Formal Modeling and Analysis of Cassandra in Maude

Si Liu, Muntasir Raihan Rahman, Stephen Skeirik,
Indranil Gupta, and José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract. Distributed key-value stores are quickly becoming a key component of cloud computing systems. In order to improve read/write latency, distributed key-value stores offer weak notions of consistency to clients by using many complex design decisions. However, it is challenging to formally analyze consistency behaviors of such systems, both because there are few formal models, and because different consistency level combinations render understanding hard, particularly under communication latency. This paper presents for the first time a formal executable model in Maude of Cassandra, a popular key-value store. We formally models Cassandra's main components and design strategies. We formally specify various *consistency properties* and model check them against our model under various communication latency and consistency combinations.

1 Introduction

Distributed key-value (e.g., Cassandra [2], RIAK [1]) storage systems are increasingly being used to store and query data in today's industrial deployments. Many diverse companies and organizations are moving away from traditional strongly consistent databases and are instead using key-value/NoSQL stores in order to store huge data sets and tackle an increasing number of users. According to DB-Engines Ranking [3] by April 2014, Cassandra advanced into the top 10 most popular database engines among 216 systems, underlining the increasing popularity of key-value database systems.

Distributed key-value stores typically replicate data on multiple servers for greater availability in the presence of failures. Since any of such servers can now respond to client read requests, it becomes costly to always keep all the replicas synchronized. This creates a tension between *consistency* (keeping all replicas synchronized) and *availability* (replying to clients quickly), especially when the network is partitioned [7]. Whereas traditional databases favor consistency over availability, distributed key-value stores risk exposing stale data to clients to remain highly available. This approach was popularized by the Dynamo [20] key-value store architecture from Amazon. Cassandra [2] is an open-source distributed key-value store which closely follows the Dynamo architecture. Many large scale Internet service companies like Netflix, IBM, HP, Facebook, Spotify, and PBS Kids rely heavily on the Cassandra key-value storage system.

Weakly consistent key-value stores like Cassandra typically employ many *complex design decisions* that can impact the consistency and availability guarantees offered to the clients. Therefore, there is an urgent need to develop *formal models* for specifying these design decisions and for reasoning about the impact of these design choices on specified consistency (correctness) and availability (performance) guarantees. For distributed key-value stores like Cassandra, at present the only way to understand the inner workings of such a system is to browse the huge code base. For example, the latest version of Apache Cassandra has $342,519$ lines of code. An important part of this work has been to study in detail the Cassandra code for its main components, to incrementally build Maude formal models of such components, and to check that the formal models faithfully capture the Cassandra design decisions. This is one of our main contributions, providing a solid basis for the subsequent formal analysis.

Once we have a formal executable model for Cassandra, we can conveniently model check various important properties about the system. Although we know Cassandra favors eventual consistency, there is no formal treatment that specifies when Cassandra satisfies eventual consistency and when it might actually offer strong consistency. Therefore it is very important to formally specify various consistency properties and check whether the system satisfies those properties under various combinations of message delays and consistency levels.

Currently there are two main approaches for verifying consistency models for distributed key-value stores. First, we can *run a given key-value store under a particular environment*, and *audit* the read/write operation logs to check for consistency violations [17]. Second, we can *analyze the algorithms used by the key-value store to ensure consistency* [15]. However, the former is not guaranteed to find all violations of the consistency model, and the latter is time-consuming and needs to be repeated for every system with different implementations of the underlying algorithms for guaranteeing consistency.

In this paper, we present a formal executable model of Cassandra using Maude [8], a modeling language based on rewriting logic that has proved suitable for modeling and analyzing distributed systems. Our Maude model includes main components of Cassandra such as data partitioning strategies, consistency levels, and timestamp policies for ordering multiple versions of data (the details of these components are given in Section 2.1). We have also specified Cassandra's main consistency properties, *strong consistency* and *eventual consistency*, in *linear temporal logic* (LTL). We have then model checked these properties using Maude's built-in LTL model checker. As a result, we can model check whether the properties hold or not for possibly different delay distributions between clients and servers, and for various combinations of consistency levels used by clients. Our analysis results indicate that eventual consistency is always satisfied by our Cassandra model. However, we discovered violations of the strong consistency property under certain scenarios. Although Cassandra is expected to violate strong consistency under certain conditions, previously there was no formal way of discovering under which conditions such violations could occur. At the software engineering level our formal modeling and analysis compares favor-

ably with previous approaches: our Maude specification is fewer than 1000 lines of code, and consistency verification in our formal approach is faster, more accurate and provides stronger assurance than existing approaches, which require analyzing the extremely large code base.

The two main contributions of this paper are:

- We present, to the best of our knowledge for the first time, a *formal executable model* for the Cassandra key-value store; this has required a large amount of effort due to Cassandra's extremely large code base.
- We formally specify and model check Cassandra' main consistency properties, namely, *strong consistency* and *eventual consistency*, and explicitly show when Cassandra satisfies these properties or not.

