View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

Formal Modeling and Analysis of RAMP Transaction Systems in Maude

Si Liu[1], Peter Csaba Ölveczky[2], Muntasir Raihan Rahman[1], Jatin Ganhotra[1], Indranil Gupta[1], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo

Abstract. To cope with ever-increasing data sets, distributed data stores partition their data across servers. However, real-world systems usually do not provide useful transactional semantics for operations accessing multiple partitions due to the delays involved in achieving multi-partition consistency. Read Atomic Multi-Partition (RAMP) transactions have recently been proposed as efficient light-weight multi-partition transactions that guarantee read atomicity: either all updates or no updates of a transaction are visible to other transactions. In this paper we formalize RAMP transactions in rewriting logic and perform model checking verification of key properties using the Maude tool. In particular, we develop detailed formal models—and formally analyze—a number of extensions and optimizations of RAMP that are only briefly mentioned by the RAMP developers.

1 Introduction

The success of cloud computing relies on software systems that store large amounts of data correctly and efficiently. It is hard to satisfy both these requirements in a distributed storage system. While traditional relational databases (like MySQL) offer strong notions of correctness (such as ACID properties) when multiple clients access data (via transactions), these systems are considered too slow for today's workloads and applications. As a result, a new generation of storage systems called NoSQL ("Not only SQL") have emerged, and are already a multi-billion dollar industry [3]. The NoSQL era was kicked off by systems from Google [8] and Facebook [13], and includes open-source systems that are wildly popular in industry today [1,2,12]. These systems offer weak notions of correctness, such as "eventual consistency," while enabling operations on stored data to execute orders of magnitude faster than relational databases.

At the heart of this dichotomy is the CAP theorem [7,16], which says that it is impossible to have both efficiency (low latency) and strong consistency (correctness) in a distributed storage system.[1] Several efforts have recently emerged to bridge this gap, and in a sense to circumvent the CAP theorem. The RAMP

[1] When there are partitions present. We exclude this clause because partitions are endemic in distributed systems in today's Internet.

system of transactions proposed by Bailis *et al.* [4,5] is one of the most promising among those proposals. RAMP allows clients to execute transactions (like in relational databases) in NoSQL-like storage systems. It offers a correctness property called "Read Atomicity" (RA) which ensures that a given transaction's updates are either all visible or not visible at all, to other transactions.[2] RAMP also allows data to be partitioned across multiple partitions.

The original RAMP paper [5] presented experimental results that demonstrated its efficiency properties of low latency and high throughput; it also gave "hand proofs" that RAMP in fact provides read atomicity. The paper [5] also briefly mentions a number of extensions and optimizations, such as faster commit and one-phase writes; however, no details or correctness proofs for these extensions are given in [5].

There is therefore a clear need for formally specifying and analyzing RAMP and its proposed extensions. Indeed, a main contribution of our work is to develop detailed models, and to formally analyze, a number of those briefly-mentioned extensions and variants of RAMP. Providing correctness in the overall protocol and its extensions are a critical step towards making RAMP a production-capable system. Without such analysis, developers and deployers might enable or disable particular extensions at will, without knowing the correctness ramifications.

In this paper, we therefore use the expressive Maude formal specification language [9] to provide the first executable formal model of RAMP and a number of its extensions. This approach allows us to analyze the correctness properties of RAMP in a fully-automated manner, rather than relying only on hand proofs. However, explicit-state model checking can only analyze the system from *single* initial configurations, which provide limited coverage. One of the new contributions of this paper is therefore a general technique in Maude for model checking a protocol for *all possible* initial configurations up to certain bounds. We use this technique to analyze some of the (unproved) conjectures in the original RAMP paper—in particular, how some of the extensions and optimizations of RAMP affect its behavior. We model and analyze the effect of the following RAMP building blocks: fast commit, one-phase write, and two-phase commit.

We explore not only the "strong" notion of read atomicity correctness but also a weaker consistency model called "read-your-writes" (whereby a client is assured of being able to read at least its own latest writes). For instance, we find that two-phase commit is a necessary building block for ensuring read atomicity, and that for one-phase writes even the weak correctness notion of read-your-writes cannot be guaranteed!

we explain various consistency models in greater detail including their formal definitions (Section 3) and specifications (Section 5.1). Second, we provide quantitative analysis of consistency using both statistical model checking and implementation-based estimation for three new consistency models (Section 5.2): monotonic reads, consistent pefix and causal consistency [37, 38, 30, 8].

[2] While RA is stronger than eventual consistency, it is still not equivalent to ACID as it is not serializable.

The rest of this paper is structured as follows. Section 2 gives some background on RAMP and Maude. Section 3 presents our formal model of RAMP and its extensions. Section 4 explains how the different consistency levels and other key properties of RAMP can be formally specified as reachability properties, which can then be analyzed using Maude. Finally, Section 5 discusses related work and Section 6 gives some concluding remarks.

2 Preliminaries

2.1 Read-Atomic Multi-Partition (RAMP) Transactions

To deal with ever-increasing amounts of data, distributed databases *partition* their data across multiple servers. Unfortunately, many real-world systems do not provide useful transactional semantics for operations accessing multiple partitions, since the latency needed to ensure correct multi-partition transactional access is often high. Therefore, trade-offs that combine efficiency with weaker transactional guarantees for operations accessing multiple partitions are needed.

In [5], Bailis *et al.* propose a new isolation model, called *read atomic* (RA) isolation, and *Read Atomic Multi-Partition* (RAMP) transactions, that together combine efficient multi-partition operations and partial fault tolerance with some kind of transactional guarantee: either all or none of a transaction's updates are visible to other transactions. For example, if A and B become "friends" in one transaction, then other transactions should *not* see that A is a friend of B but that B is not a friend of A; either both relationships are visible or neither is.

RAMP trades serializability for a weaker isolation model (read atomic isolation) and high availability and efficiency, where transactions are never blocked waiting for concurrent transactions. Read isolation of RAMP transactions should nevertheless be useful for a number of real-world applications such as foreign key constraints, secondary indexing, and materialized view maintenance [5]. It is also worth noting that RAMP does not consider replication of items across partitions.

RAMP transactions use metadata and multi-versioning. RAMP writers attach metadata to each write and the reads use this metadata to get the correct version. There are three versions of RAMP, which offer different trade-offs between the size of the attached metadata and performance:[3] RAMP-Fast, RAMP-Small, and RAMP-Hybrid. The write protocols in these algorithms only differ in the amount of attached metadata. To guarantee that all partitions perform a transaction successfully or that none do, RAMP performs two-phase writes by using the two-phase commit protocol (2PC). 2PC involves two phases: In the *prepare* phase, each timestamped write is sent to the respective partition and each partition adds the write to its local database. In the *commit* phase, each partition updates an index which contains the highest-timestamped committed version of each item. The RAMP algorithms described in [5] only deal with read-only and write-only transactions.

