Read Atomic Transactions with Prevention of Lost Updates: ROLA and its Formal Analysis

Si Liu[1], Peter Csaba Ölveczky[2], Qi Wang[1], Indranil Gupta[1], and José Meseguer[1]

[1]University of Illinois, Urbana-Champaign, USA

[2]University of 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) (either all or none of a transaction's updates are visible to other transactions) and *prevention of lost updates* (PLU). Existing distributed transaction systems that meet these requirements also provide additional stronger consistency guarantees (such as *causal consistency*), but this comes at the price of lower performance. In this paper we propose a new distributed transaction protocol, ROLA, that targets application scenarios where only RA and PLU are needed. We formally specify 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) perform statistical model checking to analyze key performance properties; and (b) compare these performance results with those obtained by also modeling and analyzing in Maude the well-known protocols Walter and Jessy that also guarantee RA and PLU. Our statistical model checking results show that ROLA outperforms both Walter and Jessy.

Keywords: Distributed Database Systems; Transaction Protocols; Consistency Models; Performance Evaluation; Statistical Model Checking; Rewriting Logic; Maude

1. Introduction

Distributed transaction protocols are complex distributed systems whose design is quite challenging because: (i) as for other distributed systems, 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 expensive to compare across different implementations; 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. This is also part of meeting design challenge (iii), since the key comparisons focus on how well each protocol balances the consistency vs. performance trade-offs for the intended applications.

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.* [11] 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): Strengthens CC by also preventing lost updates. PSI allows different commit orders at different sites, while still guaranteeing that a transaction reads the most recent version committed at the transaction execution site, as of the time when the transaction begins. For example, A sees C's post before seeing D's post, whereas B sees D's post before C's post, assuming the two posts are independent of each other. In other words, C and D can commit their posts without waiting for each other.
- 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 [11] satisfying both RA and PLU is PSI. However, PSI, and the well-known protocol Walter [34] implementing PSI, also guarantee CC. Furthermore, in [6], Ardekani et al. propose a consistency model called *non-monotonic snapshot isolation* (NMSI)—and a distributed transaction protocol called Jessy that implements NMSI—that is weaker than PSI, but still satisfies RA, CC, and PLU. To the best of our knowledge, up to now NMSI has in fact been the weakest consistency model satisfying both RA and PLU, which means that all current such models also satisfy CC. However, Cerone et al. conjecture in [11] that a system guaranteeing RA and PLU *without* guaranteeing CC should be useful:

"*existing consistency models do not include a counterpart of Read Atomic obtained by adding the* NoConflict *axiom [preventing lost updates]. Such an 'Update Atomic' consistency model would prevent lost update anomalies without having to enforce causal consistency [...]. Update Atomic could be particularly useful [...].*"
There was until now no database design supporting such "update atomicity"[1] without also providing CC. Filling this gap; that is, presenting a design, ROLA, that does exactly this for multi-partition transactions, is what we do in this paper.

The main idea of the ROLA algorithm is to extend the RAMP algorithm of Bailis et al. [7], that ensures read atomicity for partitioned data stores (i.e., data are partitioned across widely distributed data centers, but are not replicated) where a transaction can read and/or write data stored at different partitions, by adding mechanisms for preventing lost updates. Therefore, unlike Jessy and Walter, which support partially replicated data stores, ROLA, like RAMP, at the moment only targets partitioned data stores.

Two key questions about ROLA's design are:

(a) Are there *natural applications* needing high performance where RA plus PLU provide a sufficient degree of consistency?
(b) Can the new ROLA design meeting RA plus PLU *outperform* existing designs, like Walter and Jessy, that also guarantee RA and PLU?

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. [7] point out that RA is crucial for this operation: If Edinson and Neymar become friends, then Thomas 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

[1] In the above hierarchy of consistency models, ROLA's "update atomic" is strictly stronger than RA, incomparable with CC, and strictly weaker than NMSI.

guarantee PLU: the new friendship between Edinson and Neymar must 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 Thomas 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), Section 7 shows that ROLA clearly outperforms both Walter and Jessy in all performance requirements for all read/write transaction rates. For a fair comparison, we have compared the performance of ROLA with those of Jessy and Walter without their replication features.

Maude-Based Formal Modeling and Analysis. In rewriting logic [26], distributed systems are specified as *rewrite theories*. Maude [12] 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 *probabilisitic rewrite theory* [4, 12]. 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 those of Jessy and Walter, we have also specified Walter and Jessy—without their data replication features—in Maude, and have subjected the Maude models of ROLA, Walter, and Jessy to *statistical model checking* analysis using the PVeStA [5] 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 and Jessy protocols showing that ROLA outperforms both Walter and Jessy in all such comparisons, including higher throughput and lower average latency; (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.

This paper is an extension our conference paper [23]. In addition to giving more details in general, the new contributions can be summarized as follows:

- First and foremost, in contrast to [23], where we only showed 2 of 15 rewrite rules specifying ROLA, we now give a much more thorough formal specification of ROLA by showing 13 rewrite rules and providing the definition of many key functions in our formal specification.
- In [23] we only compared ROLA to the Walter protocol guaranteeing PSI. However, since Jessy only guarantees the weaker NMSI property (that still guarantees RA and PLU), Jessy should outperform Walter. Therefore, a performance comparison with Jessy is necessary and is provided in this paper.
- We provide an informal correctness argument for ROLA.
- In [23] we show how to formalize and model check in Maude the RA property; in this paper we also show how PLU and CC can be formalized and analyzed using Maude.
- This paper also analyzes the transaction commit rate for ROLA, Jessy, and Walter.
- This paper shows how to generate initial states probabilistically.

The rest of this paper is structured as follows: Section 2 gives preliminaries on the RAMP transaction protocol, which ROLA extends, rewriting logic and Maude, and statistical model checking of probabilistic rewrite theories using PVeStA. Section 3 gives an informal overview of the ROLA transaction protocol, and Section 4 gives an informal correctness argument that ROLA satisfies RA and PLU. Section 5 presents our formal specification of ROLA. Section 6 explains how the RA, PLU, and CC properties can be formalized in Maude, and how we can use Maude reachability analysis to automatically check whether ROLA satisfies these properties for given initial system states. Section 7 explains how we can use our formal model and the PVeStA statistical model to estimate the performance of ROLA, as well as of Jessy and Walter, in terms of average transaction latency, transaction commit rate, and transaction throughput, and shows the results of estimating the performance of both ROLA, Walter, and Jessy. Finally, Section 8 discusses related work, and Section 9 gives some concluding remarks.

2. Preliminaries

2.1. Rewriting Logic and Maude

A *membership equational logic* (MEL) [27] *signature* is a triple $\Sigma = (K, \Sigma, S)$ with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A Σ-*algebra* A consists of a set A_k for each kind k, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1\cdots k_n,k}$, and a subset inclusion $A_s \subseteq A_k$ for each sort $s \in S_k$. The set $T_{\Sigma,k}$ denotes the set of ground Σ-terms with kind k, and $T_\Sigma(X)_k$ denotes the set of Σ-terms with kind k over the set X of kinded variables.

A MEL *theory* is a pair (Σ, E) with Σ a MEL-signature and E a finite set of MEL sentences, which are either conditional equations or conditional memberships of the forms:

$$(\forall X)\; t = t' \;\textbf{if}\; \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j, \qquad (\forall X)\; t : s \;\textbf{if}\; \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j,$$

where $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$ for some kind $k \in \Sigma$, the latter stating that t is a term of sort s, provided the condition holds. In Maude, an individual equation in the condition may also be a *matching equation* p_l `:=` q_l, which is mathematically interpreted as an ordinary equation. However, operationally, the new variables occurring in the term p_l become instantiated by matching the term p_l against the canonical form of the instance of q_l (see [12] for further explanations). Order-sorted notation $s_1 < s_2$ abbreviates the conditional membership $(\forall x : [s_1])\; x : s_2$ **if** $x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall\, x_1 : [s_1], \ldots, x_n : [s_n])\; f(x_1, \ldots, x_n) : s$ **if** $\bigwedge_{1\leq i\leq n} x_i : s_i$.

A Maude module specifies a *rewrite theory* [26] of the form $(\Sigma, E \cup A, R)$, where: (i) $(\Sigma, E \cup A)$ is a membership equational logic theory specifying the system's state space as an algebraic data type with A a set of equational axioms (such as a combination of associativity, commutativity, and identity axioms), to perform equational deduction with the equations E (oriented from left to right) *modulo* the axioms A, and (ii) R is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form:

$$l : q \longrightarrow r \;\textbf{if}\; \bigwedge_i p_i = q_i \;\wedge\; \bigwedge_j w_j : s_j \;\wedge\; \bigwedge_m t_m \longrightarrow t'_m,$$

where l is a *label*, and q, r are Σ-terms of the same kind. Intuitively, such a rule specifies a *one-step transition* from a substitution instance of q to the corresponding substitution instance of r, *provided* the condition holds; that is, that the substitution instance of each condition in the rule follows from $\mathcal{R}$.

We briefly summarize the syntax of Maude and refer to [12] for more details. Sorts and subsort relations are declared by the keywords `sort` and `subsort`, and operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1\, \ldots\, s_n$ are the sorts of its arguments, and s is its (value) *sort*. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (`ctor`) that defines the data elements of a sort.

There are three kinds of logical statements in the Maude language, *equations*, *memberships* (declaring that a term has a certain sort), and *rewrite rules*, introduced with the following syntax:

- equations: `eq` $u = v$ or `ceq` $u = v$ `if` *condition*;
- memberships: `mb` $u : s$ or `cmb` $u : s$ `if` *condition*;
- rewrite rules: `rl [`l`]:` u `=>` v or `crl [`l`]:` u `=>` v `if` *condition*.

An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they have the form *var*`:`*sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

In object-oriented Maude specifications, a *class* declaration `class` C `|` att_1 `:` s_1`, ...,` att_n `:` s_n declares a class C of objects 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 `<` $O : C \mid att_1\colon\ val_1, \ldots, att_n\colon\ val_n$ `>`, 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`. A system state is modeled as a term of the sort `Config`, 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 (with label `l`)

```
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 `a2`, need not be mentioned in a rule; all such "superfluous" attributes can be replaced by a variable (such as `AS`) of sort `AttributeSet`, so that the above can also be written

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

Reachability Analysis in Maude. Maude provides a number of high-performance analysis methods, including rewriting for simulation purposes, reachability analysis, and linear temporal logic (LTL) model checking. In this paper, we use reachability analysis to model check consistency properties. Given an initial state *init*, a state pattern *pattern* and an (optional) condition *cond*, Maude's `search` command searches the reachable state space from *init* in a breadth-first manner for states that match *pattern* such that *cond* holds:

`search [`*bound*`]` *init* `=>!` *pattern* `such that` *cond* `.`

where *bound* provides an upper bound in the number of solutions to be found (if omitted, there is no such upper bound). The arrow `=>!` means that Maude only searches for reachable *final* states (i.e., states that cannot be further rewritten) that match *pattern* and satisfies *cond*. If the arrow used is instead `=>*` then Maude searches for all reachable states matching the search pattern and satisfying *cond*.

2.2. Statistical Model Checking and PVeStA

Distributed systems are probabilistic in nature, e.g., network latency such as message delay may follow a certain probability distribution, plus some algorithms may be probabilistic. Systems of this kind can be modeled by *probabilistic rewrite theories* [4] with rules of the form:

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

where the term t' has additional new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. Since for a given matching instance of the variables $\overrightarrow{x}$ there can be many (often infinite) ways to instantiate the extra variables $\overrightarrow{y}$, such a rule is *non-deterministic*. The probabilistic nature of the rule stems from the probability distribution $\pi(\overrightarrow{x})$, which depends on the matching instance of $\overrightarrow{x}$, and governs the probabilistic choice of the instance of $\overrightarrow{y}$ in the result $t'(\overrightarrow{x}, \overrightarrow{y})$ according to $\pi(\overrightarrow{x})$. In this paper we use the above PMaude [4] notation for probabilistic rewrite rules.

Statistical model checking [32, 36] is an attractive formal approach to analyzing probabilistic systems against temporal logic properties. Instead of offering a binary yes/no answer, it provides a quantitative real-valued answer and can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. The quantitative answer, however, need not be a percentage or a probability: it may instead be a latency estimation, or a quantitative estimation of some other performance property. For example, a statistical model checking result may be "86.87% ROLA transactions commit successfully with 95% confidence". Existing statistical verification techniques assume that the system model is purely probabilistic. Using the methodology in [4, 13] we can eliminate non-determinism in the choice of firing rules. We then use PVeStA [5], an extension and parallelization of the tool VeStA [33], to statistically model check purely probabilistic systems against properties expressed by QuaTEx probabilistic temporal logic [4]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ provided as input by sampling until the size of `(1-`α`)100%` confidence interval is bounded by δ, where the result of evaluating a formula is not a Boolean value, but a real number.

2.3. 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 [7], 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). 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 transactions use metadata and multi-versioning. Metadata is attached to each write, and the reads use this metadata to get the correct version. There are three versions of RAMP; in this paper we extend and modify 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.[2] 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 only sequentially increasing 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 of *versions*.

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.

The pseudo-code of RAMP-Fast in [7] is shown in Appendix A.

3. The ROLA Multi-Partition Transaction Algorithm

This section gives an informal overview of our proposed new algorithm, called ROLA, that guarantees both RA and PLU, but not CC, for transactions accessing multiple partitions in a setting where the data are *partitioned* (but not replicated) across a number of widely distributed sites.

ROLA extends RAMP-Fast to also ensure PLU. RAMP-Fast guarantees RA, but not 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

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

perspective by adding to each partition a map which maps each incoming version to an incremental sequence number. For example, if the transactions T_1 and T_2 both read version x_0 (with mapped sequence number 0) of the key x, and both try to write the respective versions x_1 and x_2. If T_1 manages to write x_1 first (with mapped sequence number 1), then T_2 is not allowed to overwrite x_1, since the local sequence number has increased by the time T_2 tries to write x_2. 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.

Algorithm 1 ROLA

```
    Server-side Data Structures
1:  $versions$: list of versions $\langle$item, value, timestamp $ts_v$, metadata $md\rangle$
2:  $latestCommit[i]$: last committed timestamp for item $i$
3:  $seq[ts]$: local sequence number mapped to timestamp $ts$
4:  $sqn$: local sequence counter

    Server-side Methods
    GET same as in RAMP-Fast (see Appendix A)

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

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

13: procedure COMMIT($ts_c$ : timestamp)
14:     $I_{ts} \leftarrow \{w.item \mid w \in versions \land w.ts_v = ts_c\}$
15:     for $i \in I_{ts}$ do
16:         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 (see Appendix A)

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

More specifically, ROLA adds two partition-side data structures: *sqn*, denoting the local sequence number 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 to the existing RAMP-F functionality: the coordinator-side[3] 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 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 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.

Example 1. Assume that we have two data items, x and y, and two partitions, P_x and P_y, storing x and y, respectively. As depicted in Figure 1, two read-write transactions $T_1 : r(y); w(x_1); w(y_1)$ and $T_2 : r(y); w(y_2)$ are attempting concurrent writes, and a read-only transaction $T_3 : r(x); r(y)$ proceeds while T_1 is writing. T_1 and T_2 read the same version y_0. Both T_1 and T_2 perform the two-phase commit protocol on two partitions, P_x and P_y. However, T_2 fails to prepare y_2 after T_1 has prepared y_1, because when T_2's prepare arrives at P_y, the timestamp of the last version store on P_y is 1, which is not equal to $ts_{prev} = 0$ in T_2's prepare. T_2, upon receiving the returned version y_1, could abort the transaction or retry with a new transaction on y_1. Either way the lost update problem is avoided. Regarding the case with T_1 and T_3, T_3 reads from P_x after P_x has committed T_1's write to x, but T_3 reads from P_y before P_y has committed T_1's write to y. Thus, T_3's first-round reads would violate RA if it returns them. Using the metadata attached to its first-round reads, T_3 determines to issue a second-round read to fetch the missing data from P_y. After completing the second-round read, T_3 can safely return T_1's writes, not violating RA. Note that in this example RAMP would allow T_2 to commit, thus overwriting T_1's writes, which are then lost.

4. Correctness Argument for ROLA

In this section we give a somewhat informal correctness argument/proof sketch for ROLA. Since Section 5 defines a *formal* model of ROLA, we could—and should in the future—formally prove that ROLA satisfies RA and PLU. However, such a full formal proof is beyond the scope of this paper.

4.1. Why ROLA Works

ROLA uses a two-phase commit protocol in order to detect concurrent writes. The first phase declares an intent to commit a write at the partition. Concurrent writes race to the partition without coordinating with each other. The partition can accept a preparation if there is no other prepared version after the latest commit associated with the incoming preparation. This in effect imposes a total order on the preparations, and thus on the commits, from the partition's perspective. In other words, the partition sees no logically concurrent updates. Our algorithm therefore provides read atomicity, and prevents updates from being lost, as concurrent updates are a necessary condition for lost updates.

By leveraging the partition-side sequence counter to commit, ROLA not only prevents lost updates, but

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

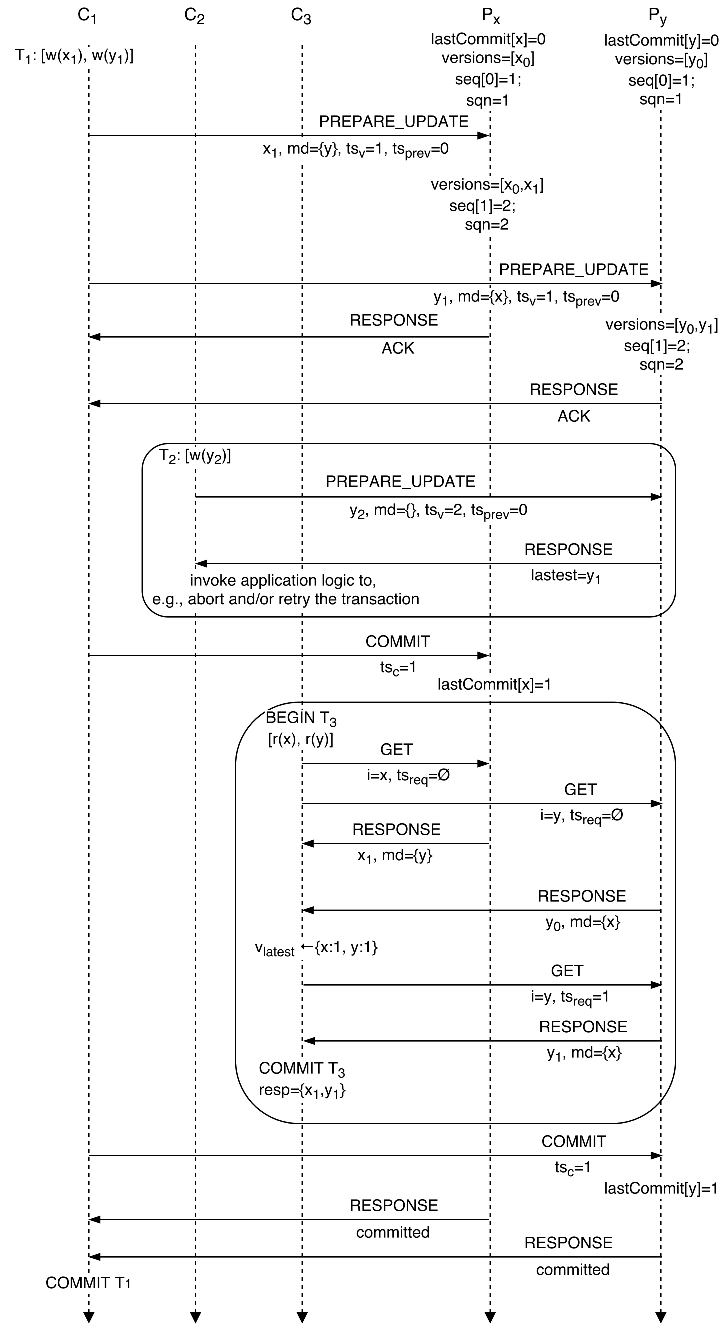

Figure 1. ROLA execution with three transactions in Example 1. Due to the space limit, we assume that T_1 and T_2 have fetched the same version y_0 (i.e., $ts_{prev} = 0$), when the sequence chart starts.

also makes writes progress at the partition-side, and thus more recent prepared version can be reflected (we refer to this as ROLA's *progress property*). This is different from RAMP-Fast, where later prepared writes may never be fetched by reads as *latestCommit* only updates by simply comparing the coordinator-side timestamps.

4.2. Proof Sketch

We consider transactions to be ordered sequences of reads and writes to arbitrary sets of data items. Each data item has a single logical copy. We call the set of data items a transaction reads and writes to its *read set* and *write set*, respectively. Each write creates a version of a data item. We identify versions of items by a timestamp from a totally ordered set (e.g., natural numbers) which is unique across all versions of each item. Thus timestamps induce a total order on versions of each item. We denote version i of item x as x_i. Given two versions x_i and x_j, we write $x_i < x_j$ if x_j appears later than x_i in the version order, and write $x_i <_{next} x_j$ if x_j is x_i's next version. Each item x has an initial version x_0. Each transaction finishes with being either committed or aborted. A *history* consists of a set of transactions, together with the versions the transactions read and wrote.

Our correctness argument for ROLA, like that for RAMP in [7], is based on Adya's formalization of consistency models [3]. Following the formal reasoning about RAMP in [7], we also use Adya's formalization in the context of the above system model.

There may be three types of dependencies: read-dependency, write-dependency and anti-dependency between two transactions.

Definition 4.1. (*Read-Dependency*). Transaction T_j directly read-depends on T_i if transaction T_j reads the version x_i that T_i has written.

Definition 4.2. (*Write-Dependency*). Transaction T_j directly write-depends on T_i if transaction T_i writes a version x_i and T_j writes x_i's next version x_j.

Definition 4.3. (*Anti-dependency*). Transaction T_j directly anti-depends on T_i if transaction T_i reads some version x_h, and T_j writes x_h's next version x_j.

Definition 4.4. (*Direct Serialization Graph*). A direct serialization graph (DSG) w.r.t. a history H, denoted by $DSG(H)$, is defined as:

- each node is the graph corresponds to a committed transaction;
- each directed edge corresponds to a type of direct dependency: there is a read-/write-/anti-dependency edge from T_i to T_j if T_j directly read-/write-/anti-depends on T_i.

In our model a transaction could also be a read-write transaction, in addition to read-only and write-only transactions, which are the only ones considered in [7].

We can formalize various anomalies for distributed transactions in terms of DSGs. These anomalies are then used to define consistency models.

Definition 4.5. (*G0: Write Cycles*). A history H exhibits phenomenon *G0* if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

Definition 4.6. (*G1a: Aborted Reads*). A history H exhibits phenomenon *G1a* if H contains an aborted transaction T_a and a committed transaction T_c such that T_c reads a version written by T_a.

Definition 4.7. (*G1b: Intermediate Reads*). A history H exhibits phenomenon *G1b* if H contains a committed transaction T_i that reads a version x_j written by T_j, and T_j also wrote a version x_k such that $j < k$.

Definition 4.8. (*G1c: Circular Information Flow*). A history H exhibits phenomenon $G1c$ if $DSG(H)$ contains a directed cycle that consists entirely of read-dependency and write-dependency edges.

Besides the above criteria, we still need the definition of *fractured reads* to define RA.

Definition 4.9. (*Fractured Reads*). A transaction T_j exhibits the *fractured reads* phenomenon if some transaction T_i writes versions x_a and x_b (in any order, where x and y may or may not be distinct items), and some transactions T_j reads versions x_a and y_c, and $c < b$.

As defined in [7]: RA isolation prevents fractured reads, and transactions from reading uncommitted, aborted, or intermediate versions:

Definition 4.10. (*Read Atomicity*). A system provides RA isolation if it prevents the phenomena $G0$, $G1a$, $G1b$, $G1c$, and fractured reads.

Lost updates (LU) happen when two transactions simultaneously make conditional modifications to the same data item(s).

Definition 4.11. (*Lost Updates*). A history H exhibits the phenomenon LU if $DSG(H)$ contains a directed cycle that consists of one or more anti-dependency edges and all edges are by the same data item.

To prove that ROLA provides RA, we must, according to Definition 4.10, prove that ROLA prevents the phenomena $G0$, $G1a$, $G1b$, $G1c$, and fractured reads. The proof is in general quite similar to that of RAMP providing RA (Appendix B in [7]), since ROLA reads are the same as RAMP reads, and ROLA writes are more restricted, thus decreasing the possibility of violating RA.

Lemma 4.1. ROLA prevents the phenomenon $G0$.

Proof. Each partition has a local sequence number that increases once a version is prepared; the increased sequence number is mapped to that version. Thus, versions (whether or not for the same item) on a partition are totally ordered. That is, there is no directed cycle consisting entirely of write-dependency edges. □

Lemma 4.2. ROLA prevents the phenomenon $G1a$.

Proof. ROLA first-round reads access *lastCommit*, so each version fetched by a first-round read is written by a committed transaction. ROLA second-round reads only access the versions in the same transaction as for the versions fetched by the first-round reads, which are also committed. Thus, ROLA never reads aborted writes. □

Lemma 4.3. ROLA prevents the phenomenon $G1b$.

Proof. The proof follows directly from Lemma 4.2. □

Lemma 4.4. ROLA prevents the phenomenon $G1c$.

Proof. Writes (on possibly different data items) in a transaction are assigned the same timestamp, which prevents read-dependency and write-dependency cycles. □

To prove ROLA preventing fractured reads, we first introduce *sibling versions*, *sibling item*, *companion version*, and *companion sets*.

Definition 4.12. (*Sibling Versions*). The set of versions produced by a transaction are called sibling versions.

Definition 4.13. (*Sibling Item*). Data item x is called a sibling item to a version y_j if there exists a version x_k written in the same transaction as y_j.

Definition 4.14. (*Companion Version*). Version x_i is a companion version of y_j if x_i is a sibling version of y_j or if the transaction that wrote y_j also wrote x_k and $i > k$.

Definition 4.15. (*Companion Sets*). A set of versions V is a companion set if, for every pair (x_i, y_j) of versions in V where x is a sibling item of y_j, x_i is a companion version of y_j.

Lemma 4.5. (*Atomicity of Companion Sets*). In the absense of $G1c$ phenomena, if the set of versions read by a transaction is a companion set, the transaction does not exhibit fractured reads.

Proof. If V is a companion set, then every version x_i in V is a companion version of every other version y_j in V that includes x in y_j's sibling items. Suppose V has fractured reads. According to Definition 4.9, there are two versions x_i and x_j such that the transaction that wrote y_j also wrote a version x_k with $i < k$. However, in this case x_i is not a companion version of y_j according to Definition 4.14. Therefore we reach a contradiction. □

Lemma 4.6. ROLA reads assemble a companion set.

Proof. Without loss of generality, suppose a transaction reads two versions x_i and y_j, and x is y_j's sibling item. The following continues the proof by comparing i and j:

- Case 1. If $i \geq j$, then x_i is already a companion version of y_j, and the set is therefore a companion set.
- Case 2. If $i < j$, then ROLA invokes RAMP-Fast's GET_ALL method to issue a second-round read to fetch the companion version x_j. Whether x_j has been committed or not by the time the second-round read reaches the partition, the ROLA partition invokes RAMP-Fast's GET method to return the prepared version x_j in *versions*.

Therefore, the resulting set of versions is a companion set. □

Theorem 4.7. ROLA guarantees RA.

Proof. The proof follows directly from Lemmas 4.1–4.6. □

To prove ROLA preventing LU, we must first prove some lemmas.

Lemma 4.8. Versions are ordered by the arrival order of the corresponding *prepare* messages.

Proof. The proof follows directly from Lemma 4.1. □

Lemma 4.9. Given a history H that is valid under ROLA, then each node in $DSG(H)$ directly read-depends on at most one other node with the same data item.

Proof. Suppose we have a transaction T_i in $DSG(H)$ that directly read-depends on two different transactions T_j and T_k with item x, namely $T_j \xrightarrow{r} T_i$ and $T_k \xrightarrow{r} T_i$. By Lemma 4.8, $x_j < x_k$ or $x_k < x_j$. In either case T_i exhibits fractured reads and H is not valid under RA (and thus ROLA), a contradiction. □

Lemma 4.10. If a (read-write) transaction reads version x_i, and then writes version x_j, then $x_i <_{next} x_j$.

Proof. Since x_j has been prepared, it means that there was no prepared versions between x_i and x_j; otherwise the condition cannot be satisfied. Thus $x_i <_{next} x_j$. □

Lemma 4.11. Given a history H that is valid under ROLA. Each node in $DSG(H)$ is then directly write-dependent on at most one other node with the same data item.

Proof. Suppose we have a transaction T_i in $DSG(H)$ that is directly write-dependent on two different transactions T_j and T_k with item x, namely, $T_i \xrightarrow{w} T_j$ and $T_i \xrightarrow{w} T_k$.

- Case 1. T_j and T_k are both write-only transactions. Say T_j prepares first, and we have $seq[x_j] < seq[x_k]$ because when prepared, the sequence number increases.
  - Case 1.1. If T_j also commits first, T_j overwrites T_i and we have $T_i \xrightarrow{w} T_j$. Currently, $latestCommit[x] = ts_j$. When T_k commits later, $latestCommit[x]$ is mapped to ts_k because $seq[ts_k] > seq[ts_j]$. Then we have $T_j \xrightarrow{w} T_k$, a contradiction.
  - Case 1.2. If T_k commits first, T_k overwrites T_i and we have $T_i \xrightarrow{w} T_k$. Currently, $latestCommit[x] = ts_k$. When T_j commits later, $latestCommit[x]$ is not updated because $seq[ts_j] < seq[ts_k]$. Then we only have $T_i \xrightarrow{w} T_k$, a contradiction.
- Case 2. If T_j is a read-write transaction while T_k is a write-only transaction. According to Lemma 4.10, T_j reads T_i. Thus T_k cannot prepare first because in that case T_j aborts. The proof then follows directly from the above Case 1.1.
- Case 3. If T_j and T_k are both read-write transactions. According to Lemma 4.10, both T_j and T_k read T_i. Either one prepares first, and the other one has to abort (the later prepare's ts_{prev} does not match the latest version's, i.e., the other prepare's, timestamp).

□

Lemma 4.12. Given a history H that is valid under ROLA. Then $DSG(H)$ does not contain a sequence $T_h \xrightarrow{anti} T_i \xrightarrow{anti} T_j$.

Proof. There is a transaction $T_{i'}$ whose version written $x_{i'}$ is read by T_i, namely $T_{i'} \xrightarrow{r} T_i$. T_j writes $x_{i'}$'s next version x_j, namely $T_{i'} \xrightarrow{w} T_j$. According to Lemma 4.10 and Lemma 4.11, we have $T_{i'} \neq T_h$. From $T_h \xrightarrow{anti} T_i$ we can build another two dependencies $T_{h'} \xrightarrow{r} T_h$ and $T_{h'} \xrightarrow{w} T_i$. We now have $x_{i'} <_{next} x_i$ and $x_{h'} <_{next} x_i$. According to Lemma 4.8 we must have $x_{i'} = x_{h'}$, and then we have $T_{i'} \xrightarrow{w} T_i$. Since we already have $T_{i'} \xrightarrow{w} T_j$, we reach a contradiction (Lemma 4.11). □

Lemma 4.13. Given a history H that is valid under ROLA. Then $DSG(H)$ does not contain a sequence $T_h \xrightarrow{w} T_i \xrightarrow{anti} T_j$.

Proof. There is a transaction $T_{i'}$ whose version written $x_{i'}$ is read by T_i, namely $T_{i'} \xrightarrow{r} T_i$. T_j writes $x_{i'}$'s next version x_j, namely $T_{i'} \xrightarrow{w} T_j$. According to Lemma 4.11, $T_h \neq T_{i'}$. Now we have $x_h <_{next} x_i$ and $x_{i'} <_{next} x_j$. The following continues the proof by comparing the version order of x_h and $x_{i'}$:

- Case 1. If $x_h < x_{i'}$, we have $x_i < x_{i'}$ since x_h's next version is x_i. According to Lemma 4.10, we have $x_{i'} <_{next} x_i$. Therefore we reach a contradiction.
- Case 2. If $x_{i'} < x_h$, because of $x_{i'} <_{next} x_j$, we have $x_{i'} <_{next} x_j < x_h <_{next} x_i$. According to Lemma 4.10, we have $x_{i'} <_{next} x_i$. Therefore we reach a contradiction.

□

Theorem 4.14. ROLA prevents LU.

Proof. We prove this theorem by contradiction. Suppose there is a history H that is valid under ROLA, so that $DSG(H)$ contains a directed cycle having one or more anti-dependency edges and all edges are labeled by the same data item x.

- Case 1. Suppose $DSG(H)$ contain a directed cycle having only one anti-dependency edge $T_a \xrightarrow{l1} ... \xrightarrow{l2} T_h \xrightarrow{l_{hi}} T_i \xrightarrow{l_{ij}} T_j \xrightarrow{l3} ... \xrightarrow{l4} T_a$. Let l_{ij} be the anti-dependency edge. Thus, there is a transaction $T_{i'}$ whose version written $x_{i'}$ is read by T_i, namely $T_{i'} \xrightarrow{r} T_i$; the transaction T_j writes $x_{i'}$'s next version x_j, namely $T_{i'} \xrightarrow{w} T_j$. According to Lemma 4.13 and Lemma 4.9, we have $T_{i'} = T_h$. Thus, we have another directed cycle $T_a \xrightarrow{l1} ... \xrightarrow{l2} T_h \xrightarrow{w} T_j \xrightarrow{l3} ... \xrightarrow{l4} T_a$ consisting entirely of read-dependency and write-dependency edges ($G1c$), which contradicts Theorem 4.7.
- Case 2. Suppose $DSG(H)$ contain a directed cycle having at least two anti-dependency edges. According to Lemma 4.12, these anti-dependency edges are not consecutive. For each anti-dependency we directly follow Case 1, and eventually we are able to construct a directed cycle consisting entirely of read-dependency and write-dependency edges ($G1c$), which contradicts Theorem 4.7.

□

5. A Formal Executable Specification of ROLA

This section presents an executable formal specification of ROLA in rewriting logic. Besides providing an unambiguous formal model of ROLA, this executable formal model can be used for both model checking analysis of the RA and PLU properties, and for performance analysis by statistical model checking.

To introduce both time and probabilities for *performance* estimation, each message gets assigned a *message delay* that is sampled probabilistically from a dense time interval according to a certain probability distribution. To analyze the *correctness* of ROLA, we just set all the message delays to zero, thereby obtaining an untimed nondeterministic model, that can subjected to standard model checking, from our probabilistic timed model. The whole specification is given at `https://sites.google.com/site/siliunobi/fac-rola`.

5.1. 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). Such a ripe message must be consumed by the receiver *rcvr* before time advances. The delay Δ can be sampled from certain distributions (lognormal, Weibull, etc.) for statistical model checking, or, as mentioned above, removed (or set to zero) for correctness analysis.

To sample message delays from different distributions, we use the following functionality provided by Maude: The built-in 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] .
eq sampleNormal = sampleNormal(float(random(counter) / 4294967296)) .

op sampleNormal : Float -> [Float] .
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) .
```

Each time `rd` rewrites, it therefore rewrites to a different delay value according to the above distribution.

5.2. 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.

5.2.1. Data Types

A *version* is a timestamped version of a data item (or key) and is modeled as a four-tuple `version(`*key*`,`*value*`,` *timestamp*`,`*metadata*`)` consisting of the key, its value, and the version's timestamp and metadata. A timestamp is modeled as a pair `ts(`*addr*`,`*sqn*`)` consisting of a partition's identifier *addr* and a local sequence number *sqn* that together uniquely identify a write transaction. Metadata are modeled as a set of keys, denoting, for each key, the other keys that are written in the same transaction. For example, if a transaction writes keys x, y, and z, then versions of x have as metadata the set $\{y, z\}$.

```
sorts Key Value Timestamp Version KeySet Versions KeyTimestampEntry KeyTimestamps .
subsort Key < KeySet .
subsort KeyTimeEntry < KeyTimestamps .
subsort Version < Versions .

op ts : Address Nat -> Timestamp .
op version : Key Value Timestamp KeySet -> Version [ctor] .

op empty : -> KeySet [ctor] .
op _,_ : KeySet KeySet -> KeySet [ctor assoc comm id: empty] .
```

```
  op empty : -> KeyTimestamps [ctor] .
  op _|->_ : Key Timestamp -> KeyTimestampEntry [ctor] .
  op _,_ : KeyTimestamps KeyTimestamps -> KeyTimestamps [ctor assoc comm id: empty] .

  op nil : -> Versions [ctor] .
  op __ : Versions Versions -> Versions [ctor assoc id: nil] .
```

A set of keys of sort `KeySet` are built from the constant `empty` and singleton sets (identified with keys of sort `Key` by means of a `subsort` declaration) with an associative, commutative, and idempotent union operator `_,_`. A set of entries, or mappings from keys to timestamps, of sort `KeyTimestamps` are built from the constant `empty` and singleton sets (identified with entries of sort `KeyTimestampEntry` by means of a `subsort` declaration) with an associative, commutative, and idempotent union operator `_,_`. A list of versions of sort `Versions` are built from the constant `nil` and singleton lists (identified with versions of sort `Version` by means of a `subsort` declaration) with an associative, and idempotent union operator `__`.

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:

```
  sorts Expression LocalVar LocalVarEntry LocalVars Operation .
  subsort Operation < OperationList .
  subsort LocalVarEntry < LocalVars .

  op write : Key Expression -> Operation [ctor] .
  op _:=read_ : LocalVar Key -> Operation [ctor] .

  op nil : -> OperationList [ctor] .
  op __ : OperationList OperationList -> OperationList [ctor assoc id: nil] .

  op empty : -> LocalVars [ctor] .
  op _|->_ : LocalVal Value -> LocalVarEntry [ctor] .
  op _,_ : LocalVars LocalVars -> LocalVars [ctor assoc comm id: empty] .
```

A list of operations of sort `OperationList` are built from the constant `nil` and singleton lists (identified with operations of sort `Operation` by means of a `subsort` declaration) with an associative, and idempotent union operator `__`. A set of entries, or mappings from local variables to their values, of sort `LocalVars` are built from the constant `empty` and singleton sets (identified with entries of sort `LocalVarEntry` by means of a `subsort` declaration) with an associative, commutative, and idempotent union operator `_,_`.

We define a collection of votes of sort `Vote` as a "set" (build with the associative and commutative operator `_;_`) of votes, where each vote, as a triple `vote(`*txn*`,`*part*`,`*result*`)`, indicates the voting result by some partition for a certain transaction:

```
  sort Vote .
  op noVote : -> Vote [ctor] .
  op vote : Tid Addr Bool -> Vote [ctor] .
  op _;_ : Vote Vote -> Vote [ctor assoc comm id: noVote] .
```

The data type `TxnAddrSet` is defined for the situation when a partition is waiting for messages such as votes from other partitions w.r.t. some transaction:

```
  sorts AddrSet TxnAddrSet .
  subsort Address < AddrSet .

  op empty : -> AddrSet [ctor] .
  op _,_ : AddrSet AddrSet -> AddrSet [ctor assoc comm id: empty] .

  op empty : -> TxnAddrSet [ctor] .
  op addrs : Tid AddrSet -> TxnAddrSet [ctor] .
  op _;_ : TxnAddrSet TxnAddrSet -> TxnAddrSet [ctor assoc comm id: empty] .
```

where AddrSet is defined as a "set" of replicas (of sort Address), while TxnAddrSet a mapping from transactions (of sort Tid) to sets of replicas.

5.2.2. 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 each key to 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. A partition executes transactions sequentially. Concurrent transactions can be modeled by multiple transactions executed by different partitions.

The attribute votes stores the votes from the partitions which participate 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 state also contains a "table" mapping each data item to the partition storing the item. Each mapping is a pair sites(*key*,*part*). The table is built with the associative and commutative operator _;;_ of mappings.

The following shows an initial state (with some parts replaced by '...') with two partitions, p1 and p2, that are coordinators for, respectively, transactions t1, 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}    {0.0, p1 <- start}    {0.0, p2 <- start}
< 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, executing: noTxn, committed: none, aborted: none, tsSqn: empty,
        latestCommit: empty, votes: noVote, voteSites: empty, 1stGetSites: empty,
        2ndGetSites: empty, commitSites: empty >
< 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), ... >
```

where {0.0 | nil} denotes the "scheduler" which holds the list of messages sent but not yet delivered, ordered according to their time of delivery (see [4]), with the initial global time `0.0` and the empty list of messages to be scheduled. The `start` messages are used to trigger the execution of the first transaction at each partition.

5.2.3. Messages

As explained in Section 5.1, we have two types of messages `[`Δ`,`*rcvr* `<-` *msg*`]` and `{`T`,`*rcvr* `<-` *msg*`}` with *msg* the message content of sort `Content`, and *rcvr* the identifier of the receiving object:

```
  op _<-_ : Address Content -> Msg .
  op [_,_] : Float Msg -> Msg .
  op {_,_} : Float Msg -> Msg .
```

where the current time Δ and the message delay T are of sort `Float` of floating-point numbers. The following presents all message contents used in our ROLA model:

- `prepare(`*txn*`,`*version*`,` *sender*`)` sends a version from a write-only transaction to its partition;
- `prepare(`*txn*`,`*version*`,` *ts*`,`*sender*`)` does the same thing for other transactions, with *ts* the timestamp of the version it has read;
- `prepare-reply(`*txn*`,`*vote*`,`*sender*`)` is the reply to the corresponding prepare message, where *vote* tells whether this partition can commit the transaction;
- `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*;
- `response1(`*txn*`,`*version*`,`*sender*`)` responds to first-round `get` request;
- `response2(`*txn*`,`*version*`,`*sender*`)` responds to second-round `get` requests;
- `commit-reads` commits read-only transactions, or completes the reads in read-write transactions; and
- `start` triggers a partition to start executing transactions.

5.3. Formalizing ROLA's Behaviors

This section formalizes the dynamic behaviors of ROLA using rewrite rules, referring to the corresponding lines in Algorithm 1.[4]

Starting a Transaction (Lines 17 – 22). Triggered by a `start` message, a partition starts executing a transaction by moving the first transaction (`TID`) in `gotTxns` to `executing`, if the partition is not currently already executing a transaction (`executing` is `noTxn`). If the new transaction is a write-only transaction (`write-only(OPS)`), the partition: (i) uses the function `genPuts` to generate all `prepare` messages; (ii) uses a function `prepareSites` to remember the sites `PIDS` from which it awaits votes for transaction `TID` in the `voteSites` attribute; and (iii) increments its local sequence number by one:[5]

```
crl [start-wo-txn] :
    {T, PID <- start}
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition |
              gotTxns: (< TID : Txn | operations: OPS, localVars: VARS, AS > ;; TXNS),
              executing: noTxn,   sqn: SQN,   voteSites: VSTS,  AS' >
   =>
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition |
              gotTxns: TXNS,
              executing: < TID : Txn | operations: OPS, localVars: VARS, AS >,
```

[4] We do not include variable declarations, but follow the convention that variables are written in (all) capital letters.
[5] The variables `AS` and `AS'` denote the "remaining" attributes in the two objects.

```
              sqn: SQN',   voteSites: (VSTS ; addrs(TID,PIDS)),  AS' >
    genPuts(OPS,PID,TID,SQN',VARS,PARTITION-TABