Certified Program Models for Eventual Consistency

Edgar Pek, Pranav Garg, Muntasir Raihan Rahman, Indranil Gupta, P. Madhusudan

University of Illinois at Urbana-Champaign

{pek1, garg11, mrahman2, indy, madhu}@illinois.edu

Paper Type: Research, Long

Abstract

We present a new approach, *certified program models*, to establish correctness of distributed protocols. We propose modeling protocols as programs in standard languages like C, where the program simulates the processes in the distributed system as well as the nondeterminism, the communication, the delays, the failures, and the concurrency in the system. The program model allows us to test the protocol as well as to verify it against correctness properties using program verification techniques. The highly automated testing and verification engines in software verification give us the tools needed to establish correctness. Furthermore, the model allows us to easily alter or make new design decisions, while testing and verifying them.

We carry out the above methodology for the distributed key-value store protocols underlying widely used frameworks such as Dynamo [30], Riak [2] and Cassandra [4]. We model the read-repair and hinted-handoff table based recovery protocols as concurrent C programs, test them for conformance with real systems, and then verify that they guarantee eventual consistency, modeling precisely the specification as well as the failure assumptions under which the results hold. To the best of our knowledge, this is the first verification technique that shows correctness of these distributed protocols using mostly-automated verification.

1. Introduction

Distributed systems are complex software systems that pose myriad challenges to formally verifying them. While many distributed protocols running in these systems stem from research papers that describe a core protocol (e.g., Paxos), their actual implementations are known to be much more complex (the "Paxos Made Live" paper [25] shows how wide this gap is).

The aim of this paper is to strike a middle-ground in this spectrum by verifying models of actual protocols implemented in systems. We propose a new methodology, called *certified programs models*, where we advocate that the fairly complex protocols in distributed systems be modeled using *programs* (programs written in traditional systems languages, like C with concurrency), and certified to be correct against it's specifications.

The idea is to model the entire distributed system in software, akin to a software simulator of the system. The model captures the distributed processes, their memory state, the secondary storage state, the communication, the delays, and the failures, using non-determinism when necessary.

The salient aspects of this modeling are that it provides:

(a) a modeling language (a traditional programming language) to model the protocols precisely,

(b) an *executable* model that can be validated for accuracy with respect to the system using testing, where the programmer can write test harnesses that control inputs as well as physical events such as node and network failures, and test using mature systematic testing tools for concurrent software, like CHESS [52, 53].

(c) an accurate modeling of specifications of the protocol using *ghost state* in the program as well as powerful assertion logics, and

(d) a program model that lends itself to *program verification techniques*, especially using tools such as VCC [29] that automate large parts of the reasoning using logical constraint solvers.

In this paper, we explore the certified model paradigm for modeling, testing, and formally proving properties of core distributed protocols that underlie eventually consistent distributed key-value/NoSQL stores. Eventually consistent key-value stores originated with the Dynamo system from Amazon [30] and are today implemented in systems such as Riak [2], Cassandra [4], and Voldemort [10]. We show how

Submission to SoCC '15, October, 2015, Hawaii.

to build program models for them in concurrent C, test them for conformance to the intended properties of the systems by using automated testing tools like CHESS [52, 53], and formally verify the eventual consistency property for them using VCC [29], a verification tool for concurrent C.

1.1 Key-value/NoSQL Storage Systems and Eventual Consistency

Key-value/NoSQL stores are on the rise [8] and are used today to store Big Data in many companies, e.g., Netflix, IBM, HP, Facebook, Spotify, PBS Kids, etc. rely heavily on the Cassandra key-value store system while Riak is used by BestBuy, Comcast, the NHS UK, The Danish Health and Medicines Authority for patient information, and Rovio, the gaming company behind AngryBirds.

Key-value/NoSQL storage systems arose out of the CAP theorem/conjecture, which was postulated by Brewer [20, 21] (a proof under a particular model is given by Gilbert and Lynch [34, 49]). The conjecture states that a distributed storage system can choose at most two out of three important characteristics— strong data Consistency (i.e., linearizability or sequential consistency), Availability of data (to reads and writes), and Partition-tolerance. Hence achieving strong consistency while at the same time providing availability in a partitioned system with failures is impossible.

While traditional databases preferred consistency and availability, the new generation of key-value/NoSQL systems are designed to be partition-tolerant in order to handle highly distributed partitions that arise due to the need of distributed access, both within a datacenter as well as across multiple data-centers. As a result, a key-value/NoSQL system is forced to chose between one of either strong consistency or availability— the latter option providing low latencies for reads and writes.

Key-value/NoSQL systems that prefer availability include Cassandra [43], Riak [2], Dynamo [30], and Voldemort [10], and support weak models of consistency (e.g., eventual consistency). Other key-value/NoSQL systems instead prefer strong consistency, e.g., HBase [7], Bigtable [26], and Megastore [17], and may be unavailable under failure scenarios.

One popular weak consistency notion is eventual consistency, which roughly speaking, says that if no further updates are made to a given data item, all replicas will eventually hold the same value (and a read would then produce this value). Eventual consistency is a *liveness property*, not a safety property [16]. The precise notion of what eventual consistency means in these protocols (the precise assumptions under which they hold, the failure models, the assumptions on the environment, etc.) are not well understood, let alone proven. Programmers also do not understand the subtleties of eventually consistent stores; for instance, default modes in Riak and Cassandra can permanently lose writes— this is dangerous, and has been exploited in a recent attack involving BitCoins [9].

1.2 Contributions of this Paper

The primary contribution of this paper is to precisely reason about the guarantees of eventual consistency that real implementations of key-value stores provide. We model two core protocols in key-value stores as programs, the *hinted-handoff protocol* and the *read-repair* protocol, which are anti-entropy mechanisms first proposed in the Amazon Dynamo system [30], and later implemented in systems such as Riak [2] and Cassandra [4].

We build certified program models— program models for these protocols written in concurrent C and that are verified for eventual consistency. The program uses *threads* to model concurrency, where each get/put operation as well as the asynchronous calls they make are modeled using concurrently running threads. The state of the processes, such as stores at replicas and the hinted-handoff tables, are modeled as shared arrays. Communication between processes is also modeled using data-structures: the network is simulated using a set that stores pending messages to replicas, with an independent thread sending them to their destinations. Failures and non-determinism of message arrivals, etc., are also captured programmatically using non-determinism (modeled using stubs during verification and using random coin-tosses during testing). In particular, system latency is captured by threads that run in the background and are free to execute anytime, modeling arbitrarily long delays.

In the case of the *hinted-handoff protocol*, we prove that this protocol working alone guarantees eventual consistency provided there are only transient faults. In fact, we prove a stronger theorem by showing that for any operation based (commutative) conflict-free replicated data-type implementing a register, the protocol ensures *strong eventual consistency*— this covers a variety of schemes that systems use, including Riak and Cassandara, to resolve conflict when implementing a key-value store. Strong eventual consistency guarantees not only eventual consistency, but that the store always contains a value that is a function of the set of updates it has received, independent of the order in which it was received. We prove this by showing that the hinted-handoff protocol (under only transient failures) ensures *eventual delivery* of updates; this combined with an idempotent CmRDT [57, 59] implementing a register ensures strong eventual consistency. We model the eventual delivery property in the program model using a ghost *taint* that taints a particular write at a coordinator (unbeknownst to protocol), and asserts that the taint propagates eventually to every replica. Eventual delivery is a *liveness property*, and is established by finding a ranking function that models abstractly the time needed to reach a consistent state, and a slew of corresponding safety properties to prove this program correct.