Due to lack of space we refer to the techincal report [14] for two more contributions: (i) the formal analysis of Cassandra's *read repair* mechanism, and (ii) the extension of our formal model to easily explore other *design decisions* for key-value stores.

The rest of the paper is organized as follows. Section 2 gives a brief overview of Cassandra and Maude. Next, in Section 3 and Section 4, we present our formal executable model for Cassandra, and the analysis of Cassandra consistency, respectively. Finally, in Section 5 we conclude and discuss related work and future directions.

2 Preliminaries

2.1 Cassandra Overview

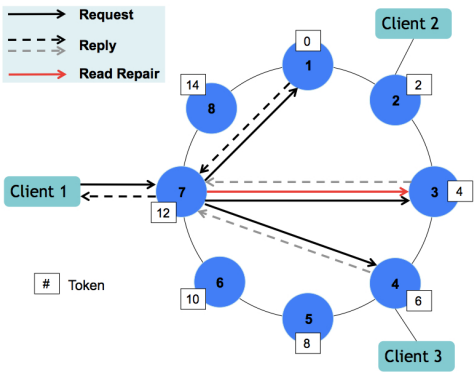

Fig. 1. The architecture of Cassandra deployed in a single data center with an 8 server ring of size 16

Apache Cassandra [2] is a high-performance, extremely scalable, and distributed NoSQL database solution. Cassandra dynamically partitions data across the cluster servers for scalability, so that data can be written to or read from any server in the system. The total amount of data managed by the cluster is represented as a *ring*. The ring is divided into *ranges*, with each server being responsible for one or more ranges of the data. Cassandra allows several *replica* servers to store a particular key-value pair. To place those replicas different strategies can be employed, e.g., the *Simple Strategy* places replicas clockwise in a single data center. For a private cloud, Cassandra is typically deployed in a single data center with a single ring structure shared by all its servers.

When a client issues a read/write request to a data center, a *coordinator* (a server in the cluster) forwards the request to all the replicas that hold copies of the same data. However, based on the specified *consistency level* for a read/write, the coordinator will reply back to the client after receiving responses from some of the replicas. This improves operation latency, since the coordinator does not have to wait for a slow server. Thus, the consistency level is the main dial to trade between consistency and availability. Cassandra supports three main consistency levels, `ONE`, `QUORUM` and `ALL`, for single data center settings, meaning that the coordinator will reply back to the client after hearing from *one* replica, a *majority* of replicas, or *all* replicas. To ensure that all replicas have the *most recent* version of any data item, Cassandra employs a *read repair* mechanism to update the out-of-date replicas.

Fig. 1 shows the architecture of Cassandra deployed in a single data center with an 8 server ring of size 16 (indicated by the tokens), where all servers are placed clockwise with each responsible for the region of the ring between itself (inclusive) and its successor (exclusive). An incoming read/write from client 1 will go to all 3 (the *replication factor* in this case) replicas (indicated by the black solid arrows). If the read/write consistency level specified by the client is `ONE`, the first node to complete the read/write (node `1` in this example) responds back to the coordinator `7`, which then forwards the value or acknowledgement back to the client (indicated by the black dashed arrows). In the case of a read, later replies from the remaining replicas, nodes `3` and `4`, may report different versions of the value w.r.t. the key issued by the client (indicated by the grey dashed arrows). If replica `3` holds an out-of-date value, in the background, `7` then issues a write (called a read repair) with the most recent value to `3` (indicated by the red arrow). Note that in a data center different clients may connect to different coordinators, and all servers maintain the same view on the ring structure. Thus, requests from any client on the same key will be always forwarded by the associated coordinator to the same replicas.

2.2 Actors and LTL Model Checking in Maude

Maude [8] is a language and tool that supports the formal specification and analysis of a wide range of concurrent systems. A Maude module specifies a *rewrite theory* $(\Sigma, E \cup A, R)$, where Σ is an algebraic *signature* declaring *sorts*, *subsorts*, and *function symbols*; $(\Sigma, E \cup A)$ is a *membership equational logic theory* [8], with E a set of possibly conditional equations, and A a set of equational axioms such as associativity, commutativity, and identity; $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type; R is a set of *labeled conditional rewrite rules* specifying the system's *local transitions*, each of which has the form $[l] : t \longrightarrow t'$ **if** $\bigwedge_{j=1}^{m} cond_j$, where each $cond_j$ is either an equality $u_j = v_j$ or a rewrite $t_j \longrightarrow t'_j$, and l is a *label*. Such a rule specifies a transition from an instance of t to the corresponding instance of t', *provided* the condition holds.

