryptographic operations is constant. On the other hand, the cost of checking role proofs does increase with roles, as the number of available roles determines the number of HMAC operations needed by the source and target. Network costs (*net*) are higher for *T.Rex*, as *T.Rex*-sync requires more RPCs during an anti-entropy session. The cost of using Bloom filters (*bloom*) is comparable to that of using version vectors (*VVC*). However, *T.Rex*'s "freshness" constraint allows the costs associated with Bloom filters to be bounded (Section 4.4.2).

Figure 4.6 shows the data another way, breaking down costs of cryptographic operations with varying numbers of roles. Public-key operations are the most expensive, but as discussed above, do not vary with the number of roles. The numbers of HMAC and AES operations do vary. HMAC operations are needed for each role proof during the role exchange step, and all data is encrypted, but data for roles not known to either of the source and target is doubly encrypted. As the number of roles increases, the odds that a given role is not played by both replicas increases, implying that more data is doubly encrypted.

The new overheads are not negligible, but only become significant with many roles. However, recent user studies [65] have shown that even a static allocation of four roles serves many users well. While four roles might not suffice for the more expansive vision of sharing assumed by this work, the number of roles that can usefully be used in a group is likely to remain relatively low. Additionally, Figures Figures 4.5 and 4.6 does not consider the costs of sending data, which dwarfs the overhead we consider.

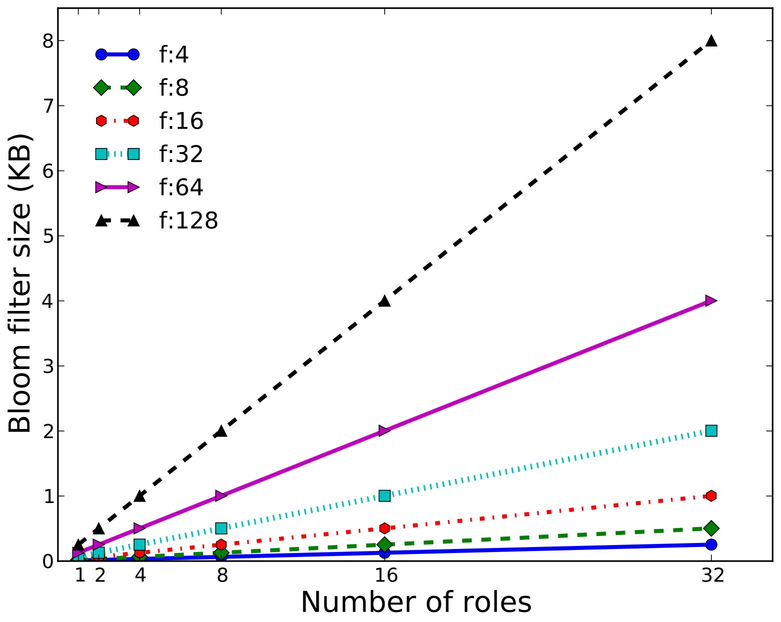

Figure 4.7: Bloom filter overhead with varying freshness.

4.4.2 Costs of update anonymity

T.Rex has update anonymity in that kernels for roles not played locally are opaque; the local replica knows literally nothing about them other than the randomly-created kernel GUID. While a conventional protocol can summarize known kernels through version vectors, *T.Rex* uses Bloom filters and local caches to store and forward kernels obliviously. This section investigates the overhead of this oblivious approach. We measure the direct overhead of using and sending Bloom filters instead of the more common version vectors.

Bloom filters *versus* version vectors

Figure 4.7 shows the size of the Bloom filters needed by *T.Rex* to support a false positive error rate of less than 0.5%. The number of elements in a Bloom filter is the number of

Number of Replicas	Version Vector (Bytes)
5	118
15	360
35	836
55	1295

Table 4.1: Representative version vector sizes (after serialization)

roles multiplied by the degree of freshness, e.g., the largest Bloom filter shown in Figure 4.7 contains $128x32 = 2^{12}$ elements. The figure demonstrates a worst-case cost, as kernels are only generated for those roles with activity during an update period. Hence, actual Bloom filter sizes are usually smaller than the numbers shown.

Version vectors grow linearly with the number of replicas that participate in the system. Our prototype implements vectors a set of tuples, each containing two integers. However, their final size is determined based on serialization algorithm of the protocol buffer implementation. Table 4.1 shows the size of the version vectors with different group sizes. These numbers were collected *after* the version vectors were serialized into protocol buffers. Protocol buffers shrink the space consumed by integers by representing them in as few bits as possible.

