Diamond: Automating Data Management and Storage for Wide-area, Reactive Applications

Irene Zhang Niel Lebeck Pedro Fonseca Brandon Holt Raymond Cheng
Ariadna Norberg Arvind Krishnamurthy Henry M. Levy

University of Washington

Abstract

Users of today's popular wide-area apps (e.g., Twitter, Google Docs, and Words with Friends) must no longer save and reload when updating shared data; instead, these applications are *reactive*, providing the illusion of continuous synchronization across mobile devices and the cloud. Achieving this illusion poses a complex *distributed data management* problem for programmers. This paper presents the first *reactive data management service*, called Diamond, which provides persistent cloud storage, reliable synchronization between storage and mobile devices, and automated execution of application code in response to shared data updates. We demonstrate that Diamond greatly simplifies the design of reactive applications, strengthens distributed data sharing guarantees, and supports automated reactivity with low performance overhead.

1 Introduction

The modern world's ubiquitous mobile devices, infinite cloud storage, and nearly constant network connectivity are changing applications. Led by social networks (e.g., Twitter), social games (e.g., Words with Friends) and collaboration tools (e.g., Google Docs), today's popular applications are *reactive* [41]: they provide users with the illusion of *continuous synchronization* across their devices without requiring them to explicitly save, reload, and exchange shared data. This trend, not limited merely to mobile apps, includes the latest distributed versions of traditional desktop apps on both Windows [13] and OSX [4].

Maintaining this illusion presents a challenging *distributed data management* problem for application programmers. Modern reactive applications consist of *widely distributed* processes sharing data across mobile devices, desktops, and cloud servers. These processes make concurrent data updates, can stop or fail at any time, and may be connected by slow or unreliable links. While distributed storage systems [17, 77, 15, 23, 20] provide persistence and availability, programmers still face the formidable challenge of synchronizing updates between application processes and distributed storage in a *fault-tolerant*, *consistent* manner.

This paper presents *Diamond*, the first *reactive data management service* (RDS) for wide-area applications that continuously synchronizes shared application data across distributed processes. Specifically, Diamond performs the following functions on behalf of an application: (1) it ensures that updates to shared data are consistent and durable, (2) it reliably coordinates and synchronizes shared data updates across processes, and (3) it automatically triggers *reactive code* when shared data changes so that processes can perform appropriate tasks. For example, when a user updates data on one device (e.g., a move in a multi-player game), Diamond persists the update, reliably propagates it to other users' devices, and transparently triggers application code on those devices to react to the changes.

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

1. **Reactive Data Map** (`rmap`), a primitive that lets applications create *reactive data types* – shared, persistent data structures – and map them into the Diamond data management service so it can automatically synchronize them across distributed processes and persistent storage.
2. **Reactive Transactions**, an interactive transaction type that automatically *re-executes* in response to shared data updates. These "live" transactions run *application code* to make local, application-specific updates (e.g., UI changes).
3. **Data-type Optimistic Concurrency Control** (DOCC), a mechanism that leverages data-type semantics to concurrently commit transactions executing commutative operations (e.g., writes to different list elements, increments to a counter). Our experiments show that DOCC copes with wide-area

latencies very effectively, reducing abort rates by up to 5x.

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

2 Traditional Data Management Techniques for Reactive Apps

Reactive applications require synchronized access to distributed shared data, similar to shared virtual memory systems [46, 10]. For practical performance in the wide-area environment, apps must be able to control: (1) *what* data in each process is shared, (2) *how* often it is synchronized, and (3) *when* concurrency control is needed. Existing applications use one of several approaches to achieve synchronization with control. This section demonstrates that these approaches are all complex, error-prone, and make it difficult to reason about application data consistency.

As an example, we analyze a simple social game based on the 100 game [1]. Such games are played by millions [78], and their popularity changes constantly; therefore, game developers want to build them quickly and focus on game logic rather than data management. Because game play increasingly uses real money (almost $2 billion last year [24]), their design parallels other reactive applications where correctness is crucial (e.g., apps for first responders [52] and payment apps [81, 72]).

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

2.1 Roll-your-own Data Management

Many current reactive apps "roll-their-own" *application-specific* synchronization across distributed processes *on top of* general-purpose distributed storage (e.g., Spanner [17], Dropbox [23]). Figure 1

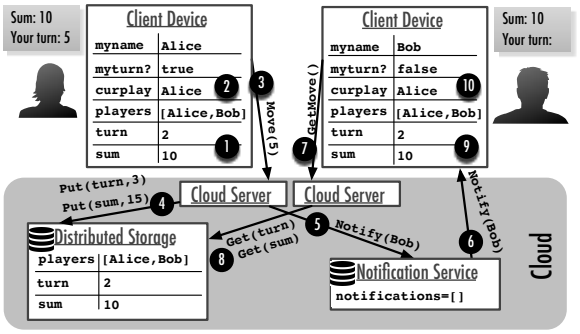

Figure 1: **The 100 game.** Each box is a separate address space. `players`, `turn` and `sum` are shared across address spaces and the storage system; `myturn?` and `curplay` are derived from shared data. When shared values change, the app manually updates distributed storage, other processes with the shared data, and any data in those processes derived from shared data, as shown by the numbered steps needed to propagate Alice's move to Bob.

shows a typical three-tiered architecture used by these apps (e.g., PlayFish uses it to serve over 50 million users/month [34]). Processes on *client devices* access stateless *cloud servers*, which store persistent game state in a *distributed storage* system and use a reliable *notification service* (e.g., Thialfi [3]) to trigger changes in other processes for reactivity. While all application processes can fail, we assume strong guarantees – such as durability and linearizability – for the storage system and notification service. Although such apps could rely on a single server to run the game, this would create a centralized failure point. Clients cache game data to give users a responsive experience and to reduce load on the cloud servers [34].

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

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

Correctly managing this application data demands that the programmer reason about failures and data races at every step. For example, the cloud server could fail in the

middle of step 4, violating atomicity. It could also fail between steps 4 and 5, making the application appear as if it is no longer reactive.