[3] Performance is measured in terms of the expected number of extra reads required to fetch the corresponding missing writes.

RAMP-Fast. In RAMP-Fast, the size of the metadata of a transaction is linear in the number of writes in the transaction. Read operations require one round trip time delay (RTT) in the common race-free case, and two RTTs in the worst case; writes require two RTTs. Read transactions proceed by first fetching the highest-timestamped committed *version* of each requested data item from the corresponding partition, and then decide if they have missed any version that has been prepared but not yet committed. The timestamp and the metadata from each version read produces a mapping from items to timestamps that represent the highest-timestamped write for each transaction, appearing in the first-round read set. If the reader has a lower timestamp version than indicated in the mapping for that item, a second-round read will be issued to fetch the missing version. Once all the missing versions have been fetched, the client can return the resulting set of versions, which include both the first-round reads as well as any missing versions fetched in the second round of reads. The detailed specification of RAMP-Fast in [5] is shown in Fig. 1.

RAMP-Small. Unlike RAMP-Fast, RAMP-Small read transactions proceed by first fetching the highest committed *timestamp* of each requested data item; the readers then send the entire set of those timestamps in a second message. The highest-timestamped version that also exists in the received set will be returned to the reader by the corresponding partition. RAMP-Small transactions require constant-size metadata, but require two RTTs for reads and writes. RAMP-Small writes only store the transaction timestamp, instead of attaching the entire write set to each write.

RAMP-Hybrid. RAMP-Hybrid transactions provide an intermediate solution between RAMP-Fast and RAMP-Small. RAMP-Hybrid algorithms only store a Bloom filter which represents the transaction set, instead of storing the entire write set as in RAMP-Fast. The RAMP-Hybrid and RAMP-Fast read protocols are identical where the first round of communication fetches the last-committed version of each item from its partition and the second round involves fetching any higher timestamped writes for each item.

In this paper we focus on RAMP-Fast and RAMP-Small, which lie at the end-points of the spectrum (in terms of the attached metadata for each write).

Extensions of RAMP The paper [5] spells out the basic algorithms for RAMP-Fast, RAMP-Small, and RAMP-Hybrid in some detail. That paper also briefly discusses a number of extensions and optimizations of these basic algorithms, but without giving any details. A main contribution of our work is therefore to develop detailed models, and to formally analyze, a number of those extensions and variants of RAMP, including:

– RAMP with one-phase writes (1PW). The performance can be improved if a client does not wish to read her own writes. In this RAMP + 1PW variant, writes only require one *prepare* phase, as the client can execute

Algorithm 1 RAMP-Fast

```
    Server-side Data Structures
1:  $versions$: set of versions $\langle item, value$, timestamp $ts_v$, metadata $md \rangle$
2:  $latestCommit[i]$: last committed timestamp for item $i$
    Server-side Methods
3:  procedure PREPARE($v$ : version)
4:      $versions$.add($v$)
5:      return
6:  procedure COMMIT($ts_c$ : timestamp)
7:      $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
8:      $\forall i \in I_{ts}$, $latestCommit[i] \leftarrow \max(latestCommit[i], ts_c)$
9:  procedure GET($i$ : item, $ts_{req}$ : timestamp)
10:     if $ts_{req} = \emptyset$ then
11:         return $v \in versions : v.item = i \wedge v.ts_v = latestCommit[item]$
12:     else
13:         return $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$
    Client-side Methods
14: procedure PUT_ALL($W$ : set of $\langle item, value \rangle$)
15:     $ts_{tx} \leftarrow$ generate new timestamp
16:     $I_{tx} \leftarrow$ set of items in $W$
17:     parallel-for $\langle i, v \rangle \in W$
18:         $v \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\}) \rangle$
19:     invoke PREPARE($v$) on respective server (i.e., partition)
20:     parallel-for server $s$ : $s$ contains an item in $W$
21:         invoke COMMIT($ts_{tx}$) on $s$
22: procedure GET_ALL($I$ : set of items)
23:     $ret \leftarrow \{\}$
24:     parallel-for $i \in I$
25:         $ret[i] \leftarrow$ GET($i, \emptyset$)
26:     $v_{latest} \leftarrow \{\}$ (default value: $-1$)
27:     for response $r \in ret$ do
28:         for $i_{tx} \in r.md$ do
29:             $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
30:     parallel-for item $i \in I$
31:         if $v_{latest}[i] > ret[i].ts_v$ then
32:             $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
33:     return $ret$
```

Fig. 1. The RAMP-Fast algorithm as described in [5].

the *commit* phase asynchronously. We have analyzed both RAMP-Fast and RAMP-Small with one-phase writes.

– RAMP with faster commit (FC). If a server returns a version with the timestamp fresher than the highest committed version of the item, then the server can mark the version as committed. This allows faster updates to correct versioning and thus fewer round trip time delays.
– RAMP without two-phase commit. We have also experimented with the possibility of decoupling two-phase commit. (This is *not* a variant proposed by the authors of [5].)

2.2 Rewriting Logic and Maude

Maude [9] is an expressive rewriting-logic-based formal specification language and high-performance simulation and model checking tool for concurrent systems. Maude specifications are executable, and the tool provides a variety of formal analysis methods, including simulation, reachability analysis, and linear temporal logic (LTL) model checking.

Specification. A Maude module specifies a *rewrite theory* $(\Sigma, E \cup A, R)$, where:

– Σ is an algebraic *signature*; that is, a set of *sorts*, *subsorts*, and *function symbols.*
– $(\Sigma, E \cup A)$ is a *membership equational logic theory* [9], with E a set of possibly conditional equations and membership axioms, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type.
– R is a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*, specifying the system's local transitions.

We briefly summarize the syntax of Maude and refer to [9] for more details. Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s. They can have user-definable syntax, with underbars '`_`' marking the argument positions. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`, or can be introduced on the fly, in which case they have the form *var*`:`*sort*. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` ("otherwise") attribute can be applied to a subterm $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.

A *class* declaration `class` C `|` att_1 `:` s_1 `, ... ,` att_n `:` s_n declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object instance* of class C is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. The state is a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages. The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  m(O,w)
          < O : C | a1 : x, a2 : O', a3 : z >
        =>
          < O : C | a1 : x + w, a2 : O', a3 : z >
          m'(O',x) .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of the object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated. Attributes whose values do not change and do not affect the next state, such as `a3` and the righthand side occurrence of `a2`, need not be mentioned in a rule.

Reachability Analysis. Maude's *search* command

(search [n] t_0 =>* *pattern* such that *condition* .)

uses a breadth-first strategy to search for at most n states that are reachable from the initial state t_0 in zero or more steps, match the search *pattern*, and satisfy the search *condition*. To search for *final* states (i.e., states that cannot be further rewritten) matching *pattern* and satisfying *condition*, the arrow `=>!` is used instead of `=>*`.