4.4.3 Storage costs

A final type of overhead is increased storage costs. Most of our protocol adds only small constant overheads, but role coherence duplicates objects across roles, and therefore potentially increases storage costs by a factor of R. However, this only occurs if objects are shared and modified in all roles, an unlikely worst-case scenario.

To summarize, updates to objects shared by multiple roles are forked (split into two distinct versions), rather than allowing updates from one role to be seen in another role. Assume $role_x$ and $role_y$ have intersection $I_{x,y}$. An update $U_i \in I_{x,y}$, created in role $role_x$, will not be seen in role $role_y$.

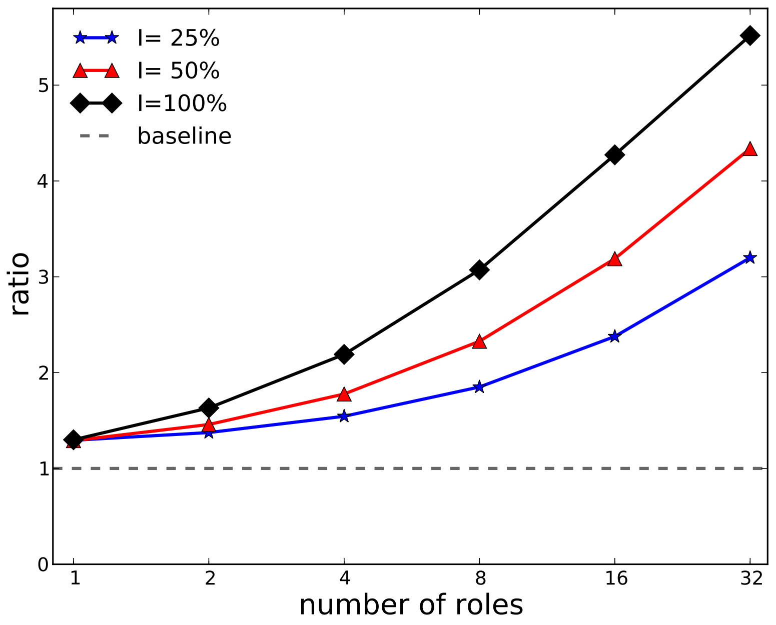

Figure 4.8: Data overheads after 10000 updates determined to 1000 files. The x axis is log-scale, while y is not. Meta-data overhead with varying number of roles.

We quantify the overhead of role coherence by directly measuring the increase in storage costs as the number of roles increases. Our experimental setup consists of 1,000 randomly selected objects from our dataset, with differing intersection sizes, I. We varied I from 25% of the original group, to 50%, and finally to 100%.

Files that fall into the intersection are assigned every available role r in the system with $r \in [1-32]$. The remaining files are each placed into a single, randomly selected role. Each run consists of 10,000 4KB file updates. Files to be updated are selected using a zipf distribution that biases towards smaller files. This is intended to model "hot" files, small files that are used and updated frequently. We also ran experiments selecting files uniformly at random, but the results were similar. We selected 4KB as the update size by observing block changes as image filters are applied to image files.

Table 4.2 shows the absolute overhead for storing the role-based versions when the

