ROLA: A New Distributed Transaction Protocol and Its Formal Analysis

Si Liu[1](✉), Peter Csaba Ölveczky[2], Keshav Santhanam[1], Qi Wang[1], Indranil Gupta[1], and José Meseguer[1]

[1] University of Illinois, Urbana-Champaign, USA
siliu3@illinois.edu
[2] University of Oslo, Oslo, Norway

Abstract. Designers of distributed database systems face the choice between stronger consistency guarantees and better performance. A number of applications only require *read atomicity* (RA) and *prevention of lost updates* (PLU). Existing distributed database systems that meet these requirements also provide additional stronger consistency guarantees (such as *causal consistency*), and therefore incur lower performance. In this paper we define a new distributed transaction protocol, ROLA, that targets applications where only RA and PLU are needed. We formally model ROLA in Maude. We then perform model checking to analyze both the correctness and the performance of ROLA. For *correctness*, we use standard model checking to analyze ROLA's satisfaction of RA and PLU. To analyze *performance* we: (a) use statistical model checking to analyze key performance properties; and (b) compare these performance results with those obtained by analyzing in Maude the well-known protocol Walter. Our results show that ROLA outperforms Walter.

1 Introduction

Distributed transaction protocols are complex distributed systems whose design is quite challenging because: (i) validating correctness is very hard to achieve by testing alone; (ii) the high performance requirements needed in many applications are hard to measure before implementation; and (iii) there is an unavoidable tension between the *degree of consistency* needed for the intended applications and the *high performance* required of the transaction protocol for such applications: balancing well these two requirements is essential.

In this work, we present our results on how to use formal modeling and analysis as early as possible in the design process to arrive at a mature design of a *new* distributed transaction protocol, called ROLA, meeting specific correctness and performance requirements *before* such a protocol is implemented. In this way, the above-mentioned design challenges (i)–(iii) can be adequately met. We also show how using this formal design approach it is relatively easy to *compare* ROLA with other existing transaction protocols.

A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 77–93, 2018.
https://doi.org/10.1007/978-3-319-89363-1_5

ROLA in a Nutshell. Different applications require negotiating the consistency vs. performance trade-offs in different ways. The key issue is the application's required *degree of consistency*, and how to meet such requirements with *high performance*. Cerone *et al.* [4] survey a *hierarchy of consistency models* for distributed transaction protocols including (in increasing order of strength):

- *read atomicity* (RA): either *all* or *none* of a distributed transaction's updates are visible to another transaction (that is, there are no "fractured reads");
- *causal consistency* (CC): if transaction T_2 is *causally dependent* on transaction T_1, then if another transaction sees the updates by T_2, it must also see the updates of T_1 (e.g., if A posts something on a social media, and C sees B's comment on A's post, then C must also see A's original post);
- *parallel snapshot isolation* (PSI): like CC but without lost updates;
- and so on, all the way up to the well-known *serializability* guarantees.

A key property of transaction protocols is the *prevention of lost updates* (PLU). The weakest consistency model in [4] satisfying both RA and PLU is PSI. However, PSI, and the well-known protocol Walter [20] implementing PSI, also guarantee CC. Cerone *et al.* conjecture that a system guaranteeing RA and PLU *without* guaranteeing CC should be useful, but up to now we are not aware of any such protocol. The point of ROLA is exactly to fill this gap: guaranteeing RA and PLU, but not CC. Two key questions are then: (a) are there *applications* needing high performance where RA plus PLU provide a sufficient degree of consistency? and (b) can a new design meeting RA plus PLU *outperform* existing designs, like Walter, meeting PSI?

Regarding question (a), an example of a transaction that requires RA and PLU but not CC is the "becoming friends" transaction on social media. Bailis *et al.* [3] point out that RA is crucial for this operation: If Edinson and Neymar become friends, then Unai should not see a *fractured read* where Edinson is a friend of Neymar, but Neymar is not a friend of Edinson. An implementation of "becoming friends" must obviously guarantee PLU: the new friendship between Edinson and Neymar should not be lost. Finally, CC could be sacrificed for the sake of performance: Assume that Dani is a friend of Neymar. When Edinson becomes Neymar's friend, he sees that Dani is Neymar's friend, and therefore also becomes friend with Dani. The second friendship therefore causally depends on the first one. However, it does not seem crucial that others are aware of this causality: If Unai sees that Edinson and Dani are friends, then it is not necessary that he knows that (this happened *because*) Edinson and Neymar are friends.

Regarding question (b), Sect. 6 shows that ROLA clearly outperforms Walter in all performance requirements for all read/write transaction rates.

Maude-Based Formal Modeling and Analysis. In rewriting logic [16], distributed systems are specified as *rewrite theories*. Maude [5] is a high-performance language implementing rewriting logic and supporting various model checking analyses. To model time and performance issues, ROLA is specified in Maude as a *probabilistic rewrite theory* [1,5]. ROLA's RA and PLU requirements are then analyzed by standard model checking, where we disregard time issues. To estimate ROLA's performance, and to compare it with that of Walter, we have also specified Walter in Maude, and subject the Maude models of both ROLA and Walter to *statistical model checking* analysis using the PVeStA [2] tool.