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

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

2.2 Wide-area Storage Systems

A simple, alternative way to manage data manually is to store shared application data in a wide-area storage system (e.g., Dropbox [23]). That is, rather than calling `move` in step 3, the application stores and updates `turn` and `sum` in a wide-area storage system. Though simple, this design can be very expensive. Distributed file systems are not designed to frequently synchronize small pieces of data, so their coarse granularity can lead to moving more data than necessary and false sharing.

Further, while this solution synchronizes Alice's updates with the cloud, it does not ensure that Bob receives Alice's updates. To simulate reactive behavior and ensure that Bob sees Alice's updates, Alice must still use a wide-area notification system (e.g., Apple Push Notifications [6]) to notify Bob's client after her update. Unfortunately, this introduces a race condition: if Bob's client receives the notification before the wide-area storage system synchronizes Alice's update, then Bob will not see Alice's changes. Worse, Bob will never check the storage system again, so he will never see Alice's update, leaving him unable to make progress. Thus, this solution retains all of the race conditions described in Section 2.1 *and* introduces some new ones.

2.3 Reactive Programming Frameworks

Several programming frameworks (e.g., Firebase [26], Parse [60] with React [64], Meteor [51]) have recently been commercially developed for reactive applications. These frameworks combine storage and notification systems and automate data management and synchronization across systems. However, they do not provide a clear consistency model, making it difficult for programmers to reason about the guarantees provided by their synchronization mechanisms. Further, they offer no distributed concurrency control, leaving application programmers to contend with race conditions; for example, they can lead to the race condition described in Section 2.1.

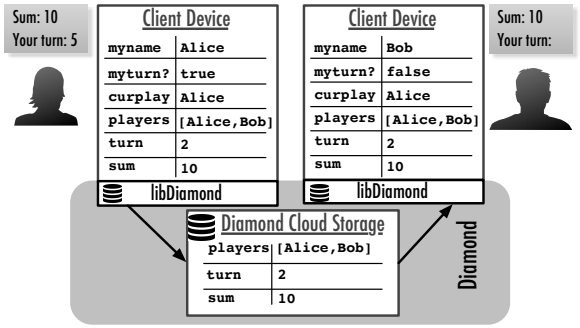

Figure 2: **Diamond 100 game data model.** The app `rmaps` `players`, `turn` and `sum`, updates them in read-write transactions and computes `myturn?` and `curplay` in a reactive transaction.

3 Diamond's System and Programming Model

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

3.1 System Model

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

Figure 2 shows the 100 game data model using Diamond. Compared to Figure 1, the application can directly read and write to shared data in memory, and Diamond ensures updates are propagated to cloud storage and other processes. Further, Diamond's strong transactional guarantees eliminate the need for programmers to reason about failures and concurrency.

3.2 Data Model

Diamond supports *reactive data types* for fine-grained synchronization, efficient concurrency control, and persistence. As with popular data structure stores [19], such as Redis [67] and Riak [68], we found that simple data types are general enough to support a wide range of applications

Table 1: **Reactive data types.**

Type	Operations	Description
`Boolean`	`Get()`, `Put(bool)`	Primitive boolean
`Long`	`Get()`, `Put(long)`	Primitive number
`String`	`Get()`, `Put(str)`	Primitive string

`Counter`	`Get()`, `Put(long)`
	`Increment(long)`
	`Decrement(long)`
		Long Counter

`IDGen`	`GetUID()`	Unique ID generator
`LongSet`	`Get(idx)`, `Contains(long)` `Insert(long)`	Ordered number set
`LongList`	`Get(idx)`, `Set(idx, long)` `Append(long)`	Number list
`StringSet`	`Get(idx)`, `Contains(str)` `Insert(str)`	Ordered string set
`StringList`	`Get(idx)`, `Set(idx, str)` `Append(str)`	String list
`HashTable`	`Get(key)`, `Set(key, val)`	Unordered map

and provide the necessary semantics to enable commutativity and avoid false sharing. Table 1 lists the supported persistent data types and their operations. In addition to primitive data types, like `String`, Diamond supports simple Conflict-free Replicated Data-types (CRDTs) [69] (e.g., `Counter`) and collection types (e.g., `LongSet`) with efficient type-specific interfaces. Using the most specific type possible provides the best performance (e.g., using a `Counter` for records that are frequently incremented).

A single Diamond *instance* provides a set of *tables*; each table is a key-to-data-type map, where each entry, or *record*, has a single persistent data type. Applications access Diamond through language bindings; however, applications need not be written in a single language. We currently support C++, Python and Java on both x86 and Android but could easily add support for other languages (e.g., Swift [5]).

3.3 System Calls

While apps interact with Diamond largely through reactive data types, we provide a minimal system call interface, shown in Table 2, to support transactions and `rmap`.

3.3.1 The `rmap` Primitive

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

Table 2: **Diamond system calls.**

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

3.3.2 Transaction model

Application processes use Diamond transactions to read and write `rmap`ped variables. Diamond transactions are *interactive* [73], i.e., they let applications interleave application code with accesses to reactive data types. We support both standard *read-write transactions* and new *reactive transactions*. Applications cannot execute transactions across `rmap`ped variables from different tables, while operations executed outside transactions are treated as single-op transactions.

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

Before `execute_txn` returns, Diamond logs the transaction, with its read and write sets, to persistent storage. This step guarantees that the transaction will eventually execute and that the completion callback will eventually execute even if the client crashes and restarts. This guarantee lets applications buffer transactions if the network is unavailable and easily implement custom retry functionality in the completion callback. If the callback reports that the transaction successfully committed, then Diamond guarantees ACID+R semantics for all accesses to `rmap`ped records; we discuss these in more detail in Section 3.4. On abort, Diamond rolls back all local modifications to `rmap`ped variables.

Reactive transactions. Reactive transactions help application processes automatically propagate changes made to reactive data types. Each time a read-write transaction modifies an `rmap`ped variable in a reactive transaction's read set, the reactive transaction re-executes, prop-

agating changes to derived local variables. As a result, reactive transactions provide a "live" view that gives the illusion of reactivity while maintaining an imperative programming style comfortable to application programmers. Further, because they read a consistent snapshot of `rmap`ped data, reactive transactions avoid application-level bugs common to reactive programming models [48].