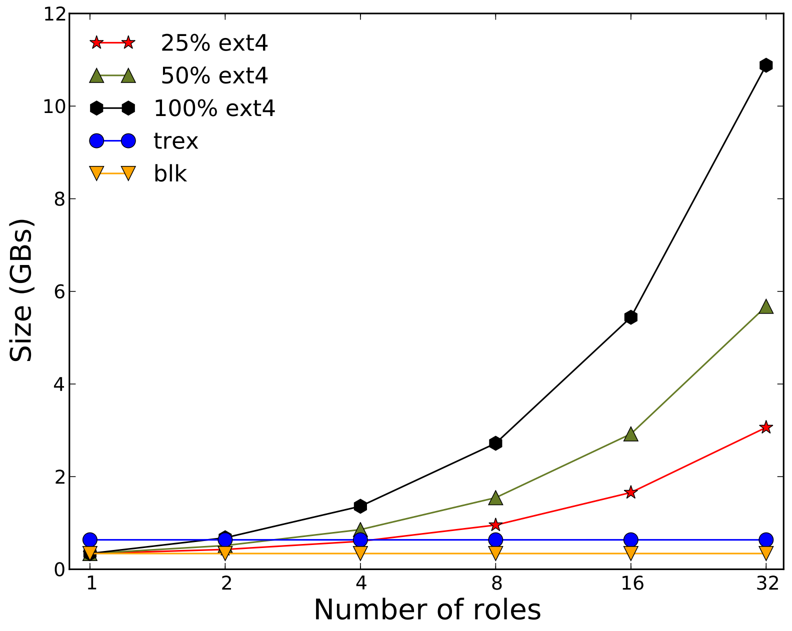

Figure 4.9: Data overheads after 10000 updates determined to 1000 files. The x axis is log-scale, while y is not. Total data consumption with varying number of roles. `blk` and `trex` are flat.

Roles	Baseline(MB)	Role Based (MB)	Cost (%)
1	30.08	38.93	29.4%
2	30.44	41.06	34.62%
4	30.96	45.88	45.54%
8	31.08	55.27	77.93%
16	34.26	73.52	114.59%
32	38.62	110.05	184.95%

Table 4.2: Absolute values of ***meta-data*** storage overhead between the baseline case and the storage mechanism supporting the role based consistency. The intersection level is at 50% over the total number of files. Role-based overhead scales linearly with the number of roles.

intersection $I = 50\%$. Figure 4.8 shows the relative overhead of the role-based approach over the baseline case, where objects are shared among all intersecting roles. Metadata costs for the role-based approach are roughly twice those of the baseline case for all I size and up to 4 roles, but rises to nearly six times as much for 32 roles with 100% intersection.

Again, however, these numbers represent worst-case overheads. For example, 32 roles with 100% intersection describes a system with 32 absolutely identical roles, neither useful nor likely in practice.

Figure 4.9 shows the computed overhead in file data storage for the experiment described in Figure 4.8, for three distinct types of systems. We use `ext4` as a straw-man to represent a generic whole-file forking approach. Assume a file *x* is present in both roles `friends` and `colleagues`, and then written in the context of `friends`. The whole-file approach of `ext4` would duplicate the entire file and then modify one version, resulting in an overhead of 100%. The more sophisticated `blk` represents files as ordered sets of fixed-size blocks [45, 93], and only duplicates blocks that differ. A 4KByte update might only modify a single file block. Finally, `trex` represents our prototype, which uses the `blk` approach, but also retains blocks from prior file versions. There are three different lines for `ext4`, as the overhead for the whole-file approach varies according to how much overlap there is among roles. Storage requirements of systems using a block-based approach, however, are unaffected by the number of roles. The storage costs of `trex` are slightly higher than those of `blk` because of the need to store old blocks.

4.5 *T.Rex* on the cloud

Thus far, I have presented *T.Rex* from the perspective of users who directly interact with one another. Recall from Figure 1.2, however, that many of the same issues encountered with local replicated services also present with cloud providers. Both data and meta-data can leak to replicated cloud servers. The *T.Rex* mechanisms deal with these issues effectively, so it is natural to explore whether *T.Rex* mechanisms can be used effectively in concert with cloud providers.

T.Rex potentially addresses another issue with cloud services: consistency. Cloud services vary greatly in their guarantees. Some guarantee response times, some make *session*