Main Contributions include: (1) the design, formal modeling, and model checking analysis of ROLA, a new transaction protocol having useful applications and meeting RA and PLU consistency properties with competitive performance; (2) a detailed performance comparison by statistical model checking between ROLA and the Walter protocol showing that ROLA outperforms Walter in all such comparisons; (3) to the best of our knowledge the first demonstration that, by a suitable use of formal methods, a completely new distributed transaction protocol can be designed and thoroughly analyzed, as well as be compared with other designs, very early on, *before* its implementation.

2 Preliminaries

Read-Atomic Multi-Partition (RAMP) Transactions. To deal with ever-increasing amounts of data, large cloud systems *partition* their data across multiple data centers. However, guaranteeing strong consistency properties for multi-partition transactions leads to high latency. Therefore, trade-offs that combine efficiency with weaker transactional guarantees for such transactions are needed.

In [3], Bailis *et al.* propose an isolation model, *read atomic* isolation, and *Read Atomic Multi-Partition* (RAMP) transactions, that together provide efficient multi-partition operations that guarantee read atomicity (RA).

RAMP uses multi-versioning and attaches metadata to each write. Reads use this metadata to get the correct version. There are three versions of RAMP; in this paper we build on RAMP-Fast. To guarantee that all partitions perform a transaction successfully or that none do, RAMP performs two-phase writes using the two-phase commit protocol (2PC). In the *prepare* phase, each timestamped write is sent to its partition, which adds the write to its local database.[1] In the *commit* phase, each such partition updates an index which contains the highest-timestamped committed version of each item stored at the partition.

RAMP assumes that there is no data *replication*: a data item is only stored at one partition. The timestamps generated by a partition P are unique identifiers but are sequentially increasing only with respect to P. A partition has access to methods GET_ALL(I : set of items) and PUT_ALL(W : set of $\langle$item, value$\rangle$ pairs).

PUT_ALL uses two-phase commit for each w in W. The first phase initiates a *prepare* operation on the partition storing $w.item$, and the second phase completes the commit if each write partition agrees to commit. In the first phase, the client (i.e., the partition executing the transaction) passes a *version* v : $\langle$item, value, $ts_v, md\rangle$ to the partition, where ts_v is a timestamp generated for the transaction and md is metadata containing all other items modified in the same transaction. Upon receiving this version v, the partition adds it to a set *versions*.

[1] RAMP does not consider write-write conflicts, so that writes are always prepared successfully (which is why RAMP does not prevent lost updates).

When a client initiates a GET_ALL operation, then for each $i \in I$ the client will first request the latest version vector stored on the server for i. It will then look at the metadata in the version vector returned by the server, iterating over each item in the metadata set. If it finds an item in the metadata that has a later timestamp than the ts_v in the returned vector, this means the value for i is out of date. The client can then request the RA-consistent version of i.

Rewriting Logic and Maude. In rewriting logic [16] a concurrent system is specified a as *rewrite theory* $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic theory* [5], with Σ an algebraic signature declaring sorts, subsorts, and function symbols, E a set of conditional equations, and A a set of equational axioms. It 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, of the form $[l] : t \longrightarrow t'$ **if** $cond$, where $cond$ is a condition 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.

Maude [5] is a language and tool for specifying, simulating, and model checking rewrite theories. The distributed state of an object-oriented system is formalized as a *multiset* of objects and messages. A class C with attributes att_1 to att_n of sorts s_1 to s_n is declared `class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n. An object of class C is modeled as a term `<` o `:` C `|` att_1 `:` v_1`, ... ,` att_n `:` v_n `>`, with o its object identifier, and where the attributes att_1 to att_n have the current values v_1 to v_n, respectively. Upon receiving a message, an object can change its state and/or send messages to other objects. For example, the rewrite rule

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

defines a transition where an incoming message `m`, with parameters `O` and `z`, is consumed by the target object `O` of class `C`, the attribute `a1` is updated to `x + z`, and an outgoing message `m'(O',x + z)` is generated.

Statistical Model Checking and PVeStA. Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [1] with rules of the form

$$[l] : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \ \ \textbf{if} \ \ cond(\overrightarrow{x}) \ \ \textit{with} \ \ \textit{probability} \ \ \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. The concrete values of the new variables $\overrightarrow{y}$ in $t'(\overrightarrow{x}, \overrightarrow{y})$ are chosen probabilistically according to the probability distribution $\pi(\overrightarrow{x})$.

Statistical model checking [18,21] is an attractive formal approach to analyzing (purely) probabilistic systems. Instead of offering a yes/no answer, it can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. We then use PVeStA [2], a parallelization of the tool VeStA [19], to statistically model check purely probabilistic systems against properties expressed as QuaTEx expressions [1]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α

and δ by sampling, until we obtain a value v so that with $(1-\alpha)100\%$ statistical confidence, the expected value is in the interval $[v - \frac{\delta}{2}, v + \frac{\delta}{2}]$.

3 The ROLA Multi-Partition Transaction Algorithm

Our new algorithm for distributed multi-partition transactions, ROLA, extends RAMP-Fast. RAMP-Fast guarantees RA, but it does not guarantee PLU since it allows a write to overwrite conflicting writes: When a partition commits a write, it only compares the write's timestamp t_1 with the local latest-committed timestamp t_2, and updates the latest-committed timestamp with t_1 or t_2. If the two timestamps are from two conflicting writes, then one of the writes is lost.