The Actor Model in Maude. The *actor model of computation* [4] is a model of concurrent computation based on asynchronous message passing between objects

called *actors*. Following the ideas of [16] and [10], the distributed state of an actor system is formalized as a *multiset* of objects and messages, including a scheduler object that keeps track of the *global time*. Multiset union is denoted by an associative and commutative juxtaposition operator, so that rewriting is *multiset rewriting*. An *actor* of the form `< id : class | a1 : v1, a2 : v2, ..., an : vn >` is an object instance of the class `class` that encapsulates the attributes `a1` to `an` with the current values `v1` to `vn`, respectively, and can be addressed using a unique name `id`. Actors communicate with each other using asynchronous messages. Upon receiving a message, an actor can change its state and can send messages to other actors. Actors can model a distributed systems such as Cassandra in a natural way. For example, the rewrite rule

```
rl [l] :  m(O,w)   < O : C | a1 : x, a2 : O' >
     =>   < O : C | a1 : x + w, a2 : O' >   m'(O',x) .
```

defines transitions where a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated.

Formal Analysis. In this paper we use Maude's *linear temporal logic model checker*, which analyzes whether *each* behavior from an initial state satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`. Their semantics is defined by conditional equations of the form: `ceq` *statePattern* `|=` *prop* `=` *b* `if` *cond* `.`, for *b* a term of sort `Bool`, stating that *prop* evaluates to *b* in states that are instances of *statePattern* when the condition *cond* holds. These equations together define *prop* to hold in all states t where t `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The model checking command `red modelCheck(`t, φ`) .` checks whether the temporal logic formula φ holds in all behaviors starting from the initial state t.

3 Formalizing Cassandra

This section presents a formal model of Cassandra. Section 3.1 shows how a Cassandra ring structure is specified in Maude, Section 3.2 describes the models of clients and servers, and Section 3.3 shows how we model messages, time and message delays and formalizes Cassandra's dynamic behaviors. The entire executable Maude specification is available at `https://sites.google.com/site/siliunobi/icfem-cassandra`.

3.1 Modeling the Ring Structure

Cassandra partitions data across a ring of cluster servers and allows several (replication factor) replicas to store a particular key-value pair. For replica placement we specify the *Simple Strategy* [12], which places replicas clockwise in a single data center without considering topology.

We model the Cassandra ring as a ring structure with natural numbers modulo the parametric ring size `RingSize`. Each token on the ring (see Fig. 1) is modeled as a pair of sort `RingPair` of `Position` and `Address`, referring to the position locating the token and the server responsible for it respectively. Each server claims the region of the ring between itself (inclusive) and its successor (exclusive). As an example, a simple 4 server ring of size 16 is specified as `(([0],1),([4],2),([8],3),([12],4))`, where the first server `1` is responsible for the range `[0...3]`, the server `2` for `[4...7]`, the server `3` for `[8...11]` and the server `4` for `[12...15]`.

3.2 Modeling Clients and Servers

Clients. A client in our model generates read or write requests. Our clients also collect responses for analysis purposes, such as checking consistency violations. We can have *one* or *multiple* clients, depending on the analysis we perform (see Section 4.1). We model a client actor with attributes for the address of the coordinator it recognizes, a store used to save the incoming messages, two queues of read/write requests that are ready or pending for sending out, and the corresponding set of locked keys. A key is *locked* if a client issues a read/write on it, and will be unlocked upon the client receiving the associated value/acknowledgement. This ensures that requests from the same client on the same key are ordered, i.e., the client locks the key until the preceding operation completes.

Servers. All servers in Cassandra are peers of each other, and a client's read/write request can reach any server in the cluster. We do not distinguish a coordinator from a replica, since the server receiving the client's request will automatically serve as the coordinator. In addition, a coordinator can also be one of the replicas that store a particular key-value pair. We model a server actor with attributes for a global ring, a table storing data, a buffer (empty if not a coordinator) caching the requests generated for replicas, and a finite set of message delays that are chosen nondeterministically for each outgoing message (see Section 3.3).

As an example, an instance of a client/server can be initialized as:

```
< 100 : Client | coord: 1, store: nil,
  requestQueue: (r1 r2), lockedKey: empty,
  pendingQueue: nil >
```

```
< 1 : Server | ring: (([0],1),([4],2),([8],3),
  ([12],4)), table: (3 |-> ("tea",10.0),
  8 |-> ("coffee",5.0), 10 |-> ("water", 0.0),
  15 |-> ("coke",2.0)), buffer: empty,
  delays: (1.0,2.0,4.0,8.0) >
```

where client `100` connects to coordinator `1`, and intends to send out two requests `r1` and `r2`; server `1` has a view of the ring structure, a set of four possible delays, and a local key-value store modeled using the predefined data module `MAP` in Maude, with each key of sort `Key` mapped to a pair of sort `TableData` consisting of the data of sort `Value` and the timestamp of sort `Float`. For example, key `8` maps to data `("coffee",5.0)`, indicating `"coffee"` is written into Server `1` at global time `5.0s`.

3.3 Formalizing Reads and Writes in Cassandra

Messages and Delays. Regarding its delivery, a message is either *active* (ready for delivery) or *inactive* (scheduled for delivery), depending on whether the associated delay has elapsed, which is determined by the system's *global clock* [10]. An active or inactive message is of the format `{T,MSG}` or `[D,MSG]`, respectively, where `D` refers to the delay of message `MSG`, and `T` refers to the global time when `MSG` is ready for delivery. For example, if message `[D,MSG]` is generated at the current global time `GT`, `T` equals to `GT + D`.