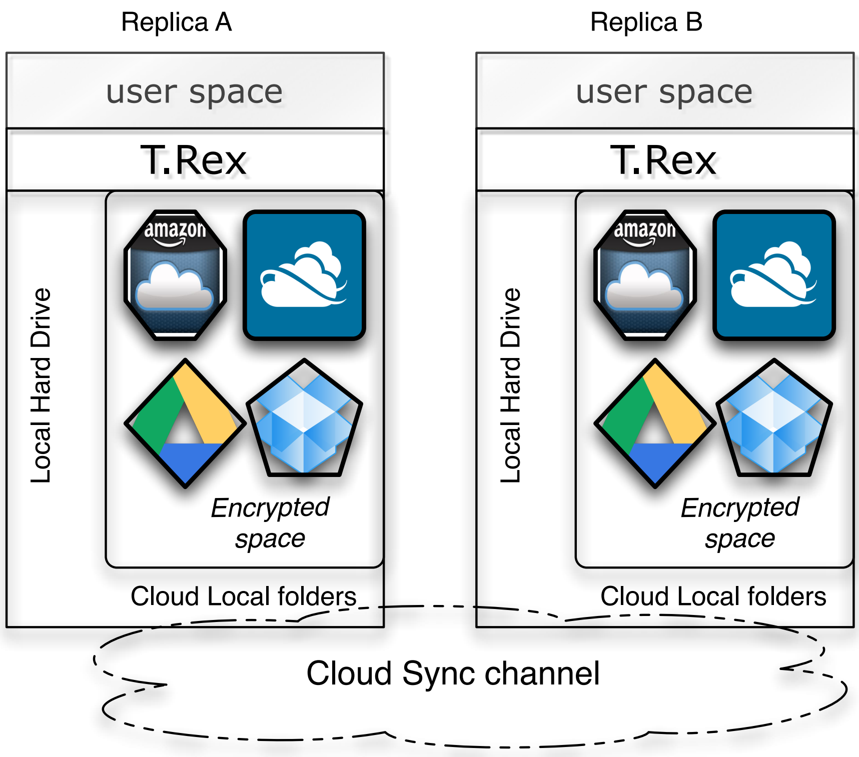

Figure 4.10: This figure illustrates how *T.Rex* can be used together with the existing auto-sync replication software. In this case *T.Rex* handles the security as well as the consistency and leaves the replication and the fault-tolerance to the cloud software agents that run locally in every user device. Every interaction of the users application is done through *T.Rex* which has hooks

guarantees [114], and at least one makes global guarantees of single-object coherence.

T.Rex makes hard consistency guarantees, and could potentially be used to regularize guarantees across multiple clouds similarly to bolt-on consistency [14]. Our approach would differ in that the consistency guarantees would be made along with security guarantees.

T.Rex-Cloud would differ from standard *T.Rex* mostly in that all communication would be through the cloud. If each *T.Rex* replica is connected to the same cloud services account, we can effectively use the cloud as a fast and wide communication channel.

The use of cloud services would allow us to dispense with the anti-entropy protocol. Kernels and shards would be created as in *T.Rex*, and then deposited into cloud storage where they would be retrieved by other *T.Rex* replicas.

Clouds and shards contain encrypted dependency information. A replica pulling a shard out of the cloud channel only *applies* if the consistency criteria in Section 4.2.2 are met. Security guarantees would also still be enforced, as kernels are created and encrypted as before.

Versus *T.Rex*, *T.Rex*-Cloud might have more durability; cloud updates would be present on many cloud servers, and might be backed up. *T.Rex*-Cloud also gets communication scheduling for free, and is simplified by the elimination of the anti-entropy protocol.

4.6 Summary

Traditional replicated data systems have assumed that all interacting users trust one another. In this paper, we have challenged this assumption by arguing that it can lead to sub-optimal performance and, particularly in the case of cloud-based systems, can lead to information leakage. Moreover, we have demonstrated that *various consistency schemes can be efficient and secure without requiring all interacting users to trust one another*. *T.Rex* makes this possible by combining several novel mechanisms to create a cryptographically secure, efficient anti-entropy protocol. Our implementation and test-bed evaluation of *T.Rex* demonstrate that it achieves security with modest computation and storage overheads.

We view *T.Rex* as the first step towards securing replicated storage systems; there remain many interesting open problems. Users worried about meta-data leakage might also worry about visible communication paths and other subtle issues; combining *T.Rex*'s techniques with anonymous communication systems like Onion Routing [47] is an interesting area of future work. We have sketched the design of a cloud-enabled *T.Rex*: one that would allow users to benefit from the reliability and availability of the cloud while maintaining control over the privacy and consistency of their data. The basic approach we take in doing so is to treat the cloud provider as a communication channel, one that potentially increases reliability. Another interesting area of future work is to understand how well consistency schemes "layer" upon one another. Moreover, with a system like *T.Rex* implementing end-to-end consistency, it is worth investigating whether cloud systems must invest in sophisticated consistency schemes of their own, or if the end-to-end argument [103] should be applied.

Chapter 5

Conclusions and Discussion

In this thesis I designed, built, and evaluated systems that allow cloud-based applications to fully use the cloud services without limiting the applications' ability to define their own consistency and information-sharing policies. My work allows applications to get the consistency they want (`Shell`) and retain control over the data that is visible during replications (*T.Rex*).

The end-to-end principle [103] states that application-specific functionality should be located at the edges of the system. In my work I separate the consistency and information-sharing polices from the mechanics of replication. Both of my designs are layers that can be placed between the application and the storage layer (Figures 3.1 and 4.10). Both of the designs are modular and can be used to support existing applications without causing major disruptions. Both `Shell` and *T.Rex* introduce new layers that are located between the applications and their storage layer. The introduction of these layers allows an application to regulate the flow of updates that it receives from the storage layer. By regulating the flow of updates, either by temporarily buffering inconsistent updates or by effectively encrypting them based on application-based sharing policies, an application can get consistency and privacy customized to its needs.

The `Shell` project (Chapter 3) focuses on consistency guarantees. In `Shell` I used a layered design to provide applications with higher consistency guarantees over eventually consistent data stores. The `Shell` layer is located between the application and the cloud services, and allows applications to achieve causal consistency guarantees without explicitly relying on dependency tracking. This design also allows applications to register customized conflict resolution handlers. Most importantly, my design illustrates a general approach of isolating application code from the cloud provider APIs. This isolation allows the application to evolve independently from any provider's API calls. Furthermore, the design isolates the consistency implementation to a unit level, allowing for correct consistency implementations that remain unchanged despite frequent changes to the application code.

The *T.Rex* project (Chapter 4) focuses on information leakage during replication. In *T.Rex* I used the idea of layering to assign the replicas of a distributed system to different levels of trust. I achieved this by designing a cryptographically-secure replica synchronization protocol that supports Topology Independence, but does not assume uniform trust among participating replicas. *T.Rex* allows applications to use cloud synchronization services such as Dropbox without having to reveal any of their data to a third-party provider.

I defended the following thesis statement: *By decoupling consistency determination and trust from the underlying distributed data store, it is possible to (1) support application-specific consistency guarantees; (2) allow for topology-independent replication protocols that do not compromise application privacy.*

The way we build cloud applications inherently shifts some of the control from the application developers to providers. This means that cloud service implementations exist between applications and their data. Moreover, provider implementations might have other dependencies, internally relying on other cloud services. Unfortunately, abstraction leakage–faulty assumptions made on the application layer (for example assuming ACID behavior over a BASE data store)–and tight coupling between provider APIs and application code leads to high complexity, which makes application code hard to understand and even harder to maintain.

In this work I focused on consistency and replication. My primary finding is that modular designs resulting from a layered approach can effectively insulate applications from the vagaries of specific cloud service implementations. This, in turn, allows us to fully exploit the benefits of cloud services, without sacrificing application flexibility.

5.1 Future Work & Open Problems

In this section, I present open problems that arose during my work in `Shell` (Chapter 3) and *T.Rex* (Chapter 4).

Consistency Testbed: The `Shell` architecture provides an easy way to plug a consistency protocol into an application stack. In this work I implemented two consistency protocols, monotonic reads and causal consistency, and tested their behavior over two different data stores. The easy introduction of a new consistency protocol, coupled with the fact that there is no need to change either the application code or the storage layer, makes the `Shell` paradigm a good candidate for the creation of a consistency testbed. In this scenario, it is possible to use the `Shell` architecture to evaluate the effects of different consistency protocols.

Probabilistic Update-Visibility: I would like to explore how a probabilistic model can be used to characterize the `update-visibility`. This would help applications to develop more effective caching schemes that will minimize any unnecessary communication with the cloud provider.

Support of Transactions: In `Shell` I support only weak consistency protocols. I would like to explore how someone can support transactions while using the layer paradigm of `Shell`.

Garbage collection: In `Shell` I used two types of storage. In the first type I store data that is visible to the application. In the second type I store the incoming updates that `Shell` cannot apply yet because they do not comply with the consistency protocol (*defer*-buffer). The *defer*-buffer is only visible to the `Shell` layer. Updates in the *defer*-buffer are immutable and have a global unique identifier. After an update is consumed from all the participating replicas, it loses its relevance. The current `Shell` implementation does not garbage collect these updates.

One solution could be for `Shell` or the source replica to assign an expiration timer (TTL) to each update. Then the `Shell` layer of the replica responsible for each update could garbage collect the update based on this TTL value. A good TTL value should be above the `update-visibility` metric. Although this approach would work most of the time, care should be taken when choosing this value because the `update-visibility` for some consistency protocols has a long tail. More specifically as I showed in Figures 3.12 and 3.13 the value of the `update-visibility` metric depends on how fast the underlying storage layer can do replication. For example, in the case of Dropbox and causal consistency an update might take up to 30 seconds to reach every other replica in the system.

Another solution to the garbage collection problem could be a quorum protocol like Paxos [67]. However, partial majority will not be enough in this case for every consistency protocol. Without seeing all the updates, `Shell` could not enforce correctness in the case of causal consistency, for example.

Division of Storage layers: Another open question is related to the existence of two storage locations. `Shell` uses the application-store for two purposes: storing consistent application data and as a *defer* buffer. The reason for this separation comes from the need to have the replicas pushing immutable updates. I did this because some NoSQL data stores might overwrite consecutive updates on the same object [14].

The overwritten histories of an object might be a problem for some consistency protocols like causal consistency but not for others for example monotonic reads. Existing research [14, 74] has shown that dependency tracking can help solve this issue in the case of causal consistency. Unfortunately, the application needs to maintain these dependencies, and of course the application developers to have maintain the development tracking software. In `Shell` I was aware of this trade-off and made the conscious decision to use two logical partitions of the storage.

Version Clocks at Scale: Unfortunately, version clocks do not scale since they require replicas to store information that grows linearly with the number of clients or updates.

DIFF_{ji} metric and Partial Ordering: In `Shell` I used traces from NBA games. Every trace contains a totally ordered set of events. During the experiment I replayed every game sequentially. In other words, replicas first completed the game_X and then moved on to the game_{X+1}. Because of this, the DIFF_{ji} metric was able to quantify inconsistencies without having the application domain field in the Shell SO populated.

In a parallel execution of NBA games, the set of their events would be partially ordered. In this case, the application programmer must populate the application domain field, for example by using the game identifier, in order to maintain correctness at per NBA-game basis. The `consistency-enforcer` will not unnecessarily delay events that belong to different games under causal consistency. But within a single application domain, nothing changes, and every update will be applied only when it meets the consistency protocol criteria.

Shell implements causal consistency by only looking at the version vectors. For a single application domain, this is conservative because Shell might treat two events as causally related when they are not. To avoid this limitation I could potentially ask for application-level dependencies. However, this defeats the purpose of Shell because one of my major goals was to isolate consistency from both application and the storage layer.

IoT Replication IoT devices would certainly benefit from the secure replication protocol of *T.Rex*. *T.Rex*-Sync is topology independent replication protocol and will allow for secure replication of configuration changes and caching even between different IoT devices of different owners. However, *T.Rex* requires some heavy cryptographic operations that might not be suitable for an environment of low-power devices.

Bibliography

[1] Strong consistency. `https://en.wikipedia.org/wiki/Strong_consistency`.

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

[3] Divyakant Agrawal, Amr El Abbadi, and Robert C Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–172. ACM, 1997.

[4] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 1995.

[5] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[6] Amazon. Aurora: Relational database build for the cloud. `https://aws.amazon.com/rds/aurora/`.

[7] Amazon. Glacier: Long-term, secure, durable object storage. `https://aws.amazon.com/glacier/`.

[8] Amazon. Kinesis: Real time data streams. `https://aws.amazon.com/kinesis/`.

[9] Amazon. Redshift: Fast simple data warehousing. `https://aws.amazon.com/redshift/`.

[10] Amazon. Regions and availability zones. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`.

[11] Amazon. S3 product details. `https://aws.amazon.com/s3/details/`.

[12] Amazon. S3: Simple storage service. `https://aws.amazon.com/documentation/s3/`.

[13] National Basketball Association. Nba stats. `http://stats.nba.com/`.

[14] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.

[15] Suman Banerjee, Seungjoon Lee, Bobby Bhattacharjee, and Aravind Srinivasan. Resilient multicast using overlays. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 102–113, New York, NY, USA, 2003. ACM.

[16] Yunus Basagalar, Vassilios Lekakis, and Pete Keleher. Growing secure distributed systems from a spore. In *ICDCS*. IEEE Computer Society, 2012.

[17] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4):619–665, December 2010.

[18] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Pravee Yalagandula, and Jiandan Zheng. Practi replication. In *NSDI*. USENIX, 2006.

[19] Nalini Moti Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Michael Dahlin, and Robert Grimm. PADS: A policy architecture for distributed storage systems. In *NSDI*. USENIX Association, 2009.

[20] David Bermbach, Jorn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A middleware guaranteeing client-centric consistency on top of eventually consistent datastores. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13. IEEE Computer Society, 2013.

[21] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, New York, NY, USA, 2011. ACM.

[22] Matthew Addison Blaze. *Caching in Large-scale Distributed File Systems*. PhD thesis, Princeton University, Princeton, NJ, USA, 1993. UMI Order No. GAX93-11182.

[23] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, New York, NY, USA, 2000. ACM.

[24] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 1987.

[25] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, 1999.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*. USENIX Association, 2006.

[27] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. Cyrus: Towards client-defined cloud storage. In *EuroSys '15*, New York, NY, USA, 2015. ACM.

[28] D. Clark. The design philosophy of the darpa internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. ACM.

[29] 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 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[30] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 2004.

[31] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 202–215, New York, NY, USA, 2001. ACM.

[32] 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 *SOSP*. ACM, 2007.

[33] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[34] Tom St Denis. Libtomcrypt. `https://github.com/libtom/libtomcrypt`.

[35] Przemysaw Wegrzyn Dhiru Kholia. Looking inside the (drop) box. In *7th Usenix Wokshop on offensive Technologies*, 2013.

[36] DropBox. File synchronization service. `https://www.dropbox.com/`.

[37] Dropbox. What is lansync. `https://www.dropbox.com/help/syncing-uploads/lan-sync-overview`.

[38] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013.

[39] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, 2014.

[40] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[41] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. Sporc: group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[42] David Ferraiolo and Richard Kuhn. Role-based access control. In *National Computer Security Conference*, 1992.

[43] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *OSDI*, Seattle, Washington, November 2006.

[44] Apache Software Foundation. Cassandra: Manage massive amounts of data, fast, without losing sleep. `http://cassandra.apache.org/`.

[45] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the nineteenth Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, Bolton Landing, NY, USA, October 2003. ACM, ACM Press.

[46] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[47] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[48] Google. Cloud dataflow. https://cloud.google.com/dataflow/.

[49] Google. Write, edit, collaborate wherever you are. https://www.google.com/docs/about/.

[50] Network Working Group. Pgm reliable transport protocol specification. https://tools.ietf.org/html/rfc3208.

[51] Oleg Gutsol. 500px. http://500px.com.

[52] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, Anaheim, CA, 1990.

[53] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 1983.

[54] Jonathan Halterman. Failsafe: Simple sophisticated failure handling. https://github.com/jhalterman/failsafe.

[55] James Hamilton. At scale, rare events aren't rare. http://perspectives.mvdirona.com/2017/04/at-scale-rare-events-arent-rare/.

[56] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 2001.

[57] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988.

[58] IEEE. Standard for software unit testing, 1986.

[59] iMatrix. Advanced pub-sub patterns. http://zguide.zeromq.org/php:chapter5.

[60] iMatrix. Zeromq: Distributed messaging. http://zeromq.org/.

[61] Mohamed Ali Kaafar, Laurent Mathy, Thierry Turletti, and Walid Dabbous. Virtual networks under attack: Disrupting internet coordinate systems. In *Proceedings of the 2006 ACM CoNEXT Conference*, CoNEXT '06. ACM, 2006.

[62] Michael Kassner. Researchers reverse-engineer the dropbox client: What it means. http://goo.gl/YVdguD.

[63] Dhiru Kholia. Long promised post module for hijacking dropbox accounts. https://github.com/rapid7/metasploit-framework/pull/1497, 2013.

[64] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on*, pages 127–128. IEEE, 1991.

[65] Tiffany Hyun-Jin Kim, Lujo Bauer, James Newsome, Adrian Perrig, and Jesse Walker. Challenges in access right assignment for secure home networks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, HotSec'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[66] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[67] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[68] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[69] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07. USENIX Association, 2007.

[70] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. *SIGOPS Oper. Syst. Rev.*, 1975.

[71] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[72] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004.

[73] R.J. Lipton and J.S. Sandberg. *PRAM: a scalable shared memory*. Number no. 180 in Research report // Princeton University, Department of Computer Science. Princeton University, Department of Computer Science, 1988.

[74] 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 Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11. ACM, 2011.

[75] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[76] Dahlia Malkhi and Doug Terry. Concise version vectors in winfs. *Distributed Computing*, 3:209–219, 2007.

[77] Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[78] Michelle L. Mazurek, Eno Thereska, Dinan Gunawardena, R.Harper, and James Scott. Zzfs: A hybrid device and cloud file system for spontaneus users. In *FAST*, 2012.

[79] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security Symposium*, 2011.

[80] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. Provenance-aware storage systems. In *USENIX*, pages 43–56. USENIX, 2006.

[81] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[82] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, February 1988.

[83] Sharman Networks. Kazaa: P2p file sharing. `https://en.wikipedia.org/wiki/Kazaa`.

[84] Derek Newton. Dropbox authentication: insecure by design. `http://dereknewton.com/2011/04/dropbox-authentication-static-host-ids/`, 2011.

[85] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In *OSDI*, 2004.

[86] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[87] Oracle. Core j2ee patterns - data access objects. `http://www.oracle.com/technetwork/java/dataaccessobject-138824.html`.

[88] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *Software Engineering, IEEE Transactions on*, 1983.

[89] Daniel Peek and Jason Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *OSDI*, pages 219–232, 2006.

[90] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301. ACM, 1997.

[91] Ansley Post, Juan Navarro, Petr Kuznetsov, and Peter Druschel. Autonomous storage management for personal devices with podbase. In *USENIX Annual Technical Conference*. USENIX, 2011.

[92] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[93] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the First USENIX conference on File and Storage Technologies*, pages 89–101, Monterey,CA, January 2002.

[94] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09. USENIX Association, 2009.

[95] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[96] RedisLab. How fast is redis? `https://redis.io/topics/benchmarks`.

[97] Oriana Riva, Qin Yin, Dejan Juric, Ercan Ucan, and Timothy Roscoe. Policy expressivity in the anzere personal cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 14:1–14:14. ACM, 2011.

[98] Martin Robert C. *Agile Software Development, Principles, Patterns, and Practices*. Pearson Higher Education; International ed edition, 2013.

[99] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[100] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 188–201, New York, NY, USA, 2001. ACM.

[101] Nicolas Ruff and Florian Ledoux. A critical analysis of dropbox software security. *ASFWS 2012, Application Security Forum*, 2012.

[102] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: semantic data management for the home. In *Proccedings of the 7th conference on File and storage technologies*, pages 167–182. USENIX Association, 2009.

[103] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 1984.

[104] Salvatore Sanfilippo. Redis In Memory Data Structure . `https://redis.io/`.

[105] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[106] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 119–132, New York, NY, USA, 2005. ACM.

[107] Joel Spolsky. The law of leaky abstractions. `https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/`.

[108] 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 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[109] Jacob Strauss, Justin Mazzola Paluska, Chris Lesniewski-Laas, Bryan Ford, Robert Morris, and Frans Kaashoek. Eyo: device-transparent personal storage. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[110] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC*. ACM, 2006.

[111] Graham Sutherland. Installing dropbox? prepare to lose aslr. `http://codeinsecurity.wordpress.com/2013/09/09/installing-dropbox-prepare-to-lose-aslr/`, 2013.

[112] TechoPedia. Minimum viable product. `https://www.techopedia.com/definition/27809/minimum-viable-product-mvp`.

[113] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, New York, NY, USA, 1995. ACM.

[114] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, Washington, DC, USA, 1994. IEEE Computer Society.

[115] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Adrian Perrig, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[116] Arthur O'Dwyer Troy D. Hanson. Uthash. http://uthash.sourceforge.net.

[117] Kenton Varda. Protocol buffers: Google's data interchange format. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googlesdata.html, 2008.

[118] Kaushik Veeraraghavan, Jason Flinn, Edmund B. Nightingale, and Brian Noble. qufiles: the right file at the right time. In *FAST*, 2010.

[119] Werner Vogels. All things distributed - eventually consistent revisited. `http://www.allthingsdistributed.com/2008/12/eventually_consistent.html`.

[120] Wikipedia. Dyn 2016 cyberattack. `https://en.wikipedia.org/wiki/2016_Dyn_cyberattack`.

[121] Ted Wobber, Thomas L. Rodeheffer, and Douglas B. Terry. Policy-based access control for weakly consistent replication. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 293–306, New York, NY, USA, 2010. ACM.

[122] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, 2013. ACM.

[123] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM*, 1974.

[124] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15. ACM, 2015.

[125] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J.Sel. A. Commun.*, 22(1):41–53, September 2006.