ROLA's key idea to prevent lost updates is to sequentially order writes on the same key from a partition's perspective by adding to each partition a data structure which maps each incoming version to an incremental sequence number. For write-only transactions the mapping can always be built; for a read-write transaction the mapping can only be built if there has not been a mapping built since the transaction fetched the value. This can be checked by comparing the last prepared version's timestamp's mapping on the partition with the fetched version's timestamp's mapping. In this way, ROLA prevents lost updates by allowing versions to be prepared only if no conflicting prepares occur concurrently.

More specifically, ROLA adds two partition-side data structures: *sqn*, denoting the local sequence counter, and *seq*[*ts*], that maps a timestamp to a local sequence number. ROLA also changes the data structure of *versions* in RAMP from a set to a list. ROLA then adds two methods: the coordinator-side[2] method UPDATE(I : set of items, OP : set of operations) and the partition-side method PREPARE_UPDATE(v : version, ts_{prev} : timestamp) for read-write transactions. Furthermore, ROLA changes two partition-side methods in RAMP: PREPARE, besides adding the version to the local store, maps its timestamp to the increased local sequence number; and COMMIT marks versions as committed and updates an index containing the highest-sequenced-timestamped committed version of each item. These two partition-side methods apply to both write-only and read-write transactions. ROLA invokes RAMP-Fast's PUT_ALL, GET_ALL and GET methods (see [3,14]) to deal with read-only and write-only transactions.

ROLA starts a read-write transaction with the UPDATE procedure. It invokes RAMP-Fast's GET_ALL method to retrieve the values of the items the client wants to update, as well as their corresponding timestamps. ROLA writes then proceed in two phases: a first round of communication places each timestamped write on its respective partition. The timestamp of each version obtained previously from the GET_ALL call is also packaged in this *prepare* message. A second round of communication marks versions as committed.

At the partition-side, the partition begins the PREPARE_UPDATE routine by retrieving the last version in its *versions* list with the same item as the received version. If such a version is not found, or if the version's timestamp ts_v matches

[2] The *coordinator*, or *client*, is the partition executing the transaction.

Algorithm 1. ROLA

```
***Server-side Data Structures***
$versions$: list of versions ⟨item, value, timestamp $ts_v$, metadata $md$⟩
$latestCommit[i]$: last committed timestamp for item $i$
$seq[ts]$: local sequence number mapped to timestamp $ts$
$sqn$: local sequence counter

***Server-side Methods***
GET same as in RAMP-Fast

procedure PREPARE_UPDATE($v$ : version, $ts_{prev}$ : timestamp)
    $latest \leftarrow$ last $w \in versions : w.item = v.item$
    if $latest =$ NULL or $ts_{prev} = latest.ts_v$ then
        $sqn \leftarrow sqn + 1$;  $seq[v.ts_v] \leftarrow sqn$;  $versions$.add($v$)
        return ACK
    else return $latest$

procedure PREPARE($v$ : version)
    $sqn \leftarrow sqn + 1$;  $seq[v.ts_v] \leftarrow sqn$;  $versions$.add($v$)

procedure COMMIT($ts_c$ : timestamp)
    $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
    for $i \in I_{ts}$ do
        if $seq[ts_c] > seq[latestCommit[i]]$ then $latestCommit[i] \leftarrow ts_c$

***Coordinator-side Methods***
PUT_ALL, GET_ALL same as in RAMP-Fast

procedure UPDATE($I$ : set of items, $OP$ : set of operations)
    $ret \leftarrow$ GET_ALL($I$); $ts_{tx} \leftarrow$ generate new timestamp
    parallel-for $i \in I$ do
        $ts_{prev} \leftarrow ret[i].ts_v$;  $v \leftarrow ret[i].value$
        $w \leftarrow \langle item = i, value = op_i(v), ts_v = ts_{tx}, md = (I - \{i\})\rangle$
        $p \leftarrow$ PREPARE_UPDATE($w$,$ts_{prev}$)
        if $p = latest$ then
            invoke application logic to, e.g., abort and/or retry the transaction
    end parallel-for
    parallel-for server $s$ : $s$ contains an item in $I$ do
        invoke COMMIT($ts_{tx}$) on $s$
    end parallel-for
```

the passed-in timestamp ts_{prev}, then the version is deemed prepared. The partition keeps a record of this locally by incrementing a local sequence counter and mapping the received version's timestamp ts_v to the current value of the sequence counter. Finally the partition returns an ACK to the client. If ts_{prev}

does not match the timestamp of the last version in *versions* with the same item, then this *latest* timestamp is simply returned to the coordinator.

If the coordinator receives an ACK from PREPARE_UPDATE, it immediately commits the version with the generated timestamp ts_{tx}. If the returned value is instead a timestamp, the transaction is aborted.

4 A Probabilistic Model of ROLA

This section defines a formal executable probabilistic model of ROLA. The whole model is given at https://sites.google.com/site/fase18submission/.