All messages have associated delays. To simplify the model without losing generality, we abstract those delays into two kinds: (i) the delay between a client and a coordinator, and (ii) the delay between a coordinator and a replica. Since we equip a coordinator with a *finite* delay set, as shown in Fig. 2, nondeterministic delays will be added to the messages generated at its side. Fig. 2 shows example delays `D4` for a read/write reply from the coordinator to the client, and `D2` for a read/write request from the coordinator to the replica, with the other two delays, `D1` and `D3`, set to `0.0s`.

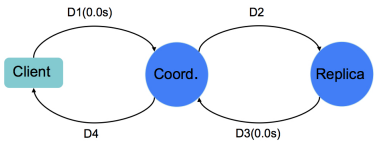

Fig. 2. Message Delays

To appropriately schedule messages, we introduce a *scheduler* object of the form `{GT|MS}` which maintains a global clock `GT` indicating the current global time, and provides a deterministic ordering of messages `MS` [10]. Inactive messages are inserted into the scheduler. When the global time in the scheduler is advanced to the moment when some delay expires, the associated message becomes active and is consumed by the target client/server. Therefore, the scheduler can be also considered as an object that advances the model's global time.

In the context of reads/writes, a message in our model can be defined based on (i) whether it is a read or write; (ii) whether it is a request or a reply; and (iii) which side it is from/to. For example, `RReqCS(...)` is a client-to-coordinator read request; and `WRepSS(...)` is a replica-to-coordinator write reply.

Reads and Writes. As mentioned in Section 2.1, communication can be between a client, a coordinator and several replicas. Due to space limitations, we only illustrate the model dynamics: (i) at the client side for generating requests, and (ii) at the coordinator side for routing read requests to the replicas.[1] We refer the reader to our longer report [14] for additional details.

(i) Generating requests by clients. In Cassandra, a client always generates *strictly ordered* requests w.r.t. a certain key. More precisely, when a client wants to issue a read/write (triggered by a `bootstrap` message to itself), it looks up the associated key in the set of locked keys. If the key is locked, the request will be added to the pending queue; otherwise, the request will be sent out to the

[1] We often omit the type declaration for the mathematical variables in the rules, but follow the Maude convention that variables are written in capital letters.

coordinator, and then the key will be locked. To emit all requests that are not restricted by the locked keys, a client will iteratively send itself a `bootstrap` message to trigger the above process until the request queue is empty. Thus, given a key, an associated read/write will block all subsequent operations with the same key until accomplished. The following rewrite rule illustrates the case when a client successfully sends out a request:

```
crl [CLIENT-REQUEST] :
    < A : Client | coord: S, requestQueue: Q, lockedKey: KS, AS > {T, A <- bootstrap}
 => < A : Client | coord: S, requestQueue: tail(Q), lockedKey: add(H,KS), AS >
    [d1, S <- request(H,T)] [ds, A <- bootstrap]
  if H := head(Q) /\ Q =/= nil /\ not pending(H,KS) .
```

where the global time `T` is put into the outgoing request `H` by function `request`, meaning that the *timestamp* in a client's request should be the moment when it is generated [12]. `d1` (corresponding to `D1` in Fig. 2) and `ds` refer to the *delays* for a client-to-coordinator request and a self-triggered message respectively. Function `pending` determines whether a key is locked. Note that requests generated by different clients are *independent* of each other, i.e., it is not the case that a read/write will block all subsequent operations from other clients on the same key until accomplished.

(ii) Forwarding reads by the coordinator. As shown below, upon receiving the request on key `K` with consistency level `L` from client `A`, coordinator `S` updates the local buffer with the received information for each request `ID`, and generates the appropriate number of read requests for the replicas according to replication factor `fa`. Function `rpl` returns a set of replica addresses:

```
crl [COORD-FORWARD-READ-REQUEST] :
    < S : Server | ring: R, buffer: B, delays: DS, AS > {T, S <- RReqCS(ID,K,L,A)}
 => < S : Server | ring: R, buffer: insert(ID,fa,L,K,B), delays: DS,AS > C
    if generate(ID,K,DS,rpl(K,R,fa),S,A) => C .
```

The coordinator nondeterministically selects a message delay `D` for each outgoing request. The following rewrite rule together with the above one show how the coordinator will send each replica a read request with a nondeterministically chosen delay:

```
rl [GENERATE-READ-REQUEST] : generate(ID,K,(D,DS),(A',AD'),S,A)
=> [D, A' <- RReqSS(ID,K,S,A)] generate(ID,K,(D,DS),AD',S,A) .
```

where the replica address `A'` is iteratively selected from the address set returned by `rpl`, and the message delay `D` (corresponding to `D2` in Fig. 2) is nondeterministically selected from the delay set.

4 Formal Analysis of Consistency in the Cassandra Model

In this section we formally analyze the Cassandra model built in Section 3, and check for *consistency violations* under various latency and consistency level

combinations. Section 4.1 presents the main consistency properties we want to check. Sections 4.2 and 4.3 describe the formal analysis of those properties with one or multiple clients, where the property formalizations, experimental scenarios and model checking results are shown, respectively.