For the *read-repair protocol*, we first believed the popularly-held opinion that a read-repair (issued during a read) would bring the nodes that are alive to a consistent state eventually, and tried to prove this property. However, while working on the proof, we realized that there is no invariant that can prove this property, and this made us realize that the property in fact does not hold. A single read is insufficient, and we hence prove a more complex property: at any point, if a set of nodes are alive and they all stay alive, and if all requests stop except for an unbounded sequence of reads to a key, then the live nodes that are responsible for the key will eventually converge.

Note that the certification that the program models satisfy their specification is for an *unbounded* number of threads, which model an unbounded number of replicas, keys, values, etc., model arbitrarily long input sequences of updates and reads to the keys, and model the concurrency prevalent in the system using parallelism in the program. The verification is hence a *complete* verification as opposed to several approaches in the literature which have used under-approximations in order to systematically test a bounded-resource system [48, 51, 54, 55]. In particular, Amazon has reported modeling of distributed protocols using TLA, a formal system, and used model-checking (systematic testing) on bounded instances of the TLA system to help understand the protocols, check their properties, and help make design decisions. Our results, in contrast, model protocols using C programs, which we believe are much simpler for systems engineers to use to model protocols, and being executable, is easy to test using test harnesses. Most importantly, we have proved the entire behavior of the protocol correct (as opposed to the work using TLA) using the state-of-the-art program verification framework VCC [29] that automates several stages of the reasoning.

We also give an account of our experience in building certified program models (Section 6). In addition to resulting in proven models, there were several other side benefits that resulted, including a vocabulary of reasoning that the model provided, a nuanced accurate formalization of the assumptions under which eventual consistency holds, as well as helping us realize that certain specifications that we believed in did *not* hold.

The paper is structured as follows. Section 2 describes key-value stores, eventual consistency, and the main anti-entropy protocols that are implemented in systems and that we study in this paper (readers familiar with these topics can choose to skip this section). We describe our main results in Section 3, where we describe the precise property we prove for the protocol models as well as some properties that we expected to be initially true, but which we learned were not true through our experience. Section 4 describes our models of protocols using programs in detail, including the testing processes we used to check that our model was reasonable. The entire verification process, including background on program verification, the invariants and ranking functions required for proving the properties, etc., are given in Section 5. A gist of the effort we put in, the experience we had, and the lessons we learned are described in Section 6. Section 7 describes related work and Section 8 concludes with interesting directions for future work.

2. Background

In this section we describe in detail the read and write paths involved in a key-value store, and the anti-entropy mechanisms which are used to implement eventual consistency by reconciling divergent replicas. Readers familiar with key-value store system internals can skip this section without loss of continuity.

2.1 Key-value stores

Key-value stores have a simple structure. They store pairs of keys and values, and they usually have two basic operations: get(key) for retrieving the value corresponding to the key, and put(key, value) for storing the value of a particular key[1]. Key-value stores typically use consistent hashing [39] to distribute keys to servers, and each key is replicated across multiple servers for fault-tolerance. When a client issues a put or get operation, it first interacts with a server (e.g., the server closest to the client). This server plays the role of a *coordinator*: it coordinates the client and replica servers to complete *put* and *get* operations. The CAP theorem [21] implies that under network partitions (the event where the set of servers splits into two groups with no communication across groups), a key-value store must choose either consistency (linearizability) [36] or availability. Even when the network is not partitioned, the system is sometimes configured to favor latency over consistency [11]. As a result, popular key-value stores like Apache Cassandra [43] and Riak [2] expose tunable *consistency levels*. These consistency levels control the number of processes the coordinator needs to hear from before returning and declaring success on reads and writes. For instance, a write threshold of one, would allow the system to return with success on a write when it has successfully written to just one replica. When the sum of read and write thresholds is greater than the number of replicas, the system will ensure strong consistency. Consistency levels weaker than quorum are the most popular since they achieve low latency (e.g., Amazon [1]).

2.2 Eventual Consistency

In general, a consistency model can be characterized by restrictions on operation ordering. The strongest models, e.g., linearizability [36] severely restrict the possible orderings of operations that can lead to correct behavior. Eventual consistency lies at the opposite end of the spectrum; it is the weakest possible consistency model. Informally, it guarantees that, if no further updates are made to a given data item, reads to that item will eventually return the same value [64]. Thus until some undefined time in the future when the system is supposed to converge, the user can never rule out

[1] We use both read/write and get/put terms to mean data fetch and data update operations.

the possibility of data inconsistency. Despite the lack of any strong guarantees, many applications have been successfully built on top of eventually consistent stores. Most stores use some variation of anti-entropy [31] protocols to implement eventual consistency mechanisms.

2.3 Anti-entropy Protocols

To achieve high availability and reliability, key value stores typically replicate data on multiple servers. For example, each key can be replicated on *N* servers, where *N* is a configurable parameter. In the weakest consistency setting (consistency level that has read and write thresholds of one), each get and put operation only touches one replica (e.g., the one closest to the coordinator). Thus in the worst case scenario, if all puts go to one server, and all gets are served by a different server, then the replicas will never converge to the same value. To ensure convergence to the same value, key-value stores like Dynamo [30], Apache Cassandra [4], and Riak [2] employ *anti-entropy* protocols. An anti-entropy protocol operates by comparing replicas and reconciling differences. The three main anti-entropy protocols are: (1) Read-Repair (RR), (2) Hinted-Handoff (HH), and (3) Node-Repair (NR). While the first two are *real-time* protocols involved in the read and write paths respectively, the third one is an offline background maintenance protocol, which runs periodically (e.g., during non-peak load hours) to repair out-of-sync nodes (e.g., when a node rejoins after recovering from a crash). In this paper we are only concerned with the real-time anti-entropy protocols. Node-repair is mostly an offline process whose correctness lies solely in the semantics of the merge, so we do not consider it in this paper.

2.3.1 Read-Repair (RR)

Read-repair [30] is a real-time anti-entropy mechanism that ensures that all replicas have (eventually) the most recent version of a value for a given key (see Figure 1). In a typical read path, the coordinator forwards read requests to all replicas, and waits for a consistency level (*CL* out of *N*) number of replicas to reply. If read-repair is enabled, the coordinator checks all the read responses (from the nodes currently alive), determines the most recent read value[2], and finally pushes the latest version to all out of date replicas.

2.3.2 Hinted-Handoff (HH)

Unlike read-repair, hinted-handoff [30] is part of the write path. It offers full write availability in case of failures, and can improve consistency after temporary network failures. When the coordinator finds that one of the replicas responsible for storing an update is temporarily down (e.g., based on failure detector predictions), it stores a hint meta-data for the down node for a configurable duration of time. Once the coordinator detects that the down node is up, it will attempt to send the stored hint to that recovered node. Thus hinted-handoff ensures that no writes are lost, even in the presence of temporary node failures. In other words this mechanism is used to ensure that eventually all writes are propagated to all the replicas responsible for the key.

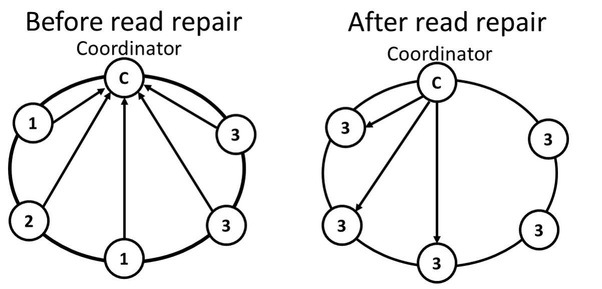

Figure 1. Read-repair propagation : before read-repair, replicas can have inconsistent values (C is the coordinator). After the propagation of the latest value to all out-of-date replicas, all replicas converge on a value.