As mentioned in Sect. 2, statistical model checking assumes that the system is *fully probabilistic*; that is, has no unquantified nondeterminism. We follow the techniques in [6] to obtain such a model. The key idea is that message delays are sampled probabilistically from dense/continuous time intervals. The probability that two messages will have the same delay is therefore 0. If events only take place when a message arrives, then two events will not happen at the same time, and therefore unquantified nondeterminism is eliminated.

We are also interested in correctness analysis of a model that captures all possible behaviors from a given initial configuration. We obtain such a nondeterministic untimed model, that can be subjected to standard model checking analysis, by just removing all message delays from our probabilistic timed model.

4.1 Probabilistic Sampling

Nodes send messages of the form `[`Δ`,` *rcvr* `<-` *msg*`]`, where Δ is the message delay, *rcvr* is the recipient, and *msg* is the message content. When time Δ has elapsed, this message becomes a *ripe* message `{`T`,`*rcvr* `<-` *msg*`}`, where T is the "current global time" (used for analysis purposes only).

To sample message delays from different distributions, we use the following functionality provided by Maude: The function `random`, where `random(`k`)` returns the k-th pseudo-random number as a number between 0 and $2^{32} - 1$, and the built-in constant `counter` with an (implicit) rewrite rule `counter => N:Nat`. The first time `counter` is rewritten, it rewrites to 0, the next time it rewrites to 1, and so on. Therefore, each time `random(counter)` rewrites, it rewrites to the next random number. Since Maude does not rewrite `counter` when it appears in the condition of a rewrite rule, we encode a probabilistic rewrite rule $t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y})$ **if** $cond(\overrightarrow{x})$ *with probability* $\overrightarrow{y} := \pi(\overrightarrow{x})$ in Maude as the rule $t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, sample(\pi(\overrightarrow{x})))$ **if** $cond(\overrightarrow{x})$. The following operator `sampleLogNormal` is used to sample a value from a lognormal distribution with mean `MEAN` and standard deviation `SD`:

```
op sampleLogNormal : Float Float -> [Float] .
eq sampleLogNormal(MEAN,SD) = exp(MEAN + SD * sampleNormal) .

op sampleNormal : -> [Float] .   op sampleNormal : Float -> [Float] .
eq sampleNormal = sampleNormal(float(random(counter) / 4294967296)) .
eq sampleNormal(RAND) = sqrt(- 2.0 * log(RAND)) * cos(2.0 * pi * RAND) .
```

random(counter)/4294967296 rewrites to a different "random" number between 0 and 1 each time it is rewritten, and this is used to define the sampling function. For example, the message delay rd to a remote site can be sampled from a lognormal distribution with mean 3 and standard deviation 2 as follows:

```
eq rd = sampleLogNormal(3.0, 2.0) .
```

4.2 Data Types, Classes, and Messages

We formalize ROLA in an object-oriented style, where the state consists of a number of *partition* objects, each modeling a partition of the database, and a number of messages traveling between the objects. A *transaction* is formalized as an object which resides inside the partition object that executes the transaction.

Data Types. A *version* is a timestamped version of a data item (or key) and is modeled as a 4-tuple version(*key*, *value*, *timestamp*, *metadata*). A timestamp is modeled as a pair ts(*addr*, *sqn*) consisting of a partition's identifier *addr* and a local sequence number *sqn*. Metadata are modeled as a set of keys, denoting, for each key, the other keys that are written in the same transaction.

The sort OperationList represents lists of read and write operations as terms such as (x := read $k1$) (y := read $k2$) write($k1, x+y$), where LocalVar denotes the "local variable" that stores the value of the key read by the operation, and Expression is an expression involving the transaction's local variables:

```
op write : Key Expression -> Operation [ctor] .
op _:=read_ : LocalVar Key -> Operation [ctor] .
pr LIST{Operation} * (sort List{Operation} to OperationList) .
```

Classes. A *transaction* is modeled as an object of the following class Txn:

```
class Txn | operations : OperationList,  readSet : Versions,
            localVars : LocalVars,       latest : KeyTimestamps .
```

The operations attribute denotes the transaction's operations. The readSet attribute denotes the versions read by the read operations. localVars maps the transaction's local variables to their current values. latest stores the local view as a mapping from keys to their respective latest committed timestamps.

A *partition* (or *site*) stores parts of the database, and executes the transactions for which it is the coordinator/server. A partition is formalized as an object instance of the following class Partition:

```
class Partition | datastore : Versions,     sqn : Nat,
                  gotTxns : ObjectList,     executing : Object,
                  committed : ObjectList,   aborted : ObjectList,
                  tsSqn : TimestampSqn,     latestCommit : KeyTimestamps,
                  votes : Vote,             voteSites : TxnAddrSet,
                  1stGetSites : TxnAddrSet, 2ndGetSites : TxnAddrSet,
                  commitSites : TxnAddrSet .
```

The `datastore` attribute represents the partition's local database as a list of versions for each key stored at the partition. The attribute `latestCommit` maps to each key the timestamp of its last committed version. `tsSqn` maps each version's timestamp to a local sequence number `sqn`. The attributes `gotTxns`, `executing`, `committed` and `aborted` denote the transaction(s) which are, respectively, waiting to be executed, currently executing, committed, and aborted.