4.1 Consistency Properties

Strong Consistency. A key-value system satisfies strong consistency if each read returns the value of the latest write that occurred before that read. More precisely, let $Tr = o_1, o_2, ..., o_n$ denote a trace of n read/write operations issued by *one* or *more* clients in a key-value system S, where any operation o_i can be expressed as $o_i = (k, v, t)$, where t denotes the *global* time when o_i was issued, and v is the value returned from key k if it is a read, or the value written to key k if it is a write. S satisfies *strong consistency* if for any read $o_i = (k, v_i, t_i)$, provided there exists a write $o_j = (k, v_j, t_j)$ with $t_j < t_i$, and without any other write $o_h = (k, v_h, t_h)$ such that $t_j < t_h < t_i$, we have $v_i = v_j$.

Eventual Consistency. If no new updates are made to a key, a key-value system is *eventually consistent*, if eventually all reads to that key will return the last updated value. More precisely, we again consider a trace $Tr = o_1, o_2, ..., o_n$ of n read/write operations in a key-value system S. Let $o_i = (k, v_i, t_i)$ be a write, and there is no other write $o_j = (k, v_j, t_j)$ such that $t_i < t_j$. S satisfies *eventual consistency* if there exists some $t > t_i$, and for all reads $o_h = (k, v_h, t_h)$ such that $t_h > t$, $v_i = v_h$.

To model check the above properties, we consider strong/eventual consistency experiment scenarios with *one* or *multiple* clients, because:

- from a single client's perspective, consecutive requests on the same key are always *strictly ordered*, i.e., subsequent operations will be pending until a preceding read/write finishes (see Section 3.3);
- for multiple clients, requests generated by different clients are *independent* of each other, i.e., it is not the case that a read/write will block all subsequent operations from other clients on the same key until accomplished.

The purpose of our experiments is to answer the following questions w.r.t. whether strong/eventual consistency is satisfied: does strong/eventual consistency depend on:

- with one client, the combination of consistency levels of consecutive requests?
- with multiple clients, additionally the latency between consecutive requests being issued?

Although a read repair mechanism is not generally used by other key-value systems, it is essential in Cassandra to guarantee consistency. The definition, experimental scenarios and model checking results w.r.t. the eventual consistency properties of the read repair mechanism can be found in our longer report [14].

4.2 Formal Analysis of Consistency with One Client

Scenarios. We define the following setting for our experimental scenarios with *one* client:

(a) We consider a data center with four servers, and a replication factor of `3`.
(b) The ring size is `16`, and the servers are responsible for the equal range clockwise, with the first server responsible for range `[0...3]`.
(c) The read/write consistency levels can be `ONE`, `QUORUM` or `ALL`.
(d) For strong/eventual consistency, the client issues two consecutive requests on the same key.
(e) The delay set is `(1.0,2.0,4.0,8.0)`.

Moreover, we consider the following scenarios, where, depending on the property and consistency level, we name each subcase Scenario (1-S/E-O/Q/A O/Q/A), e.g., Scenario (1-S-QA) refers to the case checking strong consistency with one client, where two consecutive requests have the consistency levels `QUORUM` and `ALL` respectively; Scenario (1-E-AO) refers to the case checking eventual consistency with one client, where two consecutive requests have the consistency levels `ALL` and `ONE`, respectively.

In Scenarios (1-S-$**$) the client issues a write on a key followed by a read on the same key. The initial state of this scenario is specified as follows:

```
eq c1 = one .    eq c2 = one .    --- consistency level: one, quorum or all
eq k = 10 .      eq v = "juice" .
eq initState = { 0.0 | nil } [0.0, 100 <- bootstrap]
   < 100 : Client | coord: 3, requestQueue: (WriteRequestCS(0,k,v,c1,100)
                    ReadRequestCS(1,k,c2,100)), ... >
   < 1 : Server | table: (3 |-> ("tea",0.0),  8 |-> ("coffee",0.0),
                          10 |-> ("water", 0.0), 15 |-> ("coke",0.0)), ...>
   < 2 : Server | ... > < 4 : Server | ... >
   < 3 : Server | ring: (([0],1),([4],2),([8],3),([12],4)), delays: (1.0,2.0,4.0,8.0), ... > .
```

where client `100` connects to server (coordinator) `3`, and servers `1`, `2` and `4` serve as the replicas. {`0.0 | nil`} refers to the initial state of the scheduler with the global time `0.0` and the schedule list `nil`.

In Scenarios (1-E-$**$) the client issues two consecutive writes on the same key. The initial state of this scenario is like that of Scenarios (1-S-$**$), except that the second operation is a write request.