3. Characterizing and proving eventual consistency

The goal of this paper is to prove eventual consistency of the hinted-handoff and read-repair protocols that systems like Cassandra and Riak implement, delineating precisely the conditions under which they hold. Our effort spanned a period of 15 months, with about 6 person months of effort for modeling and verification. In order to accomplish this task, we *abstract* away from the particular instantiation of these protocols in these systems, and also abstract away from the various options they provide to users to modify the behavior of the system.

For instance, in Riak, using one set of options, every write is tagged with a vector clock at the client, and every replica responsible for it maps it to a *set of values*, one for each last concurrent write that it has received. When a read is issued, Riak can return the set of *all* last concurrently written values to it (these values are called "siblings" in Riak). However, in Cassandra, vector clocks are not used; instead each client labels every write with a timestamp, and despite there being drift amongst the clocks of clients, each replica stores only the last write according to this timestamp. Further, these policies can be changed; for instance in Riak, a user can set options to mimic the Cassandra model.

We will capture these instantiations by generalizing the semantics of how the store is maintained. For the hinted-handoff protocol, we prove eventual consistency under the assumption that the stores are maintained using some *idempotent* operation-based commutative replicated data-type (CRDT) [57, 58] that implements a *register*, while for read-

[2] Determining the most recent version of data to push to out of date replicas is implementation dependent. For Apache Cassandra, the replica value with highest client timestamp wins. Riak uses vector clocks to decide the winner, and can deduce multiple winners in case of concurrent writes.

repair, we prove eventual consistency assuming an arbitrary form of conflict resolution.

Failure Models

Let us first discuss the failure models we consider, which are part of the assumptions needed to prove properties of protocols. We consider two failure modes:

- *Transient failure:* Nodes or network edges can fail, but when they come back, they preserve the state at which they crash and resume from there.
- *Permanent failure:* Nodes or network edges can fail, and when they come back, they have lost main memory and start with some default store.

3.1 Properties of the Hinted-Handoff Protocol

The hinted-handoff protocol is an opportunistic anti-entropy mechanism that happens during writes. When a write is issued, and the asynchronous call to write to certain replicas fail (either explicitly or due to a time-out), the coordinator knows that these replicas could be out of sync, and hence stores these update messages in a hinted-handoff table locally to send them later to the replicas when they come back alive. However, if there is a memory crash (or a permanent failure), the hinted-handoff table would be lost, and all replicas may not receive the messages. In practice, the read-repair (and node-repair) protocols protect against permanent failures.

Commutative Replicated Data Type for Registers: Our main abstraction of the key-value store is to view the underlying protocol as implementing a *register* using an operation-based conflict-free replicated datatype (CRDT)(also called a commutative replicated data-type CmRDT [58, 59]).

We also assume another property of these CmRDTs, namely idempotency— we assume that all messages are tagged with a unique id, and when a message is delivered multiple times, the effect on the store is the same as when exactly one message is delivered. Let us call these idempotent CRDTs[3].

When implementing a simple key-value store, the vector-clock based updates in Riak and the simpler time-stamp based update in Cassandra can in fact be both seen as idempotent CmRDTs, the former being a *multi-valued* (MV) register, and the latter being a *last write wins* (LWW) register (see [57]). (However, since a global wall-clock time is not available, in general, this strategy in Cassandra can *lose* updates [3]). The CmRDTs for both Last Write Wins (LWW) and Multi-valued (MV) registers are in fact idempotent— the systems tags each write with a timestamp, and the conflict-resolution will ignore the future deliveries of a message with same time-stamp (see [57], Section 3.2).

[3] Standard definitions of Operation-based CRDTs do not guarantee idempotency— instead they assume the environment delivers every message precisely once to each replica (see [58], text after Definition 5). Note that state-based CRDTs are defined usually to be idempotent.

The main property we prove about the hinted-handoff protocol is a property called *eventual delivery*, which says that every successful write eventually gets delivered to every replica at least once (under assumptions of kinds of failure, assumptions on replicas being eventually alive, etc.). Hence, instead of eventual consistency, we argue eventual delivery, which in fact is the precise function of these protocols, as they are agnostic of the conflict resolution mechanism that is actually implemented in the system. Furthermore, assuming that each replica actually implements an idempotent operation-based CRDT register, and update procedures for these datatypes are terminating, eventual delivery ensures eventual consistency, and in fact *strong eventual consistency* [58]. Recall that strong eventual consistency guarantees not only eventual consistency, but that the store always contains a value that is a function of the set of updates it has received, independent of the order in which it was received.

Our first result is that a system running only hinted-handoff-based repair provides eventual delivery of updates to all replicas, provided there are only transient faults.

Result#1: *The hinted-handoff protocol ensures eventual delivery of updates to all replicas, provided there are only transient faults. More precisely, if there is any successful write, then assuming that all replicas recover at some point, and reads and write requests stop coming at some point, the write will get eventually propagated to every replica.*

We formally prove the above result (and Result#2 mentioned below) for arbitrary system configurations using program verification techniques on the program model (see Section 4 and Section 5 for details).

The following is an immediate corollary from the properties of eventual delivery and idempotent CRDTs:

Corollary#1: *A system following the hinted-handoff protocol, where each replica runs an operation-based idempotent CRDT mechanism that has terminating updates, is strongly eventually consistent, provided there are only transient faults.*

Aside: The above corollary may lead us to think that we can use any operation-based CmRDT for counters at stores to obtain strong eventually consistent counters in the presence of transient failures. However, CmRDTs for counters are in fact *not idempotent* (and the CmRDT counters in [58] assume that the system will deliver messages precisely once, which hinted handoff cannot guarantee).

3.2 Properties of the Read-repair Protocol

Our second result concerns the read-repair protocol. Read-repair is expected to be resilient to memory-crash failures, but only guarantees eventual consistency on a key provided future reads are issued at all to the key. Again, we abstract away from the conflict resolution mechanism, and we assume that the coordinator, when doing a read and getting

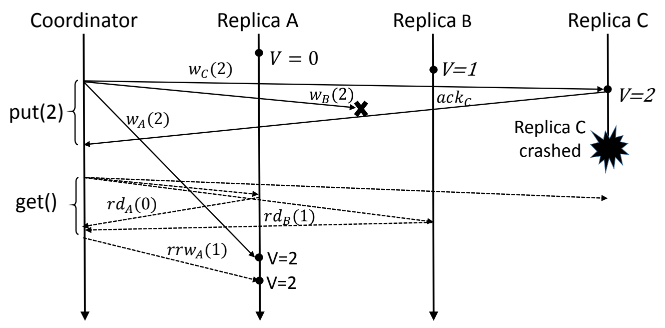

Figure 2. The time-line showing that a single read-repair operation does not guarantee convergence of the live replicas. In the figure w_r are write messages to replica r, rd_r are messages from replica r to the coordinator on the read path, and rrw_r is the read-repair message to replica r. Time in the figure advances from top to bottom. The messages along the read(-repair) path are shown as dotted lines and along the write path as solid lines.

different replies from replicas, propagates *some* consistent value back to all the replicas. This also allows our result to accommodate anti-entropy mechanisms [31] that are used instead of read-repair, in a reactive manner after a read. Note that this result holds irrespective of the hinted-handoff protocol being enabled or disabled.

It is commonly believed that when a read happens, the read repair will repair the live nodes at the time of the read (assuming they stay alive), bringing them to a common state. We modeled the read-repair protocol and tried to prove this property, but we failed to come up with appropriate invariants that would ensure this property. This led us to the hypothesis that the property may not be true.