The attribute `votes` stores the votes in the two-phase commit. The remaining attributes denote the partitions from which the executing partition is awaiting votes, committed acks, first-round get replies, and second-round get replies.

The following shows an initial state (with some parts replaced by '...') with two partitions, `p1` and `p2`, that are coordinators for, respectively, transactions `t1`, and `t2` and `t3`. `p1` stores the data items `x` and `z`, and `p2` stores `y`. Transaction `t1` is the read-only transaction `(xl := read x) (yl := read y)`, transaction `t2` is a write-only transaction `write(y,3) write(z,8)`, while transaction `t3` is a read-write transaction on data item `x`. The states also include a buffer of messages in transit and the global clock value, and a table which assigns to each data item the site storing the item. Initially, the value of each item is `[0]`; the version's timestamp is empty (`eptTS`), and metadata is an empty set.

```
eq init = { 0.0 | nil}
< tb : Table | table : [sites(x,p1) ;; sites(y,p2) ;; sites(z,p1)] >
< p1 : Partition |
        gotTxns: < t1 : Txn | operations: ((xl :=read x) (yl :=read y)),
                              readSet: empty, latest: empty,
                              localVars: (xl |-> [0], yl |-> [0]) >,
        datastore: (version(x, [0],eptTS,empty)
                    version(z, [0],eptTS,empty)),
        sqn: 1,  ... >
< p2 : Partition |
        gotTxns: < t2 : Txn | operations: (write(y,3) write(z,8)), ... >
                 < t3 : Txn | operations: ((xl := read x)
                                           write(x, xl plus 1)),  ... >
        datastore: version(y, [0],eptTS,empty), ... > .
```

Messages. The message `prepare(`*txn*, *version*, *sender*`)` sends a version from a write-only transaction to its partition, and `prepare(`*txn*, *version*, *ts*, *sender*`)` does the same thing for other transactions, with *ts* the timestamp of the version it read. The partition replies with a message `prepare-reply(`*txn*, *vote*, *sender*`)`, where *vote* tells whether this partition can commit the transaction. A message `commit(`*txn*, *ts*, *sender*`)` marks the versions with timestamp *ts* as committed. `get(`*txn*, *key*, *ts*, *sender*`)` asks for the highest-timestamped committed version or a missing version for *key* by timestamp *ts*, and `response1(`*txn*, *version*, *sender*`)` and `response2(`*txn*, *version*, *sender*`)` respond to first/second-round `get` requests.

4.3 Formalizing ROLA's Behaviors

This section formalizes the dynamic behaviors of ROLA using rewrite rules, referring to the corresponding lines in Algorithm 1. We only show 2 of the 15 rewrite rules in our model, and refer to the report [14] for further details.[3]

Receiving `prepare` Messages (lines 5–10). When a partition receives a `prepare` message for a read-write transaction, the partition first determines whether the timestamp of the last version (`VERSION`) in its local version list `VS` matches the incoming timestamp `TS'` (which is the timestamp of the version read by the transaction). If so, the incoming version is added to the local store, the map `tsSqn` is updated, and a positive reply (`true`) to the prepare message is sent ("**return** *ack*" in our pseudo-code); otherwise, a negative reply (`false`, or "**return** *latest*" in the pseudo-code) is sent. Depending on whether the sender `PID'` of the *prepare* message happens to be `PID` itself, the reply is equipped with a local message delay `ld` or a remote message delay `rd`, both of which are sampled probabilistically from distributions with different parameters:[4]

```
crl [receive-prepare-rw] :
    {T, PID <- prepare(TID,version(K,V,TS,MD),TS',PID')}
    < PID : Partition | datastore: VS,  sqn: SQN,  tsSqn: TSSQN,  AS' >
   =>
    if VERSION == eptVersion or tstamp(VERSION) == TS'
    then < PID : Partition | datastore: (VS version(K,V,TS,MD)), sqn: SQN',
                             tsSqn: insert(TS,SQN',TSSQN), AS' >
         [if PID == PID' then ld else rd fi,
             PID' <- prepare-reply(TID,true,PID)]
    else < PID : Partition | datastore: VS,  sqn: SQN,  tsSqn: TSSQN, AS' >
         [if PID == PID' then ld else rd fi,
             PID' <- prepare-reply(TID,false,PID)]  fi
    if SQN' := SQN + 1 /\ VERSION := latestPrepared(K,VS) .
```

Receiving Negative Replies (lines 23–24). When a site receives a `prepare-reply` message with vote `false`, it aborts the transaction by moving it to the `aborted` list, and removes `PID'` from the "vote waiting list" for this transaction:

```
rl [receive-prepare-reply-false-executing] :
   {T, PID <- prepare-reply(TID,false,PID')}
   < PID : Partition | executing: < TID : Txn | AS >,  aborted: TXNS,
                       voteSites: VSTS addrs(TID,(PID',PIDS)), AS' >
 =>
   < PID : Partition | executing: noTxn,
                       aborted: (TXNS ;; < TID : Txn | AS >),
                       voteSites: VSTS addrs(TID,PIDS), AS' > .
```

[3] We do not give variable declarations, but follow the convention that variables are written in (all) capital letters.

[4] The variable `AS'` denotes the "remaining" attributes in the object.