Formalizing Consistency Properties. Regarding *strong consistency*, for Scenarios (1-S-$**$) we define a parameterized atomic proposition `strong(A,K,V)` that holds if we can match the value `V` returned by the subsequent read on key `K` in client `A`'s local store with that in the preceding write.

```
op strong : Address Value -> Prop .
eq < A : Client | store: (ID,K,V), ... > REST |= strong(A,K,V) = true .
```

Since two requests will eventually be consumed with the client receiving the returned value, the strong consistency property can in this case be formalized as the LTL formula `<> strong(...)`. Given an initial state `initConfig`, the following command returns `true` if the property eventually holds; otherwise, a trace showing a counterexample is provided.

```
red modelCheck(initConfig, <> strong(client,key,value)) .
```

Regarding *eventual consistency*, we only need to check if eventually all replicas will be consistent with the last updated value. Obviously, if all replicas agree on the last updated value, a sufficiently later read (e.g., after all replicas finish the update with the writes) will return such a value, regardless of the consistency level (we can also check this using the experiment for the read repair property [14]). Thus, we check at each replica's side by defining a parameterized atomic proposition `eventual(R1,R2,R3,K, V)` to hold if we can match the value `V` on key `K` in the subsequent (or the last) write with those in the local tables of all replicas `R1`, `R2` and `R3`.

```
op eventual : Address Address Address Key Value -> Prop .
eq < R1 : Server | table: (K |-> (V,T1), ...), ... >
   < R2 : Server | table: (K |-> (V,T2), ...), ... >
   < R3 : Server | table: (K |-> (V,T3), ...), ... > REST |= eventual(R1,R2,R3,K,V) = true .
```

The eventual consistency property can then be formalized as the LTL formula `<>[] eventual(...)`. Given an initial state `initConfig`, the following command returns `true` if the property eventually always holds; otherwise, a trace showing a counterexample is provided.

```
red modelCheck(initConfig, <>[] eventual(r1,r2,r3,key,value)) .
```

Analysis Results. The model checking results show that strong consistency holds in Scenarios (1-S-OA), (1-S-QQ), (1-S-QA), (1-S-AO), (1-S-AQ) and (1-S-AA), but not in Scenario (1-S-OO), (1-S-OQ) or (1-S-QO), and that eventual consistency holds in all its scenarios.

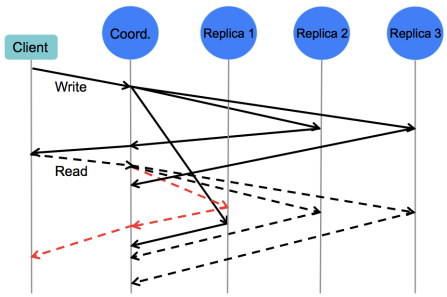

Fig. 3. Sequence chart for a strong consistency violation regarding a `ONE` write followed by a `ONE` read issued by one client

For strong consistency, we show the experimental results in Table 1, with a cross (×) marking a violation. Note that three out of nine combinations of consistency levels violate strong consistency, where at least one of the read and the write has a consistency level weaker than `QUORUM`. Fig. 3 illustrates how a strong consistency violation happens w.r.t. a `ONE` write *strictly* followed by a `ONE` read, where the red dashed arrow identifies the violation.

The reason why a strong consistency violation occurs is that some read forwarded by the coordinator reaches a certain replica before a write request does. Thus, it seems fair to say that unless a client uses consistency level at least `QUORUM`, strong consistency may not be guaranteed.

Table 1 also shows the results of model checking eventual consistency, where no violation occurs, regardless of the consistency level combinations. The reason

why eventual consistency holds is that a replica updates its table only if the incoming request has a higher timestamp, and therefore, even if the first request reaches the replica later due to message delay, it will be simply discarded. As a result, the replica will eventually store the value in the last generated request.

Table 1. Results on Checking Strong (indicated by the left table) and Eventual Consistency (indicated by the right table) with One Client

Write_1 \ Read_2	`ONE`	`QUORUM`	`ALL`
`ONE`	×	×	✓
`QUORUM`	×	✓	✓
`ALL`	✓	✓	✓

Write_1 \ Write_2	`ONE`	`QUORUM`	`ALL`
`ONE`	✓	✓	✓
`QUORUM`	✓	✓	✓
`ALL`	✓	✓	✓

4.3 Formal Analysis of Consistency with Multiple Clients

When dealing with multiple clients, requests generated by different clients are *independent of each other*, i.e., it is not the case that a read/write will block all subsequent operations from other clients with the same key until accomplished. Whenever a request is issued by a client, it will be immediately sent out. Thus it will make no difference which consistency level a preceding request uses. For example, if client 2 intends to issue a request R' L seconds after client 1 sends out its request R at global time T, R' will be exactly sent out at global time $T + L$, despite of the arrival of R at client 1. However, as mentioned in Section 3, a single client's consecutive requests are always strictly ordered.