To see why, consider the time-line in Figure 2. In this scenario, the client issues a *put* request with the value 2, which is routed by the coordinator to all three replicas– A, B, and C (via messages $w_A(2), w_B(2)$, and $w_C(2)$). The replica C successfully updates its local store with this value. Consider the case when the write consistency is one and the put operation succeeds (inspite of the message $w_B(2)$ being lost and the message $w_A(2)$ being delayed). Now assume that the replica C crashes, and the last write (with value 2) is in *none* of the alive replicas– A and B. If we consider the case where B has the latest write (with value 1) amongst these two live nodes, a subsequent read-repair would write the value 1 read from B to $A's$ store (via message $rrw_A(1)$ in Figure 2). But before this write reaches A, A could get a pending message from the network ($w_A(2)$) and update its value to a more recent value– 2. In this situation, after replica A has updated its value to 2, the two alive replicas (A and B) do not have consistent values. Due to the lack of hints or processes with hints having crashed B may never receive the later write (message $w_B(2)$).
We therefore prove a more involved property of read-repair:

Result#2: *After any sequence of reads and writes, if all operations stop except for an infinite sequence of reads of a key, then assuming the set R of replicas are alive at the time of the first such read and thereafter, the replicas in R will eventually converge to the same value.*

We prove the above result also using program verification on the program model. Intuitively, as long as an indefinite number of reads to the key happen, the system will ensure that the subset of live replicas responsible for the key converge to the same value, eventually. A read-repair may not bring the live replicas to sync if there are some pending messages in the system. However, since there is only a finite amount of *lag* in the system (pending messages, pending hints, etc.), and once the system is given enough time to finish its pending work, a read-repair will succeed in synching these replicas.

3.3 Read-repair and CRDT

It is tempting to think that one could implement any CRDT and reach eventual consistency of the CRDT store using solely read-repair, similar to the Corollary we obtained for Result#1. However, this is tricky when clients send operations to do on the CRDT and the conflict-resolution in read-repair happens using state-based merges.

For instance, assume that we implement a counter CRDT, where state-merges take the maximum of the counters, and operations increment the counter [58]. Then we could have the following scenario: there are 7 increments given by clients, and the counter at replica A has the value 5 and replica B has 7 (with two increments yet to reach A), and where a read-repair merges the values at these replicas to 7, after which the two pending increments arrive at A incrementing it to 9 (followed by another read-repair where B also gets updated to 9). Note that consistency is achieved (respecting our Result#2), but the counter stores the wrong value.

Systems such as Riak implement CRDTs [6] using these underlying protocols by *not* propagating operations (like increments) across replicas, but rather increment one replica, and pass the *state* to other replicas, and hence implement a purely state-based CRDT [5].

4. Program Models for Protocols

In this section we describe how we model the anti-entropy protocols used in eventually consistent key-value stores. The architecture of our model is depicted in Figure 3.

Our model consists of several methods (*get*, *put*, *write_ls*, *read_ls*, etc.) for each replica, that run concurrently as threads, make asynchronous calls to each other, and keep their state through shared data-structures (*local_store*, *hint_store*, etc.). Furthermore, in order to model asynchronous calls, we maintain a data-structure *pending_store*, that models messages in the network that haven't yet been delivered.
The methods in our model include:

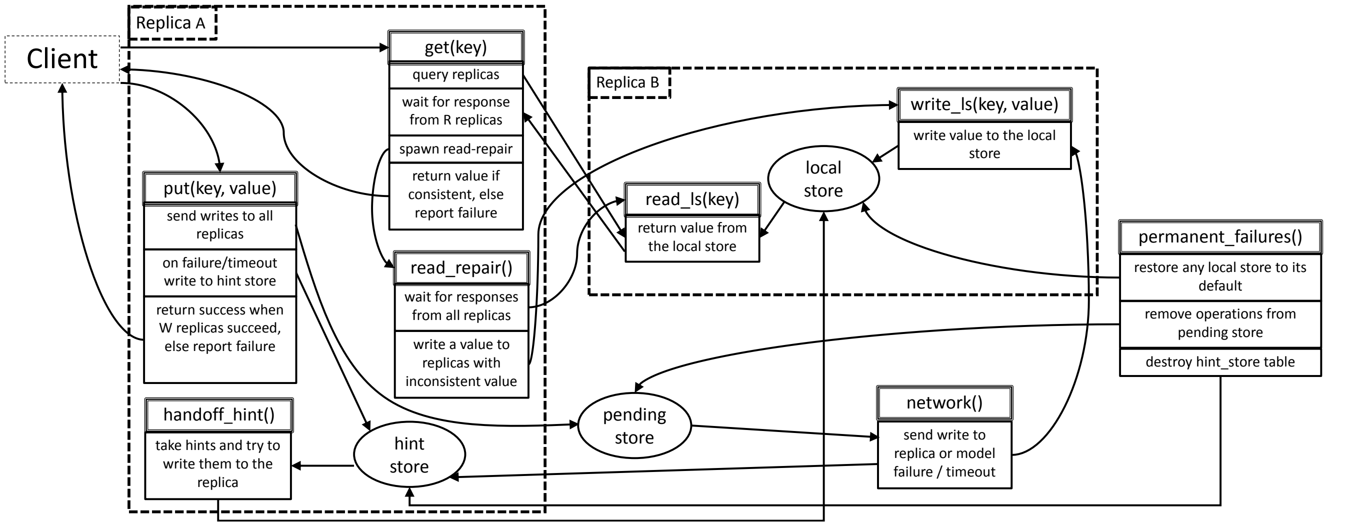
Replica A
Client
get(key)
query replicas
wait for response from R replicas
spawn read-repair
return value if consistent, else report failure
put(key, value)
send writes to all replicas
on failure/timeout write to hint store
return success when W replicas succeed, else report failure
read_repair()
wait for responses from all replicas
write a value to replicas with inconsistent value
handoff_hint()
take hints and try to write them to the replica
hint store
Replica B
write_ls(key, value)
write value to the local store
local store
read_ls(key)
return value from the local store
permanent_failures()
restore any local store to its default
remove operations from pending store
destroy hint_store table
pending store
network()
send write to replica or model failure / timeout

Figure 3. Architecture of the model: boxes indicate methods, ellipses show data structures, arrows show communications.

- The *get* and *put* methods at coordinators that forms the interface to clients for reading and writing key-values.
- An internal method *handoff_hint* for each replica that runs all the time and removes hints from the hinted-handoff table and propagates them to the appropriate replicas (provided they are alive).
- An internal method *read_repair* which is part of the read path, waits for all the replicas to reply, and on detecting replicas with inconsistent values writes the consistent value to those replicas.
- Internal methods *read_ls* and *write_ls*, that read from and write to the local stores (provided they are alive).
- An internal method *network* that runs all the time and delivers messages in the pending store to replicas.
- An internal method *permanent_failures*, which when permanent failure is modeled, runs all the time, and can remove elements from the pending set (modeling loss of messages), restore any local store to its default value (modeling store crashes), and destroy hinted-handoff tables.

We use the following data-structures:

- An array *LocalStore[]* that stores the local store for each replica and each key that the replica maintains.
- An array *HintStore[]* that stores, for each replica, the set of hints stored at the replica.
- An array *PendingStore[]* that stores a set of pending messages on the network between replicas.

Note that the modeling of these methods using fine-grained concurrency ensures arbitrary interleaving of these processes as well as arbitrary delays in them. Also, transient failures, where nodes fail but resume later with the correct state, can be seen as delays in processes, and hence are captured in this concurrency model. The thread that delivers messages in the pending set models arbitrary delays in the network.

The *read_ls* and *write_ls* methods are modeled abstractly as idempotent CRDTs by defining them as stubs which maintain properties. When testing, these methods need to be instantiated to particular conflict-resolution strategies (like *MV* and *LWW*).