3 Modeling RAMP Transaction Algorithms in Maude

This section presents formal models of RAMP and its variants. We describe the specification of RAMP-Fast in detail, and only show the main differences for the other algorithms. Since we focus on analyzing the correctness of RAMP, and not its performance, our model is an *untimed* one, so that all possible interleavings are analyzed by Maude model checking. The executable Maude specifications are available at `https://sites.google.com/site/siliunobi/sac16-ramp`.

3.1 Data Types, Objects, and Messages

We formalize (the different versions of) RAMP in an object-oriented style, where the state consists of a number of `Partition` objects, each modeling a partition of the database, a number of `Client` objects modeling clients that issue transactions, and a number of messages traveling between the objects.

We assume that transactions do not have conditional reads and that a transaction does not write the same item multiple times; read-only and write-only transactions can then largely be seen as *sets* of operations. As in [5], we assume that data are not replicated.

Data Types. A *version* is a timestamped version of a data item and is modeled as a 4-tuple `version(`*item*`,`*value*`,`*timestamp*`,`*metadata*`)` consisting of the data item, its value, and the version's timestamp and metadata. A timestamp is modeled as a pair `timestamp(`*id*`,`*sqn*`)` consisting of a client's identifier *id* and a sequence number *sqn* that together uniquely identify a writing transaction. Metadata (which in RAMP-Fast are write sets) are modeled as a set of data items, denoting, for each item, the other items that are written in the same transaction. For example, if a write-only transaction has writes on items x, y, and z, then versions of x have as metadata the write set $\{y, z\}$.

```
sorts Item Value Timestamp Metadata Version Latest .
subsort Set{Item} < Metadata .

op timestamp : Nat Nat -> Timestamp .   --- client ID, sequence number
op version : Item Value Timestamp Metadata -> Version [ctor] .
```

A transaction (request) is a set of read operations `read(`*id*`,` *item*`)` and write operations `write(`*id*`,` *item*`,` *value*`)`, where *id* denotes the identity of the operation. The sort `Transactions` denotes lists of transactions.

```
sort Operation .
op read : Nat Item -> Operation .
op write : Nat Item Value -> Operation .

pr SET{Operation} * (sort Set{Operation} to Transaction) .
pr LIST{Transaction} * (sort List{Transaction} to Transactions) .
```

Objects and Messages. A *client* issues transactions and collects responses for analysis purposes. To avoid cluttering the specification, we assume without loss of generality that a client issues transactions sequentially: a transaction from a client is issued when its preceding transaction has been committed. Concurrent transactions can be modeled by multiple transactions issued by different clients.

A client is modeled as an object instance of the class `Client`, and has the following attributes: a list of transactions it wants to issue (`transac`); the sequence number that together with the client identifier determine timestamps (`sqn`); a flag indicating whether a transaction has been committed (`trflag`); several sets of operations buffering intermediate information (`pendingOps`, denoting pending reads/writes, `pendingPrep`, denoting pending writes in the *prepare* phase, `1stGets`, denoting pending first-round reads); a mapping from each item to its latest committed timestamp from a client's perspective (`latest`); and the returned value and operations for each transaction (`result`):

```
class Client | transac : Transactions, sqn : Int,        trflag : Bool,
               pendingOps : OpsInfo,   pendingPrep : Set{Nat},
               1stGets : Set{Nat},     latest : Latest,  result : Results .
```

A *partition* stores a set of versions for a number of data items. We model a partition as an object of class `Partition` with attributes: `versions`, denoting a set of versions for each data item in the partition, and `latestCommit`, denoting the timestamp of the last commit of each item:

```
pr SET{Version} * (sort Set{Version} to Versions) .
pr MAP{Item,Timestamp} * (sort Map{Item,Timestamp} to Latest) .

class Partition | versions : Versions, latestCommit : Latest .
```

For example, the state fragment

```
< c1 : Client |
   transac : (read(3,x),read(4,y)), sqn : 3, trflag : true,
   pendingOps : {1,2}, pendingPrep : {2}, 1stGets : empty,
   latest : empty, result : nil >
< p1 : Partition |
   versions : version(x,v,timestamp(c2,3),y),
              version(y,w,timestamp(c1,1),x),
   latestCommit : (x |-> timestamp(c2,3) ,
                   y |-> timestamp(c1,1)) >
```

models a setting where partition `p1` holds a version for each of the items `x` and `y`, with respective values `v` and `w`, and associated timestamps `timestamp(c2,3)` and `timestamp(c1,1)`. Client `c1` intends to issue a transaction of two reads, reading items `x` and `y`; `trflag` is `true`, meaning that one of `c1`'s transactions is being executed, and `pendingOps` has two pending operations `1` and `2`. We can tell from `pendingPrep` that the ongoing transaction is a write-only transaction with a write `1` that has already committed. The remaining three attributes are `empty` or `nil`, since they are only updated when a read-only transaction is issued.

Messages travel between clients and partitions, and have the form `msg` *msg-Content* `from` *sender* `to` *receiver*, where the message content *msgContent* is either `prepare(`*id*`,` *version*`)` (placing a timestamped write (or *version*) on its partition), `commit(`*id*`,`*timestamp*`)` (marking versions as committed using the *timestamp* generated by a client), `prepared(`*id*`)` (reply to `prepare` for write *id*), `committed(`*id*`)` (reply to `commit`), `get(`*id*`,`*item*`,`*timestamp*`)`, (fetching the highest-timestamped committed version or any missing version for an *item* by *timestamp*) `1st-response(`*id*`,`*version*`)` (returned *version* for first-round `get` for read *id*), or `2nd-response(`*id*`,`*version*`)` (returned version for second-round `get` for read *id*), where *id* denotes the operation's identifier.

3.2 Formalizing RAMP-Fast

This section formalizes the dynamic behaviors of RAMP-Fast using rewrite rules. We also refer to the corresponding lines of code in the description in Fig. 1.

Starting a New Transaction (Lines 14–19 for writes and lines 22–26 for reads). Whenever a client is not executing a transaction (the attribute `trflag` has the value `false` and the buffers storing intermediate information are empty) and there is a pending transaction in the client's transaction list `transac` (the pattern[4] `TR TRL` denotes a list with a single transaction `TR` followed by a (possibly empty) list of `TRL` of the other remaining transactions), the client starts issuing the transaction `TR` by sending a `prepare` message for each write in a write-only transaction, or by sending a `get` message for each read in a read-only transaction. The function `genOps` generates these messages.