5 Correctness Analysis of ROLA

In this section we use reachability analysis to analyze whether ROLA guarantees read atomicity and prevents lost updates.

For both correctness and performance analysis, we add to the state an object

```
< m : Monitor | log: log >
```

which stores crucial information about each transaction. The *log* is a list of records `record(`*tid*, *issueTime*, *finishTime*, *reads*, *writes*, *committed*`)`, with *tid* the transaction's ID, *issueTime* its issue time, *finishTime* its commit/abort time, *reads* the versions read, *writes* the versions written, and *committed* a flag that is `true` if the transaction is committed.

We modify our model by updating the `Monitor` when needed. For example, when the coordinator has received all `committed` messages, the monitor records the commit time (`T`) for that transaction, and sets the "committed" flag to `true`[5]:

```
crl [receive-committed] :
    {T, PID <- committed(TID,PID’)}
    < M : Monitor | log: (LOG record(TID, T’, T’’, RS, WS, false) LOG’) >
    < PID : Partition | executing: < TID : Txn | AS >,
                        committed: TXNS, commitSites: CMTS,  AS’ >
  =>
    if CMTS’[TID] == empty  --- all "committed" received
    then < M : Monitor | log: (LOG record(TID, T’, T, RS, WS, true) LOG’) >
         < PID : Partition | executing: noTxn,  commitSites: CMTS’,
                             committed: (TXNS ;; < TID : Txn | AS >, AS’ >
    else < M : Monitor | log: (LOG record(TID, T’, T’’, RS, WS, false) LOG’) >
         < PID : Partition | executing: < TID : Txn | AS >,
                             committed: TXNS, commitSites: CMTS’, AS’ > fi
    if CMTS’ := remove(TID,PID’,CMTS) .
```

Since ROLA is terminating if a finite number of transactions are issued, we analyze the different (correctness and performance) properties by inspecting this monitor object in the final states, when all transactions are finished.

Read Atomicity. A system guarantees RA if it prevents fractured reads, and also prevents transactions from reading uncommitted, aborted, or intermediate data [3], where a transaction T_j exhibits *fractured reads* if transaction T_i writes version x_m and y_n, T_j reads version x_m and version y_k, and $k < n$ [3].

We analyze this property by searching for a reachable *final* state (arrow `=>!`) where the property does *not* hold:

```
search [1] initConfig =>! C:Config < M:Address : Monitor | log: LOG:Record >
    such that fracRead(LOG) or abortedRead(LOG) .
```

[5] The additions to the original rule are written in italics.

The function `fracRead` checks whether there are fractured reads in the execution log. There is a fractured read if a transaction `TID2` reads `X` and `Y`, transaction `TID1` writes `X` and `Y`, `TID2` reads the version `TSX` of `X` written by `TID1`, and reads a version `TSY'` of `Y` written *before* `TSY` (`TSY' < TSY`). Since the transactions in the log are ordered according to start time, `TID2` could appear *before* or *after* `TID1` in the log. We spell out the case when `TID1` comes before `TID2`:

```
op fracRead : Record -> Bool .
ceq fracRead(LOG ;
     record(TID1,T1,T1',RS1,(version(X,VX,TSX,MDX), version(Y,VY,TSY,MDY)),true) ; LOG' ;
     record(TID2,T2,T2',(version(X,VX,TSX,MDX), version(Y,VY',TSY',MDY')), WS2,true) ; LOG'')
   = true if TSY' < TSY .
ceq fracRead(LOG ; record(TID2, ...) ; LOG' ; record(TID1, ...) ; LOG'') = true if TSY' < TSY .
eq fracRead(LOG) = false [owise] .
```

The function `abortedRead` checks whether a transaction `TID2` reads a version `TSX` that was written by an aborted (flag `false`) transaction `TID1`:

```
op abortedRead : Record -> Bool .
eq abortedRead(LOG ;
      record(TID1, T1, T1', RS1, (version(X,VX,TSX,MDX), VS), false) ; LOG' ;
      record(TID2, T2, T2', (version(X,VX,TSX,MDX), VS), WS2, true) ; LOG'') = true .
eq abortedRead(LOG ; record(TID2,...) ; LOG' ; record(TID1,...) ; LOG'') = true.
eq abortedRead(LOG) = false [owise] .
```

No Lost Updates. We analyze the PLU property by searching for a final state in which the monitor shows that an update was lost:

```
search [1] initConfig =>! C:Config < M:Address : Monitor | log: LOG:Record >
    such that lu(LOG) .
```

The function `lu`, described in [14], checks whether there are lost updates in `LOG`.

We have performed our analysis with 4 different initial states, with up to 8 transactions, 2 data items and 4 partitions, without finding a violation of RA or PLU. We have also model checked the causal consistency (CC) property with the same initial states, and found a counterexample showing that ROLA does *not* satisfy CC. (This might imply that our initial states are large enough so that violations of RA or PLU could have been found by model checking.) Each analysis command took about 30 seconds to execute on a 2.9 GHz Intel 4-Core i7-3520M CPU with 3.7 GB memory.