Modeling the get operation: When a client issues a *get* request for a key, in our model the request is routed to the coordinator that is determined for this key according to an abstract map (our verification hence works for all possible hashing schemes). Every key-value datum is replicated across multiple nodes, where the number of nodes that contain the key-value datum is determined by a replication factor. The coordinator maintains a preference list of replicas that contain data values for keys that are mapped to it. Along the read path, the coordinator asynchronously issues the read request to all replica threads (an asynchronous call to a replica is depicted in Figure 3 as an arrow from the *get* method to *read_ls*). As shown in Figure 3, the coordinator blocks for a non-deterministic amount of time or until it receives enough responses (the arrow directed from *read_ls* to *get*) as specified by the read consistency level *R*. On receiving responses from *R* replicas, it returns the read value(s) to the client. If read repair is enabled, the coordinator also spawns a background thread (depicted as a call to *read_repair* from *get* in Figure 3) which will wait for responses from the other replicas (it already knows about responses from the *R* replicas) for a non-deterministic amount of time. This thread determines the most recent data value of all the values stored in various replicas, and writes it to the replicas with stale values.

Modeling the put operation: When a client issues a *put* request to store a key-value pair, the request is routed to the appropriate coordinator, as explained before. The coordinator asynchronously issues write requests to all replica threads in its preference list. The coordinator then blocks for a non-deterministic amount of time or until it receives enough responses, as specified by the write consistency level W. To model arbitrary network delays or failures of the replicas, the write operations to these replicas are inserted by the coordinator into the pending store data structure (in Figure 3 this is depicted as an arrow from *put* to the *pending_store*). If the coordinator receives responses from W replicas, it informs the client about the successful *put* operation.

Modeling the network: A background *network* thread models arbitrary network delays or failure scenarios as it removes a write operation from the pending store data structure and, non-deterministically, either updates the local store of the appropriate replica with the write or simply loses the operation. When the hinted-handoff protocol is enabled and read-repair is disabled, we assume that the write operations are not lost. In this scenario, when losing/removing the write operation from the pending store, the *network* thread inserts the operation as a hint in the hinted-handoff table of the appropriate coordinator. The *permanent_failures* thread does not execute in this case and data in the global data structures is not lost.

Testing Program Models: Once we devised a model of the anti-entropy protocols, we tested it to make sure that it corresponds to actual systems. In our testing model we provide implementations for the stubs that model failure and non-determinism in message arrivals. In particular, for testing we use random coin-tosses instead of non-deterministic choices present in the verification model. Besides this, we also provide concrete implementations for conflict-resolution strategies for operations on CRDTs based on the last write wins (LWW) and vector clocks (MV).

We wrote a test harness that arbitrarily issues put and get operations for various key-value pairs. We then checked if the results of these operations can be realized by the actual eventually consistent key-value stores. We also used CHESS [53], which is a systematic testing tool for concurrent programs, to systematically enumerate all possible thread schedules. Using CHESS we were able to ensure that our model realized strange but possible behaviors of the eventually-consistent stores.

We exhaustively tested a number of possible scenarios. Here, we discuss a configuration with three replicas, where the write consistency level is set to two, and the read consistency level is set to one. One interesting scenario is where the client successfully performs a write operation on a key with a value 0, followed by an unsuccessful write on the same key with a value 1. A subsequent read of the key returns the value 1. This is a nonintuitive scenario, but it can manifest in a real system because failures are not guaranteed to leave the stores unaffected and an unsuccessful write can still write to some of the replicas.

In another scenario, the client successfully performs two consecutive write operations to a key with values 0 and 1. Subsequently, one read returns the value 1, while a subsequent read returns the stale value 0. This behavior can happen in a real system where the client gets staler values over time. In particular, this scenario occurs when the two replicas store the value 1 after the second write operation (remember the write consistency level is two) and the third replica still stores the stale value 0.

Now consider a configuration with three replicas but where both read and write consistency levels are set to one. In the third scenario we consider the case when the client issues three consecutive successful writes with values 0, 1, and 2. Subsequent reads first return a stale value 1 followed by returning an even more stale value 0. This unusual behavior manifests when the three replicas all store different values (the read requests may access any of them).

Finally, we consider a scenario where there are four consecutive successful writes to a key with values 0, 1, 2, and 3. If the subsequent two reads for the same key return values 2 followed by 1, then a following third read cannot return the value 0. This scenario cannot happen because the three replicas must have values 1, 2, and 3 at the time of the last read (the reader is invited to work this case out on paper).

We used CHESS to confirm the realizability of the first three scenarios, and infeasibility of the last scenario. CHESS took from less than a second to up to 10 minutes to exhaustively explore all interleavings corresponding to these four test harnesses. We were also able to observe some of these scenarios in a real installation of Cassandra.

5. Verification of Anti-entropy protocols

In this section we first describe our verification methodology, followed by our verification of the hinted-handoff and read-repair anti-entropy protocols.

5.1 Verification Methodology

Verification process: We use the *deductive verification* style for proving programs correct. For sequential programs, this style is close to Hoare logic style reasoning [15, 37]. It proceeds by the programmer annotating each method with pre/post conditions and annotating loops with loop invariants in order to prove assertions in the program. Furthermore, in order to prove that functions terminate, the user provides *ranking functions* for loops (and recursive calls) that are mappings from states to natural numbers that must strictly decrease with each iteration [15, 63]. Reasoning that these annotations are correct is done *mostly* automatically using calls to constraint solvers (SMT solvers), with very little help from the user.

There are several different approaches to verify concurrent programs, especially for modular verification. We use

VCC [29] tool to verify our models. VCC is a verifier for concurrent C programs [4]. The basic approach we take to verify our models is to treat each concurrent thread as a sequential thread for verification purposes, but where every access to a shared variable is preceded and succeeded by a *havoc* that entirely destroys the structures shared with other threads. However, this havoc-ing is guarded by an *invariant* for the global structures that the user provides. Furthermore, we check that whenever a thread changes a global structure, it maintains this global invariant. This approach to verification is similar to *rely-guarantee* reasoning [38], where all threads rely and guarantee to maintain the global invariant on the shared structures.

Specifications: Another key aspect of the verification process is writing the specification. Though the specification is written mainly as assertions and demanding that certain functions terminate, specifications are often described accurately and naturally using *ghost code* [15, 29]. Ghost code is code written purely for verification purposes (it does not get executed) and is written as instructions that manipulate ghost variables. It is syntactically constrained so that real code can never see the ghost state. Hence this ensures that the ghost code cannot affect the real code.

In our framework, we use ghost code to model the taint-based specification for eventual delivery (see Section 5.2). It is important that the protocol does not see the tainted write, because we do not want a flow of information between the executable program and the specification. We also use ghost code to maintain mathematical abstractions of concrete data-structures (like the set associated with an array, etc.).

Testing annotations: We extensively used testing, especially in early stages, to assert invariants that we believe held in the system at various points in the code. Prior to verification, which requires strong inductive invariants, testing allowed us to gain confidence in the proof we were building (as well as the model we were constructing). These invariants then were the foundation on which the final proof was built upon.

5.2 Verifying the hinted-handoff protocol

As explained in Section 3, verification that hinted-handoff protocol maintains strong eventual consistency under transient failures and for idempotent operation-based CRDT reduces to verification of eventual delivery (Result#1 in Section 3.1). Recall that, eventual delivery is the property that every successful write eventually gets delivered to every replica at least once.

Taint-based specification of eventual delivery: We model eventual delivery using a ghost field *taint*, that records a particular (exactly one) write operation issued to the coordinator. For a sequence of reads (r) and writes (w) operations, a write for an arbitrary key is designated as tainted:

$$\textit{history}\ \xrightarrow{\cdots,\mathbf{r},\mathbf{r},\mathbf{w},\mathbf{w},\mathbf{r},\mathbf{r},\mathbf{r},\mathbf{w},\mathbf{r},\mathbf{r},\mathbf{w},\mathbf{r},\mathbf{w}^{\mathbf{taint}}\cdots}$$

We now assert the specification that this taint will eventually propagate to each replica's local store. Intuitively, the write that was chosen to be tainted will taint the value written, and this taint will persist as the value moves across the network, including when it is stored in the hint store and the pending store, before being written to the local store. Taints are persistent and will not disappear once they reach the local store. Hence demanding that the local stores eventually get tainted captures the property that the chosen write is eventually delivered at least once to every local store.

Note that the tainted values are ghost fields which the system is agnostic to, and hence proving the above property for an arbitrary write in fact ensures that *all writes* are eventually delivered.

Proving the taint-based specification: To prove the specification we introduce several ghost fields:

(a) *ls_tainted_nodes*, the set of replicas that have updated their local store with the tainted write,

(b) *hs_tainted_nodes*, the set of replicas for which the coordinator has stored the tainted write operation as a hint in its hint store, and

(c) *ps_tainted_nodes*, the set of replicas for which the tainted write has been issued, but its delivery is pending on the network.

We add ghost-code to maintain the semantics of the taint in various methods, including *put*, *network* and *handoff_hint*. Every time any of these methods transfers values, we ensure that the taints also get propagated. At local stores, when a value is written, the value at the local store is tainted if it either already had a tainted value or the new value being written is tainted; otherwise, it remains untainted. (In fact, the taint-based store can itself be seen as an operation based CRDT which never loses the taints.) Furthermore, the ghost fields described above, which are set abstractions of the related stores, are also kept up to date using ghost updates.

For eventual delivery we want to prove that, when all replicas remain available and all the read/write operations have stopped, regardless of how all concurrent operations are scheduled, the tainted write operation is indeed eventually propagated to the local stores of all the replicas.

We model eventual taintedness of stores as a *termination property* by modeling a schedule harness that takes over the scheduler, and arbitrarily schedules *network* and *handoff_hint* threads until the taint has propagated to all replicas. The termination of this harness then proves eventual taintedness, which in turn proves eventual delivery. In the schedule harness, the *permanent_failures* thread is not scheduled since we assume only transient failures can occur.

[4] Even though our model and invariants apply to unbounded number of instances, verification of C programs, strictly speaking, assumes integer manipulations to `MAX_INT` (i.e., typically 2^{32} on 32-bit architectures).

```
_(invariant (\forall int j;
  (j >= 0 && j < PREFLIST_SIZE) ==>
  (ls->tainted_nodes[pl->pref_list[coord+j]]
|| hs->tainted_nodes[pl->pref_list[coord+j]]
|| ps->tainted_nodes[pl->pref_list[coord+j]])))

_(decreases hs->size + 2 * ps->size)
```

Figure 4. The invariant and the decreases clause for the scheduler in the hinted-handoff protocol

In order to prove termination of the harness, we specify a (safety) invariant for the scheduler and specify a ranking function for arguing the termination of this method. The invariant for the scheduler loop states that for every replica responsible for the tainted key, either its local store is tainted or there is a tainted write pending in the network for it, or there is a hint in the corresponding coordinator which has a tainted write for it. More precisely, for each replica responsible for the tainted key, we demand that the replica is present in one of the ghost-sets, namely, *ps_tainted_nodes*, *hs_tainted_nodes*, and *ls_tainted_nodes*:

$$\forall r.(\quad r \in ps\_tainted\_nodes \quad \vee \quad r \in hs\_tainted\_nodes \quad \vee \quad r \in ls\_tainted\_nodes)$$

where the quantification is over replicas r responsible for the tainted key. In VCC, this invariant is written as shown in Figure 4.

The ranking function for the scheduler is a function that quantifies, approximately, the *time* it would take for the system to reach a consistent state. In our case, the ranking function $|hint\_store| + 2 \cdot |pending\_store|$ suffices. Note that we prove that the rank decreases with the scheduling of any thread, thereby guaranteeing termination. In VCC, Figure 4 shows the *decreases* clause that represents this ranking function. Since the *network* thread can remove a write from the pending store and can insert it into the *hint_store*, the factor 2 in the ranking function is necessary.

5.3 Verifying the read-repair protocol

As explained in Section 3, we want to verify that the read-repair protocol maintains eventual consistency in the presence of permanent failures (as stated in Result#2 in Section 3.2). We prove this result both when hinted-handoff is turned on as well as when it is disabled (we capture whether hinted-handoff is enabled/disabled using a macro directive, and prove both versions correct). For simplicity of presentation we only explain here the case when the hinted handoff protocol is disabled.

Recall that permanent failures could: (a) modify the local store by setting them to default values, (b) remove an operation from the pending store, and (c) destroy the hint store.

For eventual consistency we want to prove that when all the write operations have successfully returned to the client, then after only a finite number of read operations on a key, the read-repair mechanism ensures that the set of R available replicas will converge. Note that we want to prove this property when the replicas in R remain available throughout these read operations, but regardless of how all the other operations are scheduled concurrently with read-repair.

When the writes stop and only the read of a particular key occurs (infinitely often), we write a schedule harness that takes over the scheduler at this point, in order to argue that consistency is eventually reached (similar to the verification of the hinted-handoff protocol).

The schedule harness arbitrarily schedules the reads and the repairs, the *network* threads and *permanent_failures*, but restricts it to not modify local stores of replicas in R (since replicas in R cannot fail any longer). The harness has an outer loop that continually issues reads of the key and executes the read-path and the read-repair with interference from other threads modeling the system (*network* and *permanent_failures*). This loop terminates only when convergence is reached, and hence our task is to prove that the loop terminates.

We verify the harness again by specifying safety invariants and a ranking function. The ranking function for the scheduler is that the size of the pending store, $|\, pending\_store\,|$, decreases with every loop.

Intuitively, an unbounded number of read-repairs get executed, and if the network thread does not interfere during the read-repair, then the replicas will reach a consistent state. However, if the network does interfere (delivering pending writes to replicas), then read-repair may not succeed in syncing the replicas in this round, but the size of the pending set must necessarily decrease.

5.4 Verification Statistics

We performed the verification on an Intel CORE-i7 laptop with 8 GB of RAM, running Windows 8 and using Visual Studio 2012 with VCC v2.3 as a plugin. Our verification model consists of about 1500 lines of code and annotations, where about 900 lines are executable C code and the rest are annotations (not seen by the C compiler). The annotations comprise ghost code (20%) and invariants (80%) [5]. The total time taken for the verification of the whole model is around a minute. Since the verification is modular, we focus on verifying one function at a time while modeling the protocol. Verification of each function takes around 5 seconds. The verification is hence highly interactive: we add small chunks of executable code and annotations, run the verifier, refine the code or annotations, re-run the verification, and iterate.

6. Discussion: Experience and Lessons

We now describe our experience and lessons learned while modeling and verifying the protocols.

Lesson 1: Building models is an iterative process of refinement, and it takes time. We did not build the model in one

[5] The web-site for the project and our code is here: `http://web.engr.illinois.edu/~pek1/cpm/`

day. Our effort spanned a period of 15 months, with about 6 person months of effort for modeling and verification. Initially, we built a coarse-grained model that eschewed parallelism of reads/writes in favor of building complete and correct models for the store mechanisms and for failures. Even with this initial coarse-grained model, we came up with the taint-based specification and realized that this specification was actually capturing not eventual consistency but rather eventual delivery (Section 3.1).

We then retrofitted the model with concurrency, which led to a much more complex and lengthy proof that took into account interactions of threads. The concurrency retrofitting required the most effort, as it involved learning the intricacies of our verification platform, VCC, and its concurrency verification model. In fact, when retrofitting concurrency, we found errors in our read-repair specification (see below and see Section 3.2).

Lesson 2: The quest to prove results lead to surprising outcomes after building, writing, and verifying with models. The result that a single read-repair does ensure eventual consistency of the stable live nodes was a result we wrongly believed in before the verification. In fact, in the coarser-grained concurrency model we built for it, we were able to prove the result, but this proof fell through when we retrofitted it with fine-grained concurrency. The inability to establish a global ranking function that ensured consistency was reached in finite time, –this fact led us to disbelieving it and disproving it. This led us to believe that an unbounded number of reads would give eventual consistency, *provided the scheduler fairly scheduled the system*. However, when we proved the property, we realized that a fair scheduler isn't necessary, which was another surprise in itself(Result#2 in Section 3.2).

Lesson 3: Non-obvious and unknown results can arise out of building, writing, and verifying with models. The taint-based modeling of eventual consistency led us to realize that the hinted-handoff protocol actually was ensuring eventual delivery, and would hence work for any CRDT register. Proving this property led us to the transient failure model that is needed for this protocol to ensure eventual delivery. The result that, under transient failures, hinted-handoff ensures strong eventual consistency of *any* idempotent CRDT (Corollary#1, in Section 3.1) was a result that we did not know before, and resulted directly from the abstract proof.

We also realized that CRDTs for *counters* in the literature are not idempotent [58]— these CRDTs assume that messages are delivered *precisely once* (as opposed to at least once), and we realized that systems like Riak and Cassandra do not assure delivery precisely once, even when only transient failures are present. This explained to us the predominance of purely state-based implementation of CRDTs in systems such as Riak [5].

Lesson 4: Systems developers need to be building and writing models either concurrent with or prior to, building the actual system. As a direct consequence of the above Lessons 2 and 3, we conclude that building models can aid systems developers understand the targeted properties of the system being built, with great clarity. The verification experience helped us to understand protocols much better than we had previously.

The building of our model and testing this model was useful in both building a faithful model as well as in understanding it, in expressing assumptions on the failure model, and figuring out the correct specifications. The model gave us a concrete vocabulary (especially data-structures like the pending set, which captures the messages in the network, etc.) to reason more formally even in discussions (similar to the way a formal modeling on paper gives such a vocabulary). We believe building models can inform the vocabulary that developers use in talking about their system and its internals at the development stage, and holds the potential to minimize bugs arising out of miscommunication.

There is corroborating evidence from Amazon researchers [54, 55] that building such models and verifying them gives additional benefits of understanding protocols, making design decisions, and adding features to protocols. Note that Amazon used TLA$^+$ to model protocols (in [54], they discuss why they chose TLA, and find VCC also met most of their requirements). However, in their work, they did not verify the protocols, but only model-checked them (i.e., systematically tested them) for all interleavings for small instantiations of the system.

Lesson 5: Modeling in a high-level language closer to implementation (like C) is programmer-friendly, testable, and supports verification. We believe that modeling in C offers many advantages (in particular, in comparison with languages such as TLA). First, systems engineers understand C, and the modeling entails building a *simulator* of the system, which is something engineers do commonly for distributed systems anyway. Being executable, the model can be subject to standard testing, where engineers can write test harnesses, tweaking the environment's behavior and fault models, simply by writing code. Third, for systematic testing (model-checking), we have powerful tools such as CHESS that can systematically explore the behavior of the system, exploring non-determinism that arises from interleavings. Fourth and finally, the ability to prove programs using pre/post conditions and invariants, using VCC, gives a *full fledged verification platform* to prove the entire protocol correct, where most reasoning is pushed down to automated logic-solvers.

We advocate certified program models as a sweet-spot for modeling, testing, and verification. It abstracts from the real system, but in doing so captures many instantiations and versions of these systems. And yet is written in code, allowing for easier model-building and testing. Finally, it affords full fledged verification using mostly-automated verification platforms.

7. Related Work

The CAP theorem [21, 49] indicates that a distributed system that can tolerate partition failures can either provide strong data consistency (eg., linearizability, sequential consistency) or high availability. Strong consistency guarantees like linearizability can be provided using a single commit point ensured by two-phase/three-phase commit protocols [61] or by distributed consensus (eg., Paxos [45]). As opposed to strong consistency, most existing distributed systems provide only weaker eventual consistency guarantees, which roughly mean that when the updates stop the entire system eventually converges to the same value. There are also known some other models of consistency such as consistent prefix, bounded staleness, monotonic reads, and read-my-writes [12, 22, 62].

Amazon's use of formal techniques [54, 55] for increasing confidence in the correctness of their production systems is in the same spirit as this work. The engineers at Amazon have successfully used TLA+ [44] to formally specify the design of various components of distributed systems. The formal TLA+ specifications are executable like our program models and these design specifications have been systematically explored using a model checker to uncover several subtle bugs in these systems. However, instead of modeling distributed algorithms in TLA+, we model them as C programs. Newcombe et al. [54, 55] acknowledge that modeling systems in a high-level language like C increases the productivity of the engineers. More importantly, in addition to checking the models using model checkers up to a certain trace length, a model written in C lends itself to mostly automated verification using tools like VCC that utilize automated constraint solvers, and that can verify unbounded instances of the system.

Previous attempts at formal modeling of distributed systems for design and verification include the Farsite project [18] and the modeling of Pastry protocol for distributed hash-tables [51] using TLA; the modeling of the Chord ring maintenance protocol using Alloy [67]; the verification of consensus algorithms using Isabelle [27]; and the verification of event-driven device drivers written in the P language [32, 33]. There have also been efforts towards formally modeling key-value stores like Cassandra using Maude [48]. In this work, consistency properties are expressed in Maude using linear temporal logic (LTL) formulae. This model checking approach either is not exhaustive or is exhaustive on bounded instances while ours is exhaustive on unbounded instances.

In "Paxos Made Live" [25], the authors show that directly building a system that implements an algorithm that is as well-studied as the Paxos consensus algorithm is a non-trivial task. In our experience, also as in Amazon's [54, 55], modeling algorithms as programs is a good intermediate step that allows one to quickly prototype the algorithm, understand issues concerning the implementation, explore the design space, and also test and verify the design.

Recently, there has been work on programming languages that ease development of distributed systems, in particular, with respect to consistency properties at the application level [14, 23, 60] and failfree idempotence [56]. Kuru et al. [42] verify properties of transactional programs running under a relaxed scheme that provides snapshot isolation.

In a recent work [65], the authors have used Coq to implement distributed algorithms that are verified. In [19, 22, 24], the authors explore logical mechanisms for specifying and verifying properties over replicated data types. Deductive verification using automatic tools, such as VCC [29] and Dafny [47] has been extensively used for verifying systems in domains other than distributed systems. Some of the examples are: verifying a hypervisor for the isolation property [46], verifying operating systems like Verve [66] and ExpressOS [50] for security, verifying the L4 microkernel for functional correctness [40, 41] and verifying high-level applications for end-to-end security [35].

8. Conclusions

In this paper we have shown how the philosophy of certified program models can be used to verify and find fault-tolerance violations in distributed systems, with a specific focus on key-value/NoSQL storage systems. We have verified both eventual delivery of the hinted-handoff protocol under transient failures (which ensures strong eventual consistency for any store maintained as a CmRDT register) as well as eventual consistency of the read-repair protocol when arbitrary number of reads are issued. We also discovered several surprising counter-examples during the verification for related conjectures, and the experience helped us develop a firm understanding of when and how these protocols guarantee eventual consistency.

Based on our experience, we believe the notion of certified program models is applicable to a broad swathe of distributed systems properties beyond hinted-handoff and read repair. For instance, while we have assumed a CRDT abstraction, some artifacts of the way they are implemented in systems deserve verification— for instance, Riak's use of vector clocks (now called Causal Context) and the associated conflict resolution and pruning mechanisms are worth verifying for correctness, even in the case where there are no failures. The verification of how counters (and in general other CRDTs) work in today's distributed systems in tandem with eventual consistency protocols, and the idempotence guarantees on them, is another worthy target. Beyond distributed systems, properties of file systems are also a good match for certified program models, e.g., works like [13, 28]. In these systems the ordering and consistency of file system operations are decoupled, and verifying that the desired ordering and consistency properties hold, under failures or otherwise, is an interesting future direction.

References

[1] Amazon: milliseconds means money. `http://goo.gl/fs9pZb`.

[2] Basho Riak. `http://basho.com/riak/`.

[3] Call me maybe: Cassandra. `https://aphyr.com/posts/294-call-me-maybe-cassandra/`.

[4] Cassandra. `http://cassandra.apache.org/`.

[5] Data Structures in Riak. `https://vimeo.com/52414903`.

[6] Data Types. `http://docs.basho.com/riak/latest/theory/concepts/crdts/`.

[7] HBase. `http://hbase.apache.org/`.

[8] Market research media, NoSQL market forecast 2013-2018. `http://www.marketresearchmedia.com/?p=568`.

[9] NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex. `http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/`.

[10] Project Voldemort. `http://goo.gl/9uhLoU`.

[11] ABADI, D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer 45*, 2 (2012), 37–42.

[12] ABADI, D. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer 45*, 2 (Feb. 2012), 37–42.

[13] ALAGAPPAN, R., CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Beyond storage apis: Provable semantics for storage stacks. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.

[14] ALVARO, P., CONWAY, N., HELLERSTEIN, J., AND MARCZAK, W. R. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings* (2011), pp. 249–260.

[15] APT, K. R. Ten years of hoare's logic: A survey&mdash;part i. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct. 1981), 431–483.

[16] BAILIS, P., AND GHODSI, A. Eventual consistency today: Limitations, extensions, and beyond. *Queue 11*, 3 (Mar. 2013), 20:20–20:32.

[17] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[18] BOLOSKY, W. J., DOUCEUR, J. R., AND HOWELL, J. The farsite project: A retrospective. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 17–26.

[19] BOUAJJANI, A., ENEA, C., AND HAMZA, J. Verifying eventual consistency of optimistic replication systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2014), POPL '14, ACM, pp. 285–296.

[20] BREWER, E. A certain freedom: Thoughts on the CAP theorem. In *Proc. ACM PODC* (2010), pp. 335–335.

[21] BREWER, E. A. Towards robust distributed systems (Invited Talk). In *Proc. ACM PODC* (2000).

[22] BURCKHARDT, S. Principles of eventual consistency. *Foundations and Trends in Programming Languages 1*, 1-2 (2014), 1–150.

[23] BURCKHARDT, S., FÄHNDRICH, M., LEIJEN, D., AND WOOD, B. P. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2012), ECOOP'12, Springer-Verlag, pp. 283–307.

[24] BURCKHARDT, S., GOTSMAN, A., YANG, H., AND ZAWIRSKI, M. Replicated data types: Specification, verification, optimality. *SIGPLAN Not. 49*, 1 (Jan. 2014), 271–284.

[25] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[26] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008).

[27] CHARRON-BOST, B., AND MERZ, S. Formal verification of a Consensus algorithm in the Heard-Of model. *Intl. J. Software and Informatics 3*, 2-3 (2009), 273–304.

[28] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 9.

[29] COHEN, E., DAHLWEID, M., HILLEBRAND, M., LEINENBACH, D., MOSKAL, M., SANTEN, T., SCHULTE, W., AND TOBIES, S. VCC: A practical system for verifying concurrent c. In *TPHOLs '09* (2009), pp. 23–42.

[30] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP '07* (2007), pp. 205–220.

[31] DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1987), PODC '87, ACM, pp. 1–12.

[32] DESAI, A., GARG, P., AND MADHUSUDAN, P. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 709–725.

[33] DESAI, A., GUPTA, V., JACKSON, E., QADEER, S., RAJAMANI, S., AND ZUFFEREY, D. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 321–332.

[34] GILBERT, S., AND LYNCH, N. A. Perspectives on the CAP theorem. *IEEE Computer 45*, 2 (2012), 30–36.

[35] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* (2014), J. Flinn and H. Levy, Eds., USENIX Association, pp. 165–181.

[36] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[37] HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

[38] JONES, C. B. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 5*, 4 (1983), 596–619.

[39] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[40] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., HEISER, G., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: formal verification of an operating-system kernel. *Commun. ACM 53*, 6 (2010), 107–115.

[41] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 207–220.

[42] KURU, I., OZKAN, B. K., MUTLUERGIL, S. O., TASIRAN, S., ELMAS, T., AND COHEN, E. Verifying programs under snapshot isolation and similar relaxed consistency models. In *9th ACM SIGPLAN Workshop on Transactional Computing* (2014).

[43] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS OSR 44*, 2 (2010), 35–40.

[44] LAMPORT, L. The TLA Home Page. `http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html`.

[45] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[46] LEINENBACH, D., AND SANTEN, T. Verifying the microsoft hyper-v hypervisor with vcc. In *Proceedings of the 2Nd World Congress on Formal Methods* (Berlin, Heidelberg, 2009), FM '09, Springer-Verlag, pp. 806–809.

[47] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers* (2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer, pp. 348–370.

[48] LIU, S., RAHMAN, M., SKEIRIK, S., GUPTA, I., AND MESEGUER, J. Formal modeling and analysis of cassandra in maude. In *Formal Methods and Software Engineering*, S. Merz and J. Pang, Eds., vol. 8829 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 332–347.

[49] LYNCH, N., AND GILBERT, S. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2 (2002), 51–59.

[50] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 293–304.

[51] MERZ, S., LU, T., AND WEIDENBACH, C. Towards Verification of the Pastry Protocol using TLA+. In *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems* (Reykjavik, Iceland, June 2011), R. Bruni and J. Dingel, Eds., vol. 6722 of *FMOODS/FORTE 2011*.

[52] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), PLDI '07, pp. 446–455.

[53] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08, pp. 267–280.

[54] NEWCOMBE, C. Why amazon chose TLA +. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings* (2014), pp. 25–39.

[55] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How amazon web services uses formal methods. *Commun. ACM 58*, 4 (Mar. 2015), 66–73.

[56] RAMALINGAM, G., AND VASWANI, K. Fault tolerance via idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2013), POPL '13, ACM, pp. 249–262.

[57] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of Convergent and Com-

mutative Replicated Data Types. Research Report RR-7506, Jan. 2011.

[58] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (2011), SSS'11, pp. 386–400.

[59] SHAPIRO, M., PREGUIÇA, N. M., BAQUERO, C., AND ZAWIRSKI, M. Convergent and commutative replicated data types. *Bulletin of the EATCS 104* (2011), 67–88.

[60] SIVARAMAKRISHNAN, K., KAKI, G., AND JAGANNATHAN, S. Declarative programming over eventually consistent data stores. In *PLDI 2015, Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation* (Portland, OR, USA, June 15–17, 2015).

[61] SKEEN, D., AND STONEBRAKER, M. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng. 9*, 3 (May 1983), 219–228.

[62] TERRY, D. Replicated data consistency explained through baseball. *Commun. ACM 56*, 12 (Dec. 2013), 82–89.

[63] TURING, A. The early british computer conferences. MIT Press, Cambridge, MA, USA, 1989, ch. Checking a Large Routine, pp. 70–72.

[64] VOGELS, W. Eventually consistent. *ACM CACM* (2009), 40–44.

[65] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed system. In *PLDI 2015, Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation* (Portland, OR, USA, June 15–17, 2015).

[66] YANG, J., AND HAWBLITZEL, C. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM 54*, 12 (2011), 123–131.

[67] ZAVE, P. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev. 42*, 2 (Mar. 2012), 49–57.