The client will then wait for intermediate response messages from the partitions by setting `pendingOps` to the operations in `TR`, `pendingPrep` to the writes (`w`) in `TR` if `TR` is a write-only transaction, `1stGets` to the reads (`r`) in `TR` if `TR` is a read-only transaction, and `latest` to default timestamps for each item requested in `TR`'s reads, respectively. Furthermore, to record the final committed versions, the client also adds to `result` the default version `rs(TR)` (with `null` value and timestamp, and `empty` metadata) for each read in `TR`. Finally, the client sets `trflag` to `true` to indicate that it is currently executing a transaction:

```
rl [init-transaction] :
   < O : Client | transac : TR TRL, sqn : SQN, trflag : false,
                  pendingOps : empty, pendingPrep : empty,
                  1stGets : empty, latest : VL, result : RS >
=>
   < O : Client | transac : TRL, trflag : true,
                  pendingOps : rw(TR), pendingPrep : w(TR),
                  1stGets : r(TR), latest : vl(TR,VL),
                  result : RS rs(TR) >
   genOps(TR,SQN,O) .
```

Receiving Prepare Messages (Lines 3–5). When a partition `O` receives a `prepare` message with identifier `ID` from the client `O'` for a version v, it adds v to its local storage `versions` and sends a `prepared` message back to the client:

```
rl [receive-prepare] :
   msg prepare(ID,version(X,V,timestamp(O',SQN),MD))
      from O' to O
   < O : Partition | versions : VS >
=>
   < O : Partition | versions :
                   version(X,V,timestamp(O',SQN),MD),VS >
   msg prepared(ID) from O to O' .
```

Receiving Prepared Messages (Lines 20–21). When a client receives a `prepared` message for the current write `ID` from a partition `O'`, it deletes `ID` from the set `pendingPrep`. If the resulting set is empty, meaning that all `prepared` messages have been received, the client starts committing the transaction using the function `startCommit`, which generates a `commit` message for each write.

[4] We do not show variable declarations, but follow the Maude convention that variables are written in capital letters.

```
crl [receive-prepared] :
    msg prepared(ID) from O’ to O
    < O : Client | pendingPrep : NS, pendingOps : OI, sqn : SQN >
 =>
    < O : Client | pendingPrep : NS’ >
    (if NS’ == empty then startCommit(OI,SQN,O) else none fi)
 if NS’ := delete(ID,NS) .
```

Receiving Commit Messages (Lines 6–8). When a partition receives a `commit` message with timestamp `timestamp(O’,SQN)`, it invokes the function `cmt` to update the latest commit timestamp in the set `latestCommit` with the fresher timestamp of the incoming one and the local one, provided those two can be matched; it then notifies the client to commit the write:

```
rl [receive-commit] :
   msg commit(ID,timestamp(O’,SQN)) from O’ to O
   < O : Partition | versions : VS, latestCommit : LC >
=>
   < O : Partition | latestCommit :
                       cmt(LC,VS,timestamp(O’,SQN)) >
   msg committed(ID) from O to O’ .
```

Receiving Committed Message. When a client receives a `committed` message, it removes the committed read/write `ID` from its set `pendingOps`. If all pending reads/writes have been committed (`pendingOps` is empty), the client gets ready to issue its next transaction by setting `trflag` to `false` and increasing the sequence number:

```
rl [receive-committed] :
   msg committed(ID) from O’ to O
   < O : Client | pendingOps : OI >
=>
   < O : Client | pendingOps : remove(ID,OI) > .

rl [commit-transaction] :
   < O : Client | trflag : true, pendingOps : empty,
                  sqn : SQN >
=>
   < O : Client | trflag : false, sqn : SQN + 1 > .
```

Receiving Get Messages (Lines 9–13). When a partition receives a `get` message with identifier `ID`, requested data item `X` and timestamp `TS’` from the client `O’`, it replies with the last committed version of the item. If the incoming `get` is for first round communication (`TS’ == null`), then the partition invokes `vmatch` to return the version of `X` in `VS` with the timestamp matched by that in `LC` for `X`:

```
rl [on-receiving-get] :
   msg get(ID,X,TS') from O' to O
   < O : Partition | versions : VS, latestCommit : LC >
=>
   < O : Partition | >
   (if TS' == null then 1st-response(ID,vmatch(X,VS,LC)) from O to O'
                   else 2nd-response(ID,vmatch(X,VS,TS')) from O to O' fi) .
```

Receiving Response to First-round Get Messages (Lines 25, 27–33). When a client first receives the (last) committed version of the requested item, it removes the transaction read `ID` from the set `1stGets`, and adds the received version into the set `result`. It updates the set `latest` with the received timestamp and set of items from the received version (i.e., its metadata), thus producing a new mapping `VL'` from the item to the timestamp representing the highest-timestamped write for the transaction appearing in the first-round read set. When all responses to the first-round `get` messages have been collected, the client invokes the function `gen2ndGets` to issue a second-round read to fetch the missing version: a `get` message will be sent to the corresponding partition if the received version of an item has a lower timestamp than indicated in `VL'`:

```
crl [on-receiving-1st-response] :
    msg 1st-response(ID,version(X,V1,TS1,MD1)) from O' to O
    < O : Client | 1stGets : NS, result : RS, latest : VL >
 =>
    < O : Client | 1stGets : NS', result : RS', latest : VL' >
    (if NS' == empty then gen2ndGets(ID,VL',RS',O) else none fi)
 if NS' := delete(ID,NS) /\ VL' := lat(VL,MD1,TS1) /\
    RS'  := addVersion(ID,version(X,V1,TS1,MD1),RS) .
```

Receiving Response to Second-round Get Messages (Lines 32–33). When a client receives the response to a second-round `get` message, it adds the fetched version into `result`:

```
rl [on-receiving-2nd-response] :
   msg 2nd-response(ID,version(X,V,TS,MD)) from O' to O
   < O : Client | pendingOps : OI, result : R >
=>
   < O : Client | pendingOps : remove(ID,OI),
                  result : addVersion(ID,v(X,V,TS,MD),RS) > .
```

3.3 Formalizing RAMP-Small

Instead of attaching the entire write set to each write, RAMP-Small clients only store the transaction timestamp. This correspond to changing the rule `init-transaction` by letting `genOps` not instantiate metadata for each outgoing write, but instead leave it as an `empty` set. Apart from that, only the following two rules in RAMP-Fast must be modified to define RAMP-Small.