Applications do not explicitly invoke reactive transactions; instead, they register them by passing a closure to `register_reactxn`, which returns a `txn_id` that can be used to unregister the transaction with `reactxn_stop`. Within the reactive transaction closure, the application can read but not write `rmap`ped records, preventing potential data flow cycles. Since reactive transactions are designed to propagate changes to local variables, the application can read and write to local variables at any time and trigger side-effects (i.e., print-outs, updating the UI). Diamond guarantees that reactive transactions never abort because it commits read-only transactions locally at the client. Section 4 details the protocol for reactive transactions.

Reactive transactions run in a background thread, concurrently with application threads. Diamond transactions do not protect accesses to local variables, so the programmer must synchronize with locks or other mechanisms. The read set of a reactive transaction can change on every execution; Diamond tracks the read set from the latest execution. Section 6.2 explains how to use reactive transactions to build general-purpose, reactive UI elements.

3.4 Reactive Data Management Guarantees

Diamond's guarantees were designed to meet the requirements of reactive applications specified in Section 2, eliminating the need for each application to implement its own complex data management. To do so, Diamond enforces *ACID+R* guarantees for reactive data types:

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

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

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

Table 3: **Diamond's isolation levels.** Isolation levels for read-write transactions and associated ones for reactive transactions.

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

Table 3 lists Diamond's isolation levels, which can be set per table. Diamond's default is strict serializability because it eliminates the need for application programmers to deal with inconsistencies caused by data races and failures. Lowering the isolation level leads to fewer aborts and more concurrency; however, more anomalies arise, so applications should either expect few conflicts, require offline access, or tolerate inaccuracy (e.g., Twitter's most popular hash tag statistics). Section 5.1 describes how DOCC increases concurrency and reduces aborts for transactions even at the highest isolation levels.

3.5 A Simple Code Example

To demonstrate the power of Diamond to simplify reactive applications, Figure 3 shows code to implement the 100 game from Section 2 in Diamond. This implementation provides persistence, atomicity, isolation and reactivity for every join and move operation in only 34 lines of code. We use three reactive data types for shared game data, declared on line 2 and `rmap`ped in lines 7-9. It is important to ensure a strict ordering of updates, so we create a table in strict serializable mode on line 6. On line 12, we define a general-purpose transaction callback for potential transaction failures. On line 16, we execute a read-write transaction to add the player to the game, passing `myname` by value into the transaction closure. Using DOCC allows Diamond to commit two concurrent executions of this transaction while guaranteeing strict serializability.

Line 20 registers a reactive transaction to print out the score and current turn. Diamond's ACID+R guarantees ensure that the transaction re-executes if `players`, `turn` or `sum` change, so the user always has a consistent, up-to-date view. Note that we can print to stdout because the reactive transaction will not abort, and the printouts reflect a serializable snapshot, avoiding reactive glitches [48]. On line 32, we wait for user input in the `while` loop and use a read-write transaction to commit the entered move.

Diamond's strong guarantees eliminate the need for programmers to reason about data races or failures. Taking our examples from Section 2, Diamond ensures that when the game commits Alice's move, the move is never

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

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

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

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

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

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

Figure 3: **Diamond code example.** Implementation of the 100 game using Diamond. Omitting includes, set up, and error handling, this code implements a working, C++ version of the 100 game [1]. `DStringSet`, `DLong` and `DCounter` are reactive data types provided by the Diamond C++ library.

lost and Bob eventually sees it. Diamond also ensures that, if Charlie joins before Bob makes his move, Alice either sees Charlie join without Bob's move, or both, but never sees Bob's move without seeing Charlie join. As a result, programmers no longer need to reason about race conditions, greatly simplifying the game's design. To our knowledge, no other system provides all of Diamond's ACID+R properties.

3.6 Offline Support

Wi-Fi and cellular data networks have become widely available, and reactive applications typically have limited offline functionality; thus, Diamond focuses on providing *online reactivity*, unlike storage systems (e.g., Bayou [77] and Simba [61]). However, Diamond still provides limited offline support. If the network is unavailable, `execute_txn` logs and transparently retries, while Diamond's CRDTs make it more likely that transactions commit after being retried. For applications with higher contention, Diamond's read committed mode enables commits locally at the client while offline, and any modifications eventually converge to a consistent state for Diamond's CRDTs.

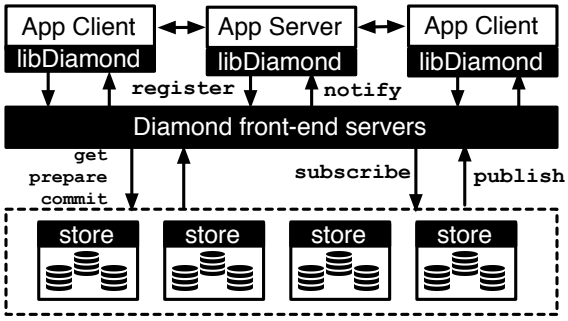

Figure 4: **Diamond architecture.** Distributed processes share a single instance of the Diamond storage system.

3.7 Security

Similar to existing client-focused services, like Firebase [26] and Dropbox [23], Diamond trusts application clients not to be malicious. Application clients authenticate with the Diamond cloud through their LIBDIAMOND client before they can `rmap` or access reactive data types. Diamond supports isolation between users through *access control lists* (ACLs); applications can set `rmap`, read, and write permissions per table. Within tables, keys function as capabilities; a client with a key to a record has permission to access it. Applications can defend against potentially malicious clients by implementing server-side security checks using reactive transactions on a secure cloud server.

4 Diamond's System Design

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

4.1 Data Management Architecture

Figure 4 presents an overview of Diamond's key components. Each LIBDIAMOND client provides client-side caching and access to cloud storage for the application process. It also registers, tracks and re-executes reactive transactions and keeps a persistent transaction log to handle device and network failures.

The Diamond cloud consists of *front-end servers* and *back-end storage* servers, which together provide durable storage and reliable notifications for reactive transactions. Front-end servers are scalable, stateless nodes that provide LIBDIAMOND clients access to Diamond's back-end storage, which is partitioned for scalability and replicated (using Viewstamped Replication (VR) [58]) for fault tolerance. LIBDIAMOND clients could directly access back-end storage, but front-end servers give clients a single connection point to the Diamond cloud, avoiding the need for

them to authenticate with many back-end servers or track the partitioning scheme.