Scenarios. We consider scenarios with two clients, where client 1 first issues its request, and after some time (the latency between two requests) client 2 issues its own. Regarding consistency levels, we fix it as `QUORUM` for the first request, but parameterize it for the second request. We still use three replicas to store a particular key-value pair. For each replica we have two possibilities about the second request's arrival, either before or after the first request. Therefore, there are eight possible cases (for simplicity we do not consider the case where two requests arrive at the same time). Thus, we need a delay set with at least two different delays. Concretely, we define the following setting for our experimental scenarios with multiple clients:

- (a), (b) and (c) in the Section 4.2 setting for one-client scenarios.
- We consider two clients, each one issuing one request on the same key. Clients 1 and 2 connect to coordinators 1 and 2 respectively.
- The delay set is `(2.0,8.0)` for coordinator 1, and `(0.0)` for coordinator 2.
- The latency between two requests can be `1.0s`, `5.0s` or `10.0s`.

Note that, given coordinator 1's delay set, coordinator 2's delay set combined with three different latencies *fully* covers all possible orders of arrivals of two requests.

For example, the initial state of model checking strong consistency with two clients is specified as:

```
...
eq l = 1.0 .    --- latency: 1.0, 5.0 or 10.0
eq initState = ... [l, 200 <- bootstrap]
   < 100 : Client | coord: 3, requestQueue: (WriteRequestCS(0,k,v,c1,100), ... >
   < 200 : Client | coord: 1, requestQueue: (ReadRequestCS(1,k,c2,200)), ... >
   < 1 : Server | delays: (0.0), ... > < 3 : Server | delays: (2.0,8.0), ... > .
```

Table 2. Results on Checking Strong Consistency (indicated by the top table) and Eventual Consistency (indicated by the bottom table) with Two Clients (The delay set is {D1,D2} with D1<D2.)

Strong

Latency \ Consistency Lv.	ONE	QUORUM	ALL
L1 (L1<D1)	×	×	×
L2 (D1<L2<D2)	×	×	×
L3 (D2<L3)	✓	✓	✓

Eventual

Latency \ Consistency Lv.	ONE	QUORUM	ALL
L1 (L1<D1)	✓	✓	✓
L2 (D1<L2<D2)	✓	✓	✓
L3 (D2<L3)	✓	✓	✓

Table 3. Detailed Results for Latency L2 on Checking Strong Consistency with Two Clients (R1, R2 and R3 refer to the replicas. "1/2" means that client 1/2's request reaches the corresponding replica first. The delay set is {D1,D2} with D1<D2.)

Latency	**First Arrival**			**Consistency Level**		
	R1	R2	R3	ONE	QUORUM	ALL
L2 (D1<L2<D2)	1	1	1	✓	✓	✓
	1	1	2	×	✓	✓
	1	2	1	×	✓	✓
	1	2	2	×	×	✓
	2	1	1	×	✓	✓
	2	1	2	×	×	✓
	2	2	1	×	×	✓
	2	2	2	×	×	×

Analysis Results. For model checking strong/eventual consistency with two clients, we use the same formalization of each property as mentioned in Section 4.2. The results of model checking *strong consistency* with two clients is shown in Table 2. We observe that whether strong consistency is satisfied or not *depends on the latency* between requests: if it is so high that all preceding requests have reached the replicas, strong consistency holds (case L3); otherwise, strong consistency does not hold, because there is at least one subcase where the

later requests reach the replicas before the preceding requests do. Although both cases L1 and L2 fail to satisfy strong consistency as shown by the model checking, we can however find some particular execution sequences in case L2 (not in case L1, because its latency is extremely low), where client 2's read returns the (most recent) value in client 1's write.[2] In Table 3 we list all possbile subcases for case L2, where we mark "1" if client 1's request reaches the corresponding replica first, otherwise "2", and a case of the form "1/2 1/2 1/2" describes which requests reach the three replicas `R1`, `R2` and `R3` first respectively. For case L2, except the two extreme subcases "1 1 1" and "2 2 2", strong consistency holds, on the one hand, if the subsequent read uses consistency level `ALL`; on the other hand, if a majority of replicas receive client 1's request first. For example, in subcase "2 1 1", even if one forwarded request by client 2 reaches `R1` first, by using consistency level at least `QUORUM` in client 2's read, strong consistency holds. In subcase "2 2 1", since two later requests reach the replicas first, strong consistency holds only if client 2 uses consistency level `ALL`. Thus with two clients (we also believe with more than two clients), it is fair to say that: (i) strong consistency depends on the latency between requests; and (ii) except extreme latencies (almost simultaneous or extremely high), to maximize strong consistency we ought to use consistency level `ALL` for subsequent requests.

Instead, as mentioned in Section 4.2, strong consistency with one client depends on the combination of consistency levels.

For *eventual consistency*, Table 2 shows the experimental results, where no violation occurs, regardless of the consistency level and latency between requests. Eventual consistency holds for two clients for the same reason why it holds for one client, i.e., a replica updates its table only if the incoming request has a higher timestamp. That is, eventual consistency is always guaranteed by our Cassandra model, regardless of whether we have one or multiple clients.