6 Statistical Model Checking of ROLA and Walter

The weakest consistency model in [4] guaranteeing RA and PLU is PSI, and the main system providing PSI is Walter [20]. ROLA must therefore outperform Walter to be an attractive design. To quickly check whether ROLA does so, we have also modeled Walter—without its data replication features—in Maude (see [11] and https://sites.google.com/site/fase18submission/maude-spec), and use statistical model checking with PVeStA to compare the performance of ROLA and Walter in terms of throughput and average transaction latency.

Extracting Performance Measures from Executions. PVeStA estimates the expected (average) value of an expression on a run, up to a desired statistical confidence. The key to perform statistical model checking is therefore to define a measure on runs. Using the monitor in Sect. 5 we can define a number of functions on (states with) such a monitor that extract different performance metrics from this "system execution log."

The function `throughput` computes the number of committed transactions per time unit. `committedNumber` computes the number of committed transactions in `LOG`, and `totalRunTime` returns the time when all transactions are finished (i.e., the largest *finishTime* in `LOG`):

```
op throughput : Config -> Float [frozen] .
eq throughput(< M : Monitor | log: LOG >  REST)
 = committedNumber(LOG) / totalRunTime(LOG) .
```

The function `avgLatency` computes the average transaction latency by dividing the sum of the latencies of all committed transactions by the number of such transactions:

```
op avgLatency : Config -> Float [frozen] .
eq avgLatency(< M : Monitor | log: LOG >  REST)
 =  totalLatency(LOG) / committedNumber(LOG) .
```

where `totalLatency` computes the sum of all transaction latencies (time between the issue time and the finish time of a committed transaction).

Generating Initial States. We use an operator `init` to *probabilistically* generate initial states: `init(`*rtx*, *wtx*, *rwtx*, *part*, *keys*, *rops*, *wops*, *rwops*, *distr*`)` generates an initial state with *rtx* read-only transactions, *wtx* write-only transactions, *rwtx* read-write transactions, *part* partitions, *keys* data items, *rops* operations per read-only transaction, *wops* operations per write-only transaction, *rwops* operations per read-write transactions, and *distr* the key access distribution (the probability that an operation accesses a certain data item). To capture the fact that some data items may be accessed more frequently than others, we also use Zipfian distributions in our experiments.

Statistical Model Checking Results. We performed our experiments under different configurations, with 200 transactions, 2–4 operations per transaction, up to 200 data items and 50 partitions, with lognormal message delay distributions, and with uniform and Zipfian data item access distributions.

The plots in Fig. 1 show the *throughput* as a function of the percentage of read-only transactions, number of partitions, and number of keys (data items), sometimes with both uniform and Zipfian distributions. The plots show that ROLA outperforms Walter for all parameter combinations. More partitions gives ROLA higher throughput (since concurrency increases), as opposed to Walter (since Walter has to propagate transactions to more partitions to advance the vector timestamp). We only plot the results under uniform key access distribution, which are consistent with the results using Zipfian distributions.

The plots in Fig. 2 show the *average transaction latency* as a function of the same parameters as the plots for throughput. Again, we see that ROLA outperforms Walter in all settings. In particular, this difference is quite large for write-heavy workloads; the reason is that Walter incurs more and more overhead for providing causality, which requires background propagation to advance the vector timestamp. The latency tends to converge under read-heavy workload (because reads in both ROLA and Walter can commit locally without certification), but ROLA still has noticeable lower latency than Walter.

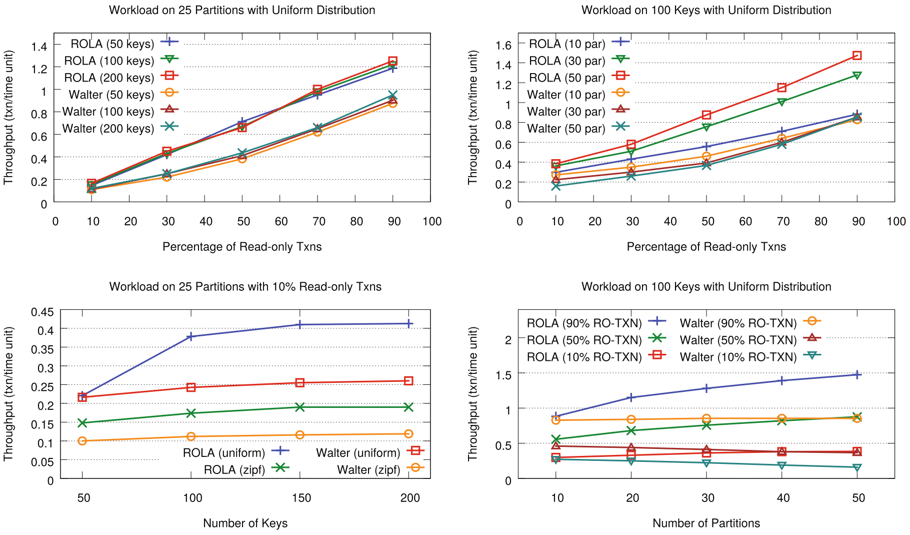

Fig. 1. Throughput comparison under different workload conditions.

Computing the probabilities took 6 hours (worst case) on 10 servers, each with a 64-bit Intel Quad Core Xeon E5530 CPU with 12 GB memory. Each point in the plots represents the average of three statistical model checking results.

7 Related Work

Maude and PVeStA have been used to model and analyze the correctness and performance of a number of distributed data stores: the Cassandra key-value store [12,15], different versions of RAMP [10,13], and Google's Megastore [7,8]. In contrast to these papers, our paper uses formal methods to develop and validate an entirely new design, ROLA, for a new consistency model.

Concerning formal methods for distributed data stores, engineers at Amazon have used TLA+ and its model checker TLC to model and analyze the correctness of key parts of Amazon's celebrated cloud computing infrastructure [17].

In contrast to our work, they only use formal methods for correctness analysis; indeed, one of their complaints is that they cannot use their formal method for performance estimation. The designers of the TAPIR transaction protocol for distributed storage systems have also specified and model checked correctness (but not performance) properties of their design using TLA+ [22].

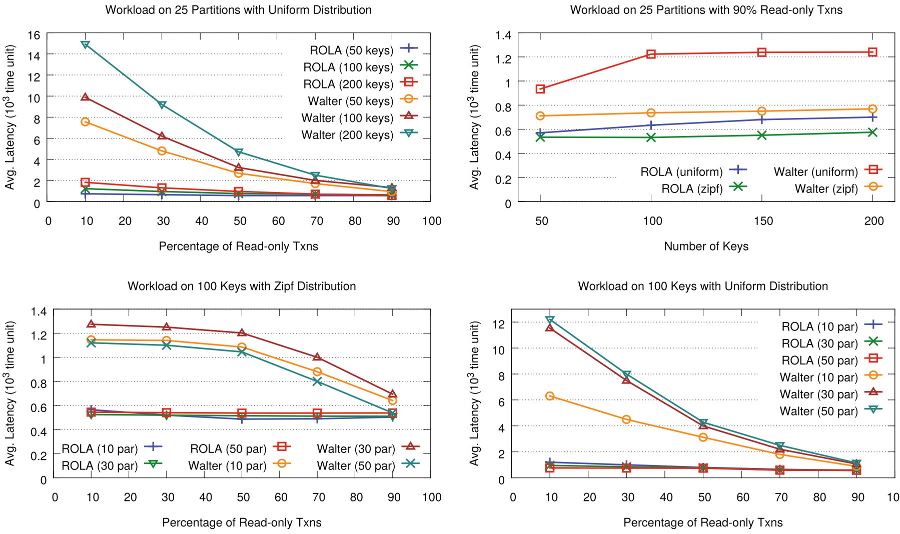

Fig. 2. Average latency comparison across varying workload conditions.

8 Conclusions

We have presented the formal design and analysis of ROLA, a distributed transaction protocol that supports a new consistency model not present in the survey by Cerone *et al.* [4]. Using formal modeling and both standard and statistical model checking analyses we have: (i) validated ROLA's RA and PLU consistency requirements; and (ii) analyzed its performance requirements, showing that ROLA outperforms Walter in all performance measures.

This work has shown, to the best of our knowledge for the first time, that the design and validation of a *new* distributed transaction protocol can be achieved relatively quickly *before* its implementation by the use of formal methods. Our next planned step is to implement ROLA, evaluate it experimentally, and compare the experimental results with the formal analysis ones. In previous work on existing systems such as Cassandra [9] and RAMP [3], the performance estimates obtained by formal analysis and those obtained by experimenting with the real system were basically in agreement with each other [10,12]. This confirmed the useful predictive power of the formal analyses. Our future research will investigate the existence of a similar agreement for ROLA.

Acknowledgments. We thank Andrea Cerone, Alexey Gotsman, Jatin Ganhotra, and Rohit Mukerji for helpful early discussions on this work, and the anonymous reviewers for useful comments. This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, and a generous gift from Microsoft.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. **153**(2), 213–239 (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_28
3. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. **41**(3), 15:1–15:45 (2016)
4. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: CONCUR. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
6. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 143–160. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37635-1_9
7. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54624-2_25
8. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 159–174. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_12
9. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media, Sebastopol (2010)
10. Liu, S., Ölveczky, P.C., Ganhotra, J., Gupta, I., Meseguer, J.: Exploring design alternatives for RAMP transactions through statistical model checking. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 298–314. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_18
11. Liu, S., Ölveczky, P.C., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store. In: Proceedings of WRLA 2018. LNCS. Springer (2018, to appear). https://sites.google.com/site/siliunobi/walter
12. Liu, S., Ganhotra, J., Rahman, M., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Trans. Embed. Syst. **4**(1), 03:1–03:26 (2017)

13. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: SAC 2016. ACM (2016)
14. Liu, S., Ölveczky, P.C., Santhanam, K., Wang, Q., Gupta, I., Meseguer, J.: ROLA: a new distributed transaction protocol and its formal analysis (2017). https://sites.google.com/site/fase18submission/tech-report
15. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 332–347. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11737-9_22
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
17. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
18. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_26
19. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: a statistical model-checker and analyzer for probabilistic systems. In: QEST 2005. IEEE Computer Society (2005)
20. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP 2011. ACM (2011)
21. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. Comput. **204**(9), 1368–1409 (2006)
22. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: Proceedings of Symposium on Operating Systems Principles, SOSP 2015. ACM (2015)