4.2 `rmap` and Language Bindings

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

4.3 Transaction Coordination Overview

Figure 5 shows the coordination needed across LIBDIAMOND clients, front-end servers and back-end storage for both read-write and reactive transactions. This section briefly describes the transaction protocols.

Diamond uses *timestamp ordering* to enforce isolation across LIBDIAMOND clients and back-end storage; it assigns every read-write transaction a unique *commit timestamp* that is provided by a replicated *timestamp service* (tss) (not shown in Figure 4). Commit timestamps reflect the transaction commit order, e.g., in strict serializability mode, they reflect a single linearizable ordering of committed, read-write transactions. Both Diamond's client-side cache and back-end storage are *multi-versioned* using these commit timestamps.

4.3.1 Running Distributed Transactions

Read-write and reactive transactions execute similarly; however, as Section 5 relates, reactive transactions can commit locally and often avoid wide-area accesses altogether. We lack the space to cover Diamond's transaction protocol in depth; however, it is similar to Spanner's [17] with two key differences: (1) Diamond uses DOCC for concurrency control rather than a locking mechanism, and (2) Diamond uses commit timestamps from the timestamp service (tss) rather than TrueTime [17].

As shown in Figure 5 (left), transactions progress through two phases, *execution* and *commit*. During the execution phase, LIBDIAMOND runs the application code in the transaction closure passed into `txn_execute`. It runs the code locally on the LIBDIAMOND client node (i.e., not on a storage node like a stored procedure).

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

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

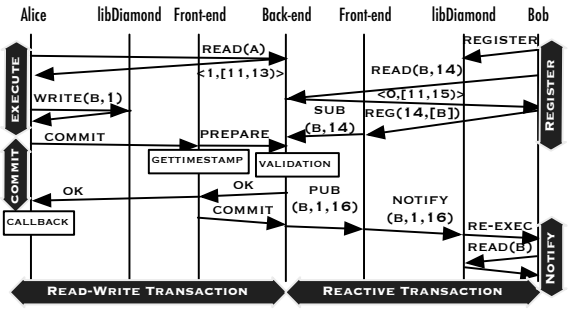

Figure 5: **Diamond transaction coordination.** Left: Alice executes a read-write transaction that reads `A` and writes `B`. Right: Bob registers a reactive transaction that reads `B` (we omit the `txn_id`). When Alice commits her transaction, the back-end server publishes the update to the front-end, which pushes the notification and the update to Bob's LIBDIAMOND, which can then re-execute the reactive transaction locally.

2. Each participant runs a DOCC validation check (described in Section 5); if DOCC validation succeeds, the participant adds the transaction to a *prepared list* and returns `true`; otherwise, it returns `false`.
3. As an optimization, the front-end server concurrently retrieves a commit timestamp from the tss.
4. If all participants respond `true`, the front-end sends `Commits` to the participants with the commit timestamp; otherwise, it sends `Aborts`. Then, it returns the transaction outcome to the LIBDIAMOND client.

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

4.3.2 Managing Reactive Transactions

As shown in Figure 5 (right), when an application registers a reactive transaction, the LIBDIAMOND client: (1) gives the reactive transaction a `txn_id`, (2) executes the reactive transaction at its latest known timestamp, and (3) sends the `txn_id`, the timestamp, and the read set in a `Register` request to the front-end server. For each key in the read set, the front-end server creates a `Subscribe` request and sends those requests, along with the timestamp, to each key's back-end partition.

For efficiency, LIBDIAMOND tracks read set changes between executions and re-registers. We expect each reactive transaction's read set to change infrequently, reducing the overhead of registrations; if it changes often, we can use other techniques (e.g., `map_objectrange` described in Section 6.2) to improve performance.

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

1. The leader in the partition sends a `Publish` request with the transaction's commit timestamp to each front-end subscribed to the updated record.
2. For each `Publish`, the front-end server looks up the reactive transactions that have the updated record in their read sets and checks if the commit timestamp

is bigger than the last notification sent to that client.
3. If so, the front-end server sends a `Notify` request to the client with the commit timestamp and the reactive transaction id.
4. The client logs the notification on receipt, updates its latest known timestamp, and re-executes the reactive transaction at the commit timestamp.

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

4.3.3 Handling Failures

While both the back-end storage and tss are replicated using VR, Diamond can suffer failures of the LIBDIAMOND clients or front-end servers. On client failure, LIBDIAMOND runs a *client recovery protocol* using its transaction log to ensure that read-write transactions eventually commit. For each completed but unprocessed transaction (i.e., in the log but with no outcome), LIBDIAMOND retries the commit. If the cloud store has a record of the transaction, it returns the outcome; otherwise, it re-runs 2PC. For each reactive transaction, the application re-registers on recovery. LIBDIAMOND uses its log to find the last timestamp at which it ran the transaction.

Although front-end servers are stateless, LIBDIAMOND clients must set up a new front-end server connection when they fail. They use the client recovery protocol to do this and re-register each reactive transaction with its latest notification timestamp. Front-end servers also act as coordinators for 2PC, so back-end storage servers use the *cooperative termination protocol* [11] if they do not receive `Commit` requests after some timeout.

5 Wide-area Optimizations

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

5.1 Data-type Optimistic Concurrency Control

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

DOCC tackles this issue in two ways. First, it uses fine-grained concurrency control based on the *semantics* of reactive data types, e.g., allowing concurrent updates to different list elements. Second, it uses conflict-free data types with commutative operations, such as counters and ordered sets. As noted in Section 4.3.1, LIBDIAMOND collects an operation set for every data type operation during the transaction's execution phase. For each operation, it collects the key and table. It also collects the read version for every `Get`, the written value for every `Put`, the index (e.g., list index or hash table key) for every collection operation, and the diff (e.g., the increment value or the insert or append element) for every commutative CRDT operation. We show in Section 6 that although fine-grained tracking slightly increases DOCC overhead, it improves overall performance.

Table 4: **DOCC validation matrix.** Matrix shows whether the committing transaction can commit (C) or must abort (A) on conflicts. Each column is further divided by the isolation level (RC=read committed, SI=snapshot isolation, SS=strict serializability). Commutative CRDT operations have the same outcome.

Committing Op \ Prepared Op / Isolation Level	read			write			CRDT op		
	RC	SI	SS	RC	SI	SS	RC	SI	SS
read	C	C	C	C	C	A	C	C	A
write	C	C	A	C	A	A	C	A	A
CRDT op	C	C	A	C	A	A	C	C	C

Using operation sets, DOCC runs a *validation* procedure that checks every committing transaction for potential violations of isolation guarantees. A *conflicting access* occurs for an operation if the table, key, and index (for collection types) match an operation in a prepared transaction. For a read, a conflict also occurs if the latest write version (or commutative CRDT operation) to the table, key, and index is bigger than the read version. For each, DOCC makes an abort decision, as noted in Table 4.

Since transactions that contain only commutative operations can concurrently commit, DOCC can allow many concurrent transactions that modify the same keys. This property is important for workloads with high write contention, e.g., the Twitter "like" counter for popular celebrities [36]. Further, because Diamond runs read-only and reactive transactions in serializable snapshot mode, they do not conflict with read-write transactions with commutative CRDT operations.

5.2 Client Caching with Bounded Validity Intervals

Some clients in the wide-area setting may occasionally be unavailable, making it impossible to atomically invalidate all cache entries on every write to enforce strong ordering. Diamond therefore uses multi-versioning in both the client-side cache and back-end storage to enforce a *global ordering of transactions*. To do this, it tags each version with a *validity interval* [62], which begins at the *start timestamp* and is terminated by the *end timestamp*. In Diamond's back-end storage, a version's start timestamp is the commit timestamp of the transaction that wrote the

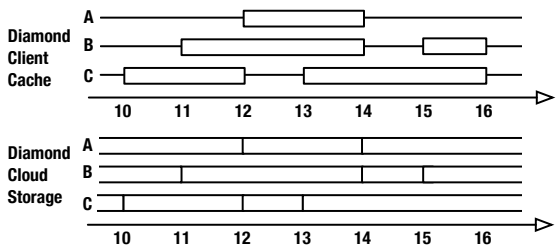

Figure 6: **Diamond versioned cache.** Every Diamond client has a cache of the versions of records stored by the Diamond cloud storage system. The bottom half shows versions for three keys (A, B and C), and the top half shows cached versions of those same keys. Note that the cache is missing some versions, and all of the validity intervals in the cache are bounded.

version. The end timestamp is either the commit timestamp of the transaction writing the *next* version (making that version out-of-date) or *unbounded* for the latest version. Figure 6 shows an example of back-end storage with three keys.

On reads, the Diamond cloud tags the returned value with a validity interval for the LIBDIAMOND client-side cache. These validity intervals are conservative; back-end storage *guarantees* that the returned version is valid *at least* within the validity interval, although it may be valid beyond. If the version is the latest, back-end storage will *bound* the validity interval by setting the end timestamp to the latest commit timestamp of a transaction that accessed that record. For example, in Figure 6, the validity interval of the latest version of `B` and `C` are capped at timestamp 16 in the cache, while they are unbounded in storage. Most importantly, bounded validity intervals *eliminate* the need for cache invalidations because the version is always valid within the validity interval. Diamond eventually garbage collects cached versions as they become too outdated to use.

5.3 Data Push Notifications

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

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

Table 5: **Application comparison.** Diamond both reduces code size and adds to the application's ACID+R guarantees.

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

6 Experience and Evaluation

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

6.1 Prototype Implementation

We implemented a Diamond prototype in 11,795 lines of C++, including support for C++, Python and Java language bindings on both x86 and ARM. The Java bindings (939 LoC) use javacpp [39], and the Python bindings (115 LoC) use Boost [2]. We cross-compiled Diamond and its dependencies for Android using the NDK standalone toolchain [29]. We implemented most Diamond data types, but not all are supported by DOCC. Our current prototype does not include client-side persistence and relies on in-memory replication for the back-end store; however, we expect disk latency on SSDs to have a low performance impact compared to wide-area network latency, with NVRAM reducing storage latency even further in the future.

6.2 Programming Experience

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

6.2.1 Simplifying Reactive Applications

To evaluate Diamond's programming benefits, we implemented applications both with and without Diamond. Table 5 shows the lines of code for both cases. For all of the apps, Diamond simultaneously decreased program size and added important reliability or correctness properties. We briefly describe the programs and results below.

100 Game. Our non-Diamond version of the 100 game is based on the design in Figure 1. For simplicity, we used Redis [67] for both storage and notifications. We found

several data races between storage updates and notifications when running experiments for Figure 9, forcing us to include updates in the notifications to ensure clients did not read stale data from the store. The Diamond version eliminated these bugs and the complexities described in Section 2 and guaranteed correctness with atomicity and isolation; in addition, it reduced the code size by 26%.

Chat Room. As another simple but representative example of a reactive app, we implemented two versions of a chat room. Our version with explicit data management used Redis for storage and the Jetty [40] web server to implement a REST [25] API. It used `POST` requests to send messages and polled using `GET` requests for displaying the log. This design is similar to that used by Twitter [80, 35] to manage its reactive data (e.g., Twitter has `POST` and `GET` requests for tweets, timelines, etc.). The Diamond version used a `StringList` for the chat log, a read-write transaction to append messages, and a reactive transaction to display the log. In comparison, Diamond not only eliminated the need for a server or storage system, it also provided atomicity (the Redis version has no failure guarantees), isolation (the Redis version could not guarantee that all clients saw a consistent view of the chat log), and reactivity (the Redis version polled for new messages). Diamond also shrunk the 355-line app by 130 lines, or 33%.

PyScrabble and Diamond Scrabble. To evaluate the impact of reactive data management in an existing application, we built a Diamond version of PyScrabble [16], an open-source, multiplayer Scrabble game. The original PyScrabble does not implement persistence (i.e., it has no storage system) and uses a centralized server to process moves and notify players. The centralized server enforces isolation and consistency only if there are no failures. We made some changes to add persistence and accommodate Diamond's transaction model. We chose to directly `rmap` the Scrabble board to reactive data types and update the UI in a reactive transaction, so our implementation had to commit and share every update to make it visible to the user; thus, other users could see the player lay down tiles in real-time rather than at the end of the move, as in the original design. Overall, our port of PyScrabble to Diamond removed over 1000 lines of code from the 8700-line app (13%) while transparently simplifying the structure (removing the server), adding fault tolerance (persistence) and atomicity, and retaining strong isolation.

Twimight and Diamond Dove. As another modern reactive application, we implemented a subset of Twitter using an open-source Android Twitter client (Twimight [79]) and a custom back-end. The Diamond version eliminated much of the data management in the Twimight version, i.e., pushing and retrying updates to the server and maintaining consistency between a client-side SQLite [71] cache and back-end storage. Diamond directly plugged into UI elements and published updates with read-write transactions. As a result, it simplified the design, eliminated 1700 lines (13%) from the 14K-line application, transparently provided stronger atomicity and isolation guarantees, and eliminated inconsistent behaviors (e.g., a user seeing a retweet before the original tweet).

6.2.2 Simplifying Reactive Libraries

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

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

These library functions were easy to build (under 75 lines of code) and greatly simplified several applications; for example, our Diamond Twitter implementation stores a user's timeline as a `LongList` of tweet ids and uses `map_objectrange` to directly bind the tail of the user's timeline into a custom Android adapter, which then plugs into the Twimight Android client and automatically manages reactivity. In addition to reducing application complexity, these abstractions also provide valuable hints for prefetching and for how reactive transaction read sets might change. Overall, we found Diamond's programming model to be extremely flexible, powerful, and easy to generalize into widely useful libraries.

6.3 Performance Evaluation

Our performance measurements demonstrate that Diamond's automated data management and strong consistency impose a low performance cost relative to custom-

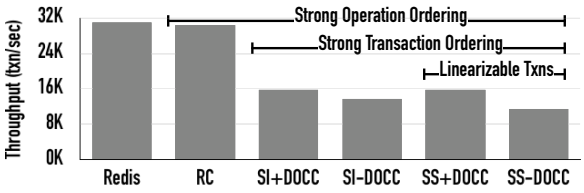

Figure 7: **Peak throughput for explicit data management vs Diamond.** We compare an implementation using Redis and Jetty to Diamond at different isolation levels with and without DOCC. We label the ordering guarantees provided by each configuration. In all cases, the back-end servers were the bottleneck.

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

6.3.1 Experimental Setup

We ran experiments on Google Compute Engine [30] using 16 front-end servers and 5 back-end partitions, each with 3 replicas placed in different availability zones in the same geographic region (US-Central). Our replication protocol used adaptive batching with a batch size of 64. We placed clients in a different geographic region in the same country (US-East). The latency between zones was ≈1 ms, while the latency between regions was ≈36 ms. For our mobile device experiments, we used Google Nexus 7 LRX22G tablets connected via Wi-Fi and, for desktop experiments, we used a Dell workstation with an Intel Xeon E5-1650 CPU and 16 GB RAM.

We used a benchmark based on Retwis [45], a Redis-based Twitter clone previously used to benchmark transactional storage systems [84]. The benchmark was designed to be a representative, although not realistic, reflection of a Twitter-like workload that provides control over contention. It ran a mix of five transactions that range from 4-21 operations, including: loading a user's home timeline (50%), posting a tweet (20%), following a user (5%), creating a new user (1%), and "like"-ing a tweet (24%). To increase contention, we used 100K keys and a Zipf distribution with a co-efficient of 0.8.

6.3.2 Overhead of Automated Data Management

For comparison, we built an implementation of the Retwis benchmark that explicitly manages reactive data using Jetty [40] and Redis [67]. The Redis `WAIT` command offers synchronous in-memory replication, which matches Diamond's fault-tolerance guarantees but provides no operation or transaction ordering [66]. The leftmost bar in Figure 7 shows the peak Retwis throughput of 31K trans./sec. for the Redis-based implementation, while the second bar in Figure 7 shows the Diamond read-committed (RC) version, whose performance (30.5K trans./sec.) is nearly identical. Unlike the Redis-based implementation, however, the Diamond benchmark provides strong consistency based on VR, i.e., it enforces a single global order of operations but not transactions. The Diamond version also provides all of its reactivity support features. Diamond therefore provides better consistency properties and simplifies programming at little additional cost.

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

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

6.3.3 Benefit of DOCC

DOCC's benefit depends on both contention and transaction duration. To evaluate this effect, we measured the throughput improvement of DOCC for each type of Retwis transaction with at least one CRDT operation (Figure 8).

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

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

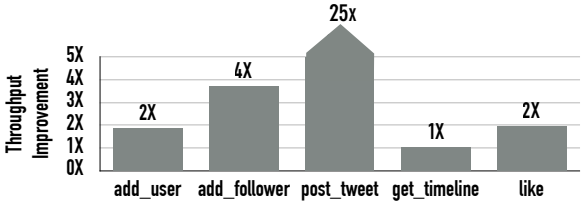

Figure 8: **Throughput improvement with DOCC for each Retwis transaction type.**

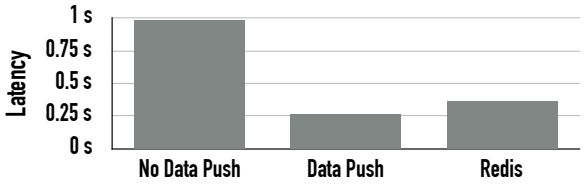

Figure 9: **Latency comparison for 100 game rounds with data push notifications.** Each round consist of 1 move by each of 2 players; latency is measured from 1 client. We implemented explicit data management and notifications using Redis and Diamond notifications with and without batched updates.

6.3.4 Benefit of Data Push Notifications

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

Data push notifications reduce this latency by batching updates with reactive transaction notifications to populate the client-side cache. We turned our implementation of the 100 game from Figure 3 into a benchmark: two players join each game, and players make a move as soon as the other player finishes (i.e., zero "think" time). This experiment is ideal because the read set of the reactive transaction does not change, and it overlaps with the read set of the read-write transaction. We also design an implementation using Redis, where notifications carry updates to clients as a manual version of data push notifications. We measure the latency from one player's client for each player to take a turn or for one *round* of the game. Figure 9 shows that data push notifications reduce the overall latency by almost 50% by eliminating wide-area reads for both the reactive and read-write transactions in the game. As a result, Diamond has 30% lower latency and stronger transactional guarantees than our Redis implementation.

6.3.5 Impact of Wide-area Storage Server Failures

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

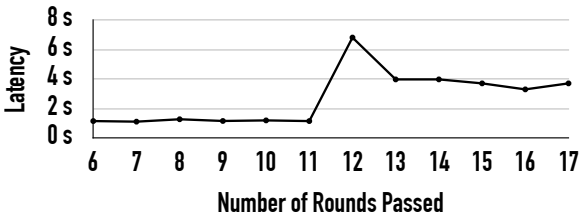

Figure 10: **Latency of 100 game rounds during failure.** We measured the latency for both players to make a move and killed the leader of the storage partition after about 15 seconds. After recovery, the leader moves to another geographic region, increasing overall messaging latency on each move.

Figure 10 shows the latency of each round. Note that the latency is higher than that in the previous experiment because the VR leader has to wait for a response from a quorum of replicas, which take at least 100 ms, and up to 150 ms, to contact. About 15 seconds into the game, we kill the leader in US-Central, switching it to Europe. The latency of each round increases to almost 4 seconds afterwards: the latency between the front-end servers and the leader in Europe increases to 100 ms, and the latency from the leader to the remaining replica in Asia increases to 250 ms. Despite this, the round during the failure takes only 7 seconds, meaning that Diamond can detect the failure and replace the leader in less than 3 seconds.

6.3.6 End-user Application Latency

To evaluate Diamond's impact on the user experience, we measure the latency of user operations in two apps from Section 6.2 built with and without Diamond. PyScrabble is a desktop application, while our Chat Room app runs on Android. The ping times to the Diamond cloud were ≈38 ms on the desktop and ≈46 ms on the Android tablet.

Figure 11 (left) shows two operations for PyScrabble: `MakeMove` commits a transaction that updates the user's move, and `DisplayMove` includes MakeMove plus the notification and reactive transaction to make it visible. Compared to the original PyScrabble, Diamond's latency is slightly higher (9% and 16%, respectively). Figure 11 (right) shows operations for the Chat Room on an Android tablet. `ReadLog` gets the full chat log, and `PostMessage` gets the chat log, appends a message, and commits it back. The Diamond version is a few percent faster than the Redis version because it runs in native C++, while the Redis version uses a Java HTTP client. Overall, we found the latency differences between Diamond and non-Diamond operations were not perceivable to users.

7 Related Work

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

Several commercial platforms [51, 26, 60] provide

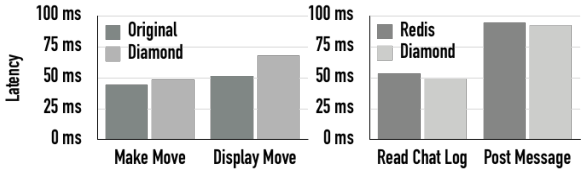

Figure 11: **End-user operation latency for PyScrabble and Chat Room on Diamond and non-Diamond implementations.**

an early form of reactive data management without distributed transactions. Other open source projects [38, 55, 21, 59, 70] have replicated the success of their commercial counterparts. Combined, they comprise a mobile back-end market of $1.32 billion dollars [49].

However, these products do not meet the requirements of reactive applications, still requiring programmers to address failures and race conditions. Meteor [51] lets client-side code directly access the database interface. However, because it uses existing databases (MongoDB [53], and most recently, Postgres [63]) that do not support distributed transactions and offer weak consistency guarantees by default, programmers must still reason about race conditions and consistency bugs. Parse [60] and Firebase [26] similarly enable clients to read, write, and subscribe to objects that are automatically synchronized across mobile devices; however, these systems offer no concurrency control or transactions. As demonstrated by these Stack Overflow questions [56, 50], programmers find this to be a significant issue with these systems. Diamond addresses this clear developer need by providing ACID+R guarantees for reactive applications.

There has been significant work in wide-area storage systems for distributed and mobile applications, including numerous traditional instantiations [77, 42, 57] as well as more recent work [18, 9, 74, 61, 75]. Many mobile applications today use commercial storage services such as Dropbox and others [23, 22, 37], while users can also employ revision-based storage (e.g., git [27]). Applications often combine distributed storage with notifications [3, 6]. As discussed, these systems help with data management, but none offers a complete solution.

Diamond shares a data-type-based storage model with data structure stores [67, 68]. Document stores (e.g., MongoDB [53]) support application objects; this prevents them from leveraging semantics for better performance. These datastores, along with more traditional key-value and relational storage systems [15, 8, 44, 76], were not designed for wide-area use although they could support reactive applications with additional work.

Reactive transactions in Diamond are similar to database triggers [47], events [14], and materialized views [12]. They differ from these mechanisms because they modify local application state and execute application code rather than database queries that update storage state. Diamond's design draws on Thialfi [3]; however, Thialfi cannot efficiently support data push notifications without insight into the application's access patterns.

DOCC is similar to Herlihy [32, 31] and Weihl's [83] work on concurrency control for abstract data types. However, Diamond applies their techniques to CRDTs [69] over a range of isolation levels in the wide area. DOCC is also related to MDCC [43] and Egalitarian Paxos [54]; however, DOCC uses commutativity for transactional concurrency control rather than Paxos ordering and supports more data types. DOCC extends recent work on software transactional objects [33] for single-node databases to the wide area; integrating the two would let programmers implement custom data types in Diamond.

Diamond does not strive to support a fully reactive, data-flow-based programming model, like functional reactive or constraint-based programming [82, 7]; however, reactive transactions are based on the idea of change propagation. Recent interest in reactive programming for web client UIs has resulted in Facebook's popular React.js [64], the ReactiveX projects [65], and Google's Agera[28]. DREAM [48], a recently proposed, distributed reactive platform, lacks transactional guarantees. Sapphire [85], another recent programming platform for mobile/could applications, does not support reactivity, distributed transactions, or general-purpose data management.

8 Conclusion

This paper described Diamond, the first data management service for wide-area reactive applications. Diamond introduced three new concepts: the `rmap` primitive, reactive transactions, and DOCC. Our evaluation demonstrated that: (1) Diamond's programming model greatly simplifies reactive applications, (2) Diamond's strong transactional guarantees eliminate data race bugs, and (3) Diamond's low performance overhead has no impact on the end-user.

9 Acknowledgements

We thank the UW systems lab for their comments throughout the project. This work was supported by Google, National Science Foundation grant CNS-1518702 and NSF GRFP, and MSR Ph.D. fellowships. We also thank our anonymous reviewers and our shepherd, Bryan Ford, for their feedback.

References

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

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

[3] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a client notification service for internet-scale applications. In *Proc. of SOSP*, 2011.

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

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

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

[7] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[8] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR*, 2011.

[9] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADS: A policy architecture for distributed storage systems. In *Proc. of NSDI*, 2009.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of PPOPP*, 1990.

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

[12] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. of SIGMOD*, 1986.

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

[14] S. Chakravarthy. Sentinel: an object-oriented DBMS with event-based rules. In *Proc. of SIGMOD*, 1997.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008.

[16] K. Conaway. Pyscrabble. http://pyscrabble.sourceforge.net/.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, 2012.

[18] M. Dahlin, L. Gao, A. Nayate, A. Venkataramana, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of NSDI*, 2006.

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

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007.

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

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

[23] Dropbox, 2015. http://www.dropbox.com.

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

[25] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

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

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

[28] Google. Agera. https://github.com/google/agera.

[29] Google. Android standalone toolchain, 2016. http://developer.android.com/ndk/guides/standalone_toolchain.html.

[30] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[31] M. Herlihy. Optimistic concurrency control for abstract data types. In *Proc. of PODC*. ACM, 1986.

[32] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 1990.

[33] N. Herman, J. P. Inala, Y. Huang, L. Tsai, E. Kohler, B. Liskov, and L. Shrira. Type-aware transactions for faster concurrent code. In *eurosys*, 2016.

[34] T. Hoff. Playfish's social gaming architecture - 50 million monthly users and growing. High Scalability, Sept 2010. highscalability.com/blog/2010/9/21/playfishs-social-gaming-architecture-50-million-monthly-user.html.

[35] T. Hoff. The architecture that Twitter uses to deal with 150m active users, 300k qps, a 22 mb/s firehose, and send tweets in under 5 seconds. High Scalability, July 2013. http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html.

[36] M. Humphries. Ellen DeGeneres crashes Twitter with Oscar selfie, 2014. http://www.geek.com/mobile/ellen-degeneres-crashes-twitter-with-an-oscars-selfie-1586464/.

[37] Apple iCloud, 2016. https://www.icloud.com/.

[38] A. Incubator. Apache Usergrid. http://usergrid.apache.org/.

[39] JavaCPP: The missing bridge between Java and native C++. github, Mar 2016. https://github.com/bytedeco/javacpp.

[40] Jetty web server. http://www.eclipse.org/jetty/.

[41] R. K. Jonas Boner, Dave Farley and M. Thompson. The reactive manifesto, Sept 2014. http://www.reactivemanifesto.org/.

[42] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 1992.

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

[44] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

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

[46] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 1989.

[47] B. F. Lieuwen, N. Gehani, and R. Arlein. The Ode active database: trigger semantics and implementation. In *Proc. of ICDE*, Feb 1996.

[48] A. Margara and G. Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proc. of DEBS*. ACM, 2014.

[49] Markets and Markets. Backend as a service (BaaS) market worth 28.10 billion USD by 2020. http://www.marketsandmarkets.com/PressReleases/baas.asp.

[50] martypdx. Firebase data consistency across multiple nodes. Stack Overflow, Apr 2015. http://stackoverflow.com/questions/29947898/firebase-data-consistency-across-multiple-nodes.

[51] Meteor, 2015. http://www.meteor.com.

[52] M. Mode. New mobile apps revolutionize how organizations respond to crises and operations issues, Aug 2014. http://www.missionmode.com/new-mobile-apps-revolutionize-organizations-respond-crises-operations-issues/.

[53] MongoDB, 2015. https://www.mongodb.org.

[54] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of SOSP*, 2013.

[55] Mozilla. Kinto. http://kinto.readthedocs.org/en/latest/.

[56] R. Mulia. Firebase - maintain/guarantee consistency. Stack Overflow, Jan 2016. http://stackoverflow.com/questions/34678083/firebase-maintain-guarantee-data-consistency.

[57] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of SOSP*, 2001.

[58] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[59] openio. openio.io: object storage grid for apps. http://openio.io/.

[60] Parse, 2015. http://www.parse.com.

[61] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: tunable end-to-end data consistency for mobile apps. In *Proc. of EuroSys*, 2015.

[62] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proc. of OSDI*, 2010.

[63] PostgreSQL, 2013. http://www.postgresql.org/.

[64] React: A JavaScript library for building user interfaces. Github, 2016. https://facebook.github.io/react/.

[65] ReactiveX: An api for asynchronous programming with observable streams, 2016. http://reactivex.io/.

[66] Redis. Wait numslaves timeout. http://redis.io/commands/WAIT.

[67] Redis: Open source data structure server, 2013. http://redis.io/.

[68] Riak, 2015. http://basho.com/products/riak-kv/.

[69] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. of SSS*, 2011.

[70] socketcluster.io. socketcluster.io: a scalable framework for realtime apps and microservices. http://socketcluster.io/#!/.

[71] Sqlite home page, 2015. https://www.sqlite.org/.

[72] Square cash. https://cash.me/.

[73] M. Stonebraker and J. M. Hellerstein. *Readings in Database Systems*. Morgan Kaufmann San Francisco, 1998.

[74] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and M. F. Kaashoek. Eyo: Device-transparent personal storage. In *Proc. of USENIX ATC*, 2011.

[75] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proc. of NSDI*, 2009.

[76] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with Project Voldemort. In *Proc. of FAST*, 2012.

[77] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP*, 1995.

[78] Global social gaming market to reach US$17.4 bn by 2019 propelled by rising popularity of fun games. Transparency Market Research Press Release, Sept 2015. http://www.transparencymarketresearch.com/pressrelease/social-gaming-market.htm.

[79] Twimight open-source Twitter client for Android, 2013. http://code.google.com/p/twimight/.

[80] Twitter. Twitter developer API, 2014. https://dev.twitter.com/overview/api.

[81] Venmo. https://venmo.com/.

[82] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proc. of PLDI*, 2000.

[83] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Prog. Lang. Syst.*, 1989.

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

[85] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proc. of OSDI*, 2014.