Receiving Response to First-round Get Messages. When a client has fetched the (highest-timestamped) committed timestamp for the requested item in the received version, it proceeds in the similar way as in RAMP-Fast except that it will not update `result` since RAMP-Small always requires two RTT for reads. Each outgoing `get` message generated by `gen2ndGets` includes the entire set of timestamps received via first-round `get` messages:

```
crl [on-receiving-1st-response] :
    msg 1st-response(ID,version(X,V1,TS1,MD1)) from O' to O
    < O : Client | 1stGets : NS, latest : VL, result : RS >
 =>
    < O : Client | 1stGets : NS', latest : VL' >
    (if NS' == empty then gen2ndGets(ID,VL',RS,O) else none fi)
 if NS' := delete(ID,NS) /\ VL' := insert(X,TS1,VL) .
```

Receiving Get Messages. When a partition receives a `get` message for the first time, it proceeds in the same way as in RAMP-Fast; however, when the second `get` message arrives that contains *the entire set of timestamps* for the requested item, it returns the highest-timestamped version (determined by `maxts`) of that item that also exists in the received set of timestamps (determined by `tsmatch`). Note that the incoming `get` message includes a set of timestamps `TSS`:

```
rl [on-receiving-get-small] :
   msg get(ID,X,TSS) from O' to O
   < O : Partition | versions : VS, latestCommit : LC >
=>
   < O : Partition | >
   (if TSS == empty
    then 1st-response(ID,vmatch(X,VS,LC)) from O to O'
    else 2nd-response(ID,vmatch(X,VS,maxts(tsmatch(X,VS,TSS)))) from O to O' fi) .
```

3.4 Formalizing Variants of RAMP

RAMP Without 2PC. RAMP uses two-phase commit (2PC) to ensure that all partitions successfully execute a transaction or that none do. Specifically, writes start to commit only after *all of them are prepared* on the partitions. This results in high latency, even "resource leak" on partitions during failures [5], since one blocked write will cause the transaction to stall. We can "decouple" 2PC from RAMP by changing the rule `on-receiving-prepare` to the following rule, in which a client simply removes the write from the pending set and commits it on the partition, instead of waiting for all `prepared` messages to arrive:

```
rl [on-receiving-prepared-decouple-2pc] :
   msg prepared(ID) from O' to O
   < O : Client | pendingPrep : NS, sqn : SQN >
=>
   < O : Client | pendingPrep : delete(ID,NS) >
   msg commit(ID,timestamp(O,SQN)) from O to O' .
```

RAMP with Faster Commit. This optimization is described as follows in [5]:

> If a server returns a version in response to a GET request and the version's timestamp is greater than the highest committed version of that item, then transaction writing the version has committed on at least one partition. In this case, the server can mark the version as committed.

That is, a partition can mark as committed the version (by sending a `commit` message to the client `O'`) that has a fresher timestamp than the highest committed version of the requested item (indicated by `LC[X] < TS'`). We model this proposed optimization of RAMP-Fast by replacing the rule `on-receiving-get` with the following rule `on-receiving-get-faster-commit`. The other rules are unchanged.

```
rl [on-receiving-get-faster-commit] :
   msg  get(ID,X,TS') from O' to O
   < O : Partition | versions : VS, latestCommit : LC >
=>
   if TS' == null
   then < O : Partition | >
        msg 1st-response(ID,vmatch(X,VS,LC)) from O to O'
   else if LC[X] < TS'
        then < O : Partition | latestCommit : insert(X,TS',LC) >
             (msg committed(ID) from O to O')
             (msg 2nd-response(ID,vmatch(X,VS,TS')) from O to O')
        else < O : Partition | >
             msg 2nd-response(ID,vmatch(X,VS,TS')) from O to O' fi fi .
```

RAMP with One-Phase Writes (1PW). An optimization proposed in [5] in case a user does not wish to read her writes is to allow a client to

> return after issuing its PREPARE round. The client can subsequently execute the COMMIT phase asynchronously.

Specifically, after collecting all `prepare` messages (indicated by `NS' == empty`), a client starts to execute the commit phase by invoking `startCommit` to generate `commit` messages, and kicks off the next transaction by sending a message `executeTransaction` to itself to force the client to start executing the next transaction.

```
crl [on-receiving-prepared-1pw] :
   msg prepared(ID) from O' to O
   < O : Client | pendingPrep : NS, pendingOps : OI, sqn : SQN >
=>
   < O : Client | pendingPrep : NS' >
   (if NS' == empty
      then (startCommit(OI,SQN,O))
           (executeTransaction O)
      else none fi)
if NS' := delete(ID,NS) .
```

4 Formal Analysis of RAMP and Its Variants

In this section we use reachability analysis to analyze whether RAMP and its variants satisfy the properties in [5]. Explicit-state model checkers like Maude are typically quite expressive but only analyze the system from a *single* initial configuration. To increase coverage, we would like to model check RAMP for *all possible* configurations up to certain bounds, for example k operations and j clients. Despite the wealth of Maude applications, we are not aware of any work doing such comprehensive model checking in Maude. Section 4.1 therefore presents a general technique in Maude for model checking a system from a range of different initial configurations. Section 4.2 formalizes the desired properties that RAMP transactions should satisfy and analyzes them for *all* configurations with four operations and two clients, as well as for a number of configurations with six operations.

4.1 Model Checking Many Initial States

The idea behind model checking many initial configurations is to introduce a new operator `init`, have a *one-step* rewrite `init`(*parameters*) $\longrightarrow t_0$ for *any* possible initial configuration t_0, and start the analysis from `init`(*parameters*). However, there are two things to take into account:

1. The analysis must take into account the additional rewrite step before the actual analysis of all behaviors from a concrete initial state begins. A search command (`search` t_0 `=>*` *pattern* `.`), which searches for states reachable in zero or more steps from t_0, must be replaced by (`init`(*parameters*) `=>+` *pattern* `.`), which searches for states reachable in *one* or more steps. Likewise, in temporal logic model checking, an LTL formula φ should be replaced by the formula $\bigcirc\,\varphi$ (which means that φ holds in the next state).
2. The property to analyze may depend on the particular initial state. When generating multiple initial states and model checking them in one command, it might be necessary to record (parts of) the initial state, and carry this record in the state throughout the analysis.

To generate all possible initial states, we declare a new sort denoting *sets* of configurations:

```
sort ConfigSet .  subsort Configuration < ConfigSet .
op empty : -> ConfigSet .
op _;_ : ConfigSet ConfigSet -> ConfigSet [assoc comm id: empty] .
```

We assume that there is a function

```
op initAux : s1 ... sn -> ConfigSet .
```

such that `initAux(`*params*`,`*params′*`)` generates all possible initial states, and add the following rewrite rule to our model:

```
var C : Configuration .  var CS : ConfigSet .
crl [init] : init(params) => C
    if C ; CS := initAux(params,params') .
```

If the state needs to carry a record $f(t_0)$ of the initial state t_0, we use the following rule instead:

```
crl [init] :
    init(params) => C  < r : Record | record : f(C) >
    if C ; CS := initAux(params,params') .
```

where `Record` is a new class which stores selected data from the chosen initial state throughout the computations.

For RAMP, `init` has the parameters: *#ops*, the total number of (read or write) operations to be performed; *#clients*, the the number of clients; and *dataItems*, the set of data items in the system, with each partition storing one item. We also store in the `record` object the list of transactions to be issued by each client.

```
  --- Generate a new client
  rl initAux(OPS, s CLS, (I,ITEMS), none, C)
   =>
     initAux(OPS, CLS, (I,ITEMS), < s CLS : Client | transac : nil, sqn : 1,
       trflag : false, pendingOps : empty, pendingPrep : empty, 1stGets : empty,
       latest : empty, result : nil >, C) .

  --- Add an op to the current transaction, if more ops needed

  --- Check for read-only transaction
  op noWriteIn : Transaction -> Bool .
  eq noWriteIn((write(ID,X,V),TR)) = false .
  eq noWriteIn(TR) = true [owise] .

  --- Check for write-only transaction
  op noReadIn : Transaction -> Bool .
  eq noReadIn((read(ID,X),TR)) = false .
```

```
 eq noReadIn(TR) = true [owise] .

 --- Add a write to a write-only transaction
crl initAux(s OPS, CLS, (I,ITEMS), < O : Client | transac : TRS TR >, C)
  =>
    initAux(OPS, CLS, (I,ITEMS), < O : Client |
      transac : TRS (TR,write(s OPS,I,s OPS)) >, C)
  if noReadIn(TR)  /\  noSameItem(TR,I) .

 --- Add a read to a read-only transaction
crl initAux(s OPS, CLS, (I,ITEMS), < O : Client |
      transac : TRS TR >, C)
  =>
    initAux(OPS, CLS, (I,ITEMS), < O : Client |
      transac : TRS (TR,read(s OPS,I)) >, C)
  if noWriteIn(TR) .

 --- Guarantee there are no writes on the same item
 op noSameItem : Transaction Item -> Bool .
 eq noSameItem((write(ID,I,V),TR),I) = false .
 eq noSameItem((read(ID,I),TR),I) = false .
 eq noSameItem(TR,I) = true [owise] .

 --- Start a new read-only transaction
 rl initAux(s OPS, CLS, (I,ITEMS), < O : Client | transac : TRS >, C)
  =>
    initAux(OPS, CLS, (I,ITEMS), < O : Client | transac :
      TRS (read(s OPS,I)) >, C) .

 --- Start a new write-only transaction
 rl initAux(s OPS, CLS, (I,ITEMS), < O : Client | transac : TRS >, C)
  =>
    initAux(OPS, CLS, (I,ITEMS), < O : Client | transac :
      TRS (write(s OPS,I,s OPS)) >, C) .

 --- Done with the client, ready for generating a new client
 rl initAux(OPS, CLS, (I,ITEMS), < O : Client | transac : TRS >,
      < R : Record | record : REC > C)
  =>
    initAux(OPS, CLS, (I,ITEMS), none, < O : Client | transac : TRS >
      < R : Record | record : insert(O,TRS,REC) > C) .

 --- Generating partitions
 rl initAux(0,0,(I,ITEMS),none,C)
  =>
    initAux(0,0,ITEMS,none,< I : Partition |
      versions : (v(I,null,timestamp(0,0),empty)),
      latestCommit : (I |-> timestamp(0,0)) > C) .
```

One of 2764 initial states defined by `init(4,2,(x,y))` is

```
< 1 : Client | transac : (read(1,x) , read(2,x)) , sqn : 1,
    1stGets : empty, latest : empty, pendingOps : empty,
    pendingPrep : empty, result : nil, trflag : false >
< 2 : Client | transac : (write(4,y,4)  write(3,x,3)) , ... >
< x : Partition | latestCommit : x |-> timestamp(0,0),
   versions : version(x,null,timestamp(0,0), empty)>
< y : Partition | latestCommit : y |-> timestamp(0,0),
  versions : version(y,null,timestamp(0,0),empty)>
< 100 : Record | record : 1 |->(read(1,x) , read(2,x)),
                          2 |-> write(4,y,4) write(3,x,3) >
```

where client 1 has one transaction with two reads, and client 2 has two transactions, each having one write operation.

4.2 Analyzing the Correctness Properties

This section formalizes the correctness requirements of RAMP as reachability properties and analyzes them using Maude.

During the execution of a transaction with multiple reads, one read will be committed before the other, leading to intermediate states where key properties do not hold. Since RAMP is terminating if each client issues a finite number of transactions, we therefore analyze most properties on final states, when all transactions are committed.

We have performed our analysis from `init(4,2,(x,y))`, which means that we consider all possible initial configurations with a total of 4 operations, 2 clients, and 2 data items. There are 2764 such initial configurations. Each analysis command took about 20 seconds to execute on a 2.9 GHz Intel 4-Core i7-3520M CPU with 3.7 GB memory.

For 6 total operations, we analyze the different variations of RAMP for all possible non-trivial scenarios with 6 total operations, 2 clients, 2 data items, and 2 partitions, each storing one item. RAMP only considers read-only and write-only transactions; furthermore, as already explained, we ignore pointless transactions in which the same item is read twice or written twice. Likewise, we ignore single-operation transactions: neither a single read transaction nor a single write transaction can lead to fractured reads. Finally, after we omit symmetric scenarios (e.g., if client c_1 has transaction(s) tl_1 and client c_2 wants to execute the transaction(s) tl_2, then there is no need to consider the symmetric scenario where c_1 executes tl_2 and c_2 executes tl_1), we are left with 6 scenarios to analyze:

$$\begin{array}{lll}
c_1 : [\mathit{read}(x), \mathit{read}(y)] & c_2 : [\mathit{write}(x), \mathit{write}(y)] & [\mathit{write}(x), \mathit{write}(y)] \\
c_1 : [\mathit{write}(x), \mathit{write}(y)] & c_2 : [\mathit{read}(x), \mathit{read}(y)] & [\mathit{write}(x), \mathit{write}(y)] \\
c_1 : [\mathit{write}(x), \mathit{write}(y)] & c_2 : [\mathit{write}(x), \mathit{write}(y)] & [\mathit{read}(x), \mathit{read}(y)] \\
c_1 : [\mathit{write}(x), \mathit{write}(y)] & c_2 : [\mathit{read}(x), \mathit{read}(y)] & [\mathit{read}(x), \mathit{read}(y)] \\
c_1 : [\mathit{read}(x), \mathit{read}(y)] & c_2 : [\mathit{write}(x), \mathit{write}(y)] & [\mathit{read}(x), \mathit{read}(y)] \\
c_1 : [\mathit{read}(x), \mathit{read}(y)] & c_2 : [\mathit{read}(x), \mathit{read}(y)] & [\mathit{write}(x), \mathit{write}(y)]
\end{array}$$

Read Atomic Isolation. The main correctness property, *read atomic isolation*, that RAMP should satisfy is defined as follows in [5]:

> "A system provides *Read Atomic* isolation if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data."

where *fractured reads* are described as follows:

> "A transaction T_j exhibits *fractured reads* if transaction T_i writes version x_m and y_n (in any order, with x possibly but not necessarily equal to y), T_j reads version x_m and version y_k, and $k < n$."

We analyze this property by searching for a reachable *final* state where the property does *not* hold; i.e., a state where one of the clients' committed results, `RES1` or `RES2`, has a fractured read:

```
(search [1] init(4,2,(x , y)) =>!
    C:Configuration
    < r:Oid : Record | record : (O1:Oid |-> TL1:TrList)
                                (O2:Oid |-> TL2:TrList) >
    < O1:Oid : Client | result : RES1:Result >
    < O2:Oid : Client | result : RES2:Result >
  such that fracRead(RES1:Result, TL1:TrList, TL2:TrList)
         or fracRead(RES2:Result, TL2:TrList, TL1:TrList) .)
```

where the function `fracRead` checks whether there are fractured reads on each client's committed result, based on the initial transactions of all clients:

```
op fracRead : Results Transactions Transactions -> Bool .

  --- handle a client's own write-only transactions
ceq fracRead((RS1 (ID1 |-> v(X1,V,TS1,MD1),ID2 |-> v(X2,V2,TS2,MD2),R) RS2),
             (TRL (write(ID,X1,V),write(ID',X2,V'),TR) TRL'
             (read(ID1,X1),read(ID2,X2),TR') TRL''),TRL2) = true
    if V2 =/= V'  /\ X1 =/= X2 .

  --- handle another client's write-only transactions
ceq fracRead((RS1 (ID1 |-> v(X1,V,TS1,MD1),ID2 |-> v(X2,V2,TS2,MD2),R) RS2),
              TRL1,(TRL (write(ID,X1,V),write(ID',X2,V'),TR) TRL')) = true
     if V2 =/= V'  /\ X1 =/= X2 .

eq fracRead(RS1,TRL,TRL') = false [owise] .
```

Our analysis results are consistent with the analytic and predicted results in [5]: RAMP *without* two-phase commit does not satisfy read atomicity; all other versions of RAMP do. Model checking the Scenario $c_1 : [\mathit{read}(x), \mathit{read}(y)] \quad c_2 : [\mathit{write}(x), \mathit{write}(y)] \quad [\mathit{read}(x), \mathit{read}(y)]$ yields the following counterexample for

RAMP without two-phase commit: The read-only transaction $c_1 : [\mathit{read}(x), \mathit{read}(y)]$ has committed and returned, before the partition holding item x has been prepared by the write-only transaction (i.e., before the associated `prepare` message has reached that partition), but the other `prepare` message even arrives at the partition holding item y. In that case, the read-only transaction will return the result where only one write $\mathit{write}(y)$ of the write-only transaction is visible, which violates RA.

Companions Present. Another property that is verified in [5] is the following invariant:

> "If a version x_i is referenced by *lastCommit* (that is, $\mathit{lastCommit}[x] = i$), then each of x_i's sibling versions[5] are present in *versions* on their respective partitions."

This invariant can be analyzed by searching for a state that does not satisfy the property:

```
(search [1] init(4,2,(x , y)) =>+ C:Configuration
    < r:Oid : Record | record : (O1:Oid |-> TL1:TrList)
                                (O2:Oid |-> TL2:TrList) >
    such that not comp-present(C:Configuration,TL1:TrList,TL2:TrList) .)
```

where `comp-present` holds if and only if for each committed item, the sibling version of its associated version is present in the other partition:

```
op comp-present : Configuration Transactions Transactions -> Bool .

ceq comp-present(< O1 : Partition | versions : (v(X1,V1,TS1,MD1),VS1),
                                    latestCommit : (X1 |-> TS1) >
                 < O2 : Partition | versions : (v(X2,V2,TS1,MD2),VS2) >
      REST, TRL,TRL') = false
    if not sib(V1,V2,TRL,TRL') .
eq comp-present(C,TRL,TRL') = true [owise] .

op sib : Value Value Transactions Transactions -> Bool .
eq sib(V1,V2,(TRL (write(ID,X1,V1),write(ID',X2,V2),TR) TRL'),TRL2) = true .
eq sib(V1,V2,TRL1,(TRL (write(ID,X1,V1),write(ID',X2,V2),TR) TRL')) = true .
eq sib(null,null,TRL1,TRL2) = true .
eq sib(V1,V2,TRL1,TRL2) = false [owise] .
```

where the function `sib` is defined to check whether versions of the same timestamp (by matching `TS1` in each version on different partitions, i.e., versions by the writes in the same transaction) are siblings. This can be done by checking if `V1` and `V2` match the writes in each write-only transaction in `TRL` or `TRL'`.

Our analysis shows that all versions, except the ones without 2PC, satisfy this property.

[5] We call the set of versions produced by a (write-only) transaction *sibling versions.*

Synchronization Independence. Synchronization Independence. This property is described as follows in [5]:

> "Synchronization Independence ensures that one client's transactions cannot cause another client's to block and that, if a client can contact the partition responsible for each item in its transaction, the transaction will eventually commit (or abort of its own volition)."

We check the stronger property that all transactions are eventually always committed on all clients' sides: since we assume that there are no uncommitted or aborted transactions, a client-side method can return only if the corresponding partition-side methods have returned. The following function `syn-indep` holds in a state if and only if there are no pending transactions (`transac` is `nil` and all buffers are `empty`) and no ongoing transactions (`trflag` is `false`):

```
op syn-indep : Configuration -> Bool .
eq syn-indep(
   < O1 : Client | transac : nil, trflag : false,
                   pendingOps : empty, pendingPrep : empty,
                   1stGets : empty >
   < O2 : Client | transac : nil, trflag : false,
                   pendingOps : empty, pendingPrep : empty,
                   1stGets : empty > REST)
   = true .
eq syn-indep(SYSTEM) = false [owise] .
```

Again, we analyze the property by searching for a *final* state which does *not* satisfy `syn-indep`. The property is satisfied by all versions of RAMP in our analysis.

Partition Independence. Partition independence means that a client never has to contact a partition that its transaction does not access. This follows directly from the fact that the function `genOps` only generates messages to the partitions that store the data items of the transaction. Hence this property holds trivially in our model.

Read Your Writes. This property says that a client's writes are visible to her subsequent reads. It can be analyzed by searching for a final state in which a client's recorded reads contain a read that read the client's write which was *not* the write immediately preceding the read:

```
(search [1] init(4,2,(x , y)) =>!
   C:Configuration
   < r:Oid : Record | record : (C1:Oid |-> TL1:TrList)
                               (C2:Oid |-> TL2:TrList) >
```

```
  < C1:Oid : Client | result : RES1:Result >
  < C2:Oid : Client | result : RES2:Result >
      such that tooOldRead(RES1:Result,TL1:TrList)
             or tooOldRead(RES2:Result,TL2:TrList) .)
```

A client has read a too old value if either it reads a *null* value, or a value it has written earlier, *and* it has a (later) write transaction writing the item.

The function `tooOldRead` returns `true` if a read `ID` has fetched either: (i) a value `V1` of an earlier write `ID1` although the client had a later write `ID2` preceding the read `ID` (first equation); or (ii) the initial value `null` even though the client had an earlier write `ID1` that has not written `null` (second equation). In the first equation below, the `result` value contains a read `ID |-> version(X,V1,TS,MD)`, and the transactions that the client should execute match the pattern

```
  TRL   (OPS1, write(ID1,X,V1), OPS1')
  TRL'  (OPS2, write(ID2,X,V2), OPS2')
  TRL'' (OPS, read(ID,X), OPS')  TRL'''
```

where the variables `TRL` denote *lists of transactions*, and the variables `OPS` denote parts of a single transaction (sets of operations). In particular, in the client's original transaction list, there are two transactions `(OPS1, write(ID1,X,V1), OPS1')` and `(OPS2, write(ID2,X,V2), OPS2')` both writing `X` before the transaction `(OPS, read(ID,X), OPS')` reads the `X`. It is clear that if `read(ID,X)` reads the value written by `write(ID1,X,V1)`, then the client has not read its own (latest) writes. The second equation takes care of the case when the read operation reads `null` even though it itself had an earlier transaction writing `X`. If none of these equations apply, and the `owise` equation returns `false`:

```
op tooOldRead : Results TrList -> Bool .

eq tooOldRead((RS  (ID |-> version(X,V1,TS,MD) ,R)  RS2),
       (TRL  (OPS1, write(ID1,X,V1), OPS1')
        TRL'  (OPS2, write(ID2,X,V2), OPS2')
        TRL'' (OPS, read(ID,X), OPS')   TRL''') = true .

eq tooOldRead((RS1 (ID |-> version(X,null,TS,MD) ,R) RS2),
       (TRL  (OPS1, write(ID1,X,V1), OPS2)
        TRL'  (OPS, read(ID,X), OPS')  TRL'')) = true .

eq tooOldRead(RS, TRL) = false [owise] .
```

Our analysis shows that only RAMP with one-phase writes does not satisfy the property. The counterexample obtained by analyzing the initial state where a client has transactions $[\mathit{write}(x), \mathit{write}(y)]\quad[\mathit{read}(x), \mathit{read}(y)]$ shows that the read operations both return `null`. The reason is that one-phase writes do not forbid the client to start the subsequent read-only transaction before its last write-only transaction has committed (by `commit` messages) on the partitions.

Summary. We have analyzed all 4 key properties of our 7 versions of RAMP. Our results agree with the proved and conjectured results in [5]: All versions satisfy the properties, except that

– RAMP without 2PC does not satisfy read atomicity, "companions present", and read-your-writes; and
– RAMP with one-phase writes does not satisfy read-your-writes.

5 Related Work

Despite the importance of distributed data stores for cloud computing, we are not aware of much work on formalizing and verifying such systems. The recent paper [17] describes experiences at Amazon Web Services using TLA+ to formalize and model check Amazon's scalable high-performance replicated NoSQL data store DynamoDB. The authors report that TLA+ model checking "[found] bugs in system designs that cannot be found through any other techniques we know of" and that "formal methods are routinely applied to the design of complex real-world software, including public cloud services," at Amazon. Maude and Real-Time Maude were used in [10,11] to define a formal model of Google's widely-replicated data store Megastore, which ensures serializability for certain classes of transactions, and to develop an extension of Megastore. These models were simulated for QoS estimation and model checked for functional correctness. In a similar vein, [15] presents a formal model of the popular distributed key-value store Cassandra, and formally analyzes different consistency properties using Maude. The authors in [6] reduce the problem of verifying *eventual consistency* of optimistic data replication systems in large-scale networks to reachability and model checking problems. The main difference between all the above-mentioned work and this paper is that we formalize and analyze a different consistency property (read atomic isolation) than the various eventual consistency and serializability properties analyzed in the related work, and that we do so for several alternative designs of RAMP, a system that, to the best of our knowledge, has not been formally modeled and analyzed before.

6 Concluding Remarks

Today's distributed storage systems are seeing a convergence of traditional "strong consistency" databases and the new generation of "fast but eventually consistent" NoSQL databases. In this paper, we have analyzed the most promising bridge system, called RAMP [5], which seeks to provide strong consistency for client transactions while still offering fast performance. While the original RAMP paper included hand proofs for only the basic RAMP algorithms, we adopted a model checking approach where we: (1) first developed fully-executable formal models of many RAMP variants, extensions, and optimizations using Maude; (2) formally analyzed these models to confirm the original correctness properties (within some constraints); (3) used our models to evaluate the correctness properties of RAMP's extensions and optimizations (something the original RAMP paper did not do); and (4) used our models to evaluate the critical dependence of RAMP's correctness properties on its building blocks.

Based on our experience, we believe that the formal modeling and model checking approach can be powerful if used by developers (of cloud storage systems) before (or even alongside) their implementation. This approach allows developers to go beyond mere hand proofs or implementation-based simulations. Once the formal model is written, correctness properties can be automatically analyzed. We could also use formal modeling with probabilistic rewrite rules and statistical model checking to evaluate the effect of further optimizations [14].

References

1. Basho Riak, `http://basho.com/riak/`
2. Cassandra, `http://cassandra.apache.org`
3. Market research media, NoSQL market forecast 2013-2018, `http://www.marketresearchmedia.com/?p=568`
4. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. PVLDB 7(3) (2013)
5. Bailis, P., Fekete, A., Hellerstein, J.M., Ghodsi, A., Stoica, I.: Scalable atomic visibility with RAMP transactions. In: Proc. SIGMOD'14. ACM (2014)
6. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Proc. POPL'14. ACM (2014)
7. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC. p. 7 (2000)
8. Chang, F., et al.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2) (2008)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
10. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Specification, Algebra, and Software. LNCS, vol. 8373. Springer (2014)
11. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: SEFM. LNCS, vol. 8702. Springer (2014)
12. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media (2010)
13. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
14. Liu, S., Nguyen, S., Ganhotra, J., Rahman, M.R., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in nosql key-value stores. In: QEST 2015. Lecture Notes in Computer Science, vol. 9259, pp. 228–243. Springer (2015)
15. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: Proc. ICFEM'14. LNCS, vol. 8829. Springer (2014)
16. Lynch, N., Gilbert, S.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51–59 (June 2002)

17. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Communications of the ACM 58(4), 66–73 (2015)