5 Related Work and Concluding Remarks

Models in Key-value/NoSQL Stores. Amazon's Dynamo [20] was the first system that adopted the eventual consistency model [21]. Recent work on consistency benchmarking includes delta consistency metrics [17], which mine logs to find consistency violations. PBS [5] proposes probabilistic notions of consistency. Compared to this, our model checking based approach can exhaustively search for all possible consistency violations. Compared to eventual-consistency based model, a slightly stronger model is causal+ consistency [15], which maintains causality of reads across writes which are causally related. Red-blue consistency [13] modifies transaction operations into blue operations which are commutatable at datacenters, and red ones, which are serialized across all datacenters.

[2] Using Maude's *search* command we can explore the reachable state space for a particular pattern, where the value received by client 2 matches that in client 1's write. See the Maude specification available at `https://sites.google.com/site/siliunobi/icfem-cassandra`.

Commutative replicated data types (CRDTs) are distributed data structures insensitive to the order of operations on a key [18], which is being incorporated by RIAK [1].

Model Checking Key-value/NoSQL Stores. Despite the importance of such stores, we are not aware of other work formalizing and verifying them with formal verification tools. There is however some recent related work on formal analysis of *cloud computing* systems, including, e.g., [6], which addresses eventual consistency verification by reducing it to reachability and model checking problems; [19], which formally models and analyzes availability properties of a ZooKeeper-based group key management service; [9], which proposes and analyzes DoS resilience mechanisms for cloud-based systems; [22], which gives formal semantics to the KLAIM language and uses it to specify and analyze cloud-based architectures; and [11], which presents a formal model of Megastore —not really a key-value store, but a hybrid between a NoSQL store and a relational database— in Real-Time Maude which has been simulated for QoS estimation and model checked for functional correctness.

Our main focus in this work has been twofold: (i) to obtain for the first time a *formal model* of Cassandra; and (ii) to formally analyze its *correctness* properties, focusing on the crucial issue of Cassandra's *consistency* properties. This work presents for the first time a detailed formal analysis of the conditions under which Cassandra can achieve strong or eventual consistency.

The other side of the coin is *availability*: weaker consistency is the price paid to achieve greater availability. Therefore, as future work we plan to *explore the various design choices* in the consistency-availability spectrum for key-value stores. Formal modeling with *probabilistic rewrite rules* and formal analysis by *statistical model checking*, like done, e.g., in [9,10,22], seems a natural next step for studying *availability and other QoS properties*. Following those ideas, our current Cassandra model can be naturally extended for further statistical model checking (though Real-Time Maude already has a notion of time, there is no systematic support for probabilistic real-time rewrite theories, which is one reason we did not build our model in Real-Time Maude). More broadly, our long-term goal is not Cassandra per se, but developing a library of *formally specified executable components* embodying the key functionalities of key-value stores. We plan to use such components and the formal analysis of their correctness and QoS properties to facilitate the exploration of the design space for such systems.

Acknowledgments. We thank the anonymous reviewers for helpful comments on a previous version of this paper. This work has been partially supported by AFOSR Contract FA8750-11-2-0084, NSF Grant CNS 13-19109, NSF CNS 13-19527 and NSF CCF 09-64471.

References

1. Basho Riak, `http://basho.com/riak/`

2. Cassandra, http://cassandra.apache.org/
3. DB-Engines, http://db-engines.com/en/ranking
4. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, USA (1986)
5. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. Proc. VLDB Endow. 5(8), 776–787
6. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: POPL. pp. 285–296 (2014)
7. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC. p. 7 (2000)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
9. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: FASE. pp. 78–93 (2012)
10. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: WADT. pp. 143–160 (2012)
11. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Specification, Algebra, and Software. pp. 494–519 (2014)
12. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media, Sebastopol, CA, USA (2010)
13. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI). pp. 265–278 (2012)
14. Liu, S., Rahman, M., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude (2014), https://sites.google.com/site/siliunobi/icfem-cassandra
15. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: Proc. ACM Symposium on Operating Systems Principles (SOSP). pp. 401–416 (2011)
16. Meseguer, J., Talcott, C.L.: Semantic models for distributed object reflection. In: Proc. European Conference on Object-Oriented Programming (ECOOP). pp. 1–36 (2002)
17. Rahman, M.R., Golab, W., AuYoung, A., Keeton, K., Wylie, J.J.: Toward a principled framework for benchmarking consistency. In: Proc. USENIX Workshop on Hot Topics in System Dependability (HotDep) (2012)
18. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Convergent and commutative replicated data types. Bulletin of the EATCS 104, 67–88 (2011)
19. Skeirik, S., Bobba, R.B., Meseguer, J.: Formal analysis of fault-tolerant group key management using ZooKeeper. In: Proc. Symposium on Cluster, Cloud, and Grid Computing (CCGRID). pp. 636–641 (2013)
20. Vogels, W.: Amazon's dynamo. All Things Distributed (October 2007), http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
21. Vogels, W.: Eventually consistent. ACM Queue 6(6), 14–19 (2008)
22. Wirsing, M., Eckhardt, J., Mühlbauer, T., Meseguer, J.: Design and analysis of cloud-based architectures with KLAIM and Maude. In: WRLA. pp. 54–82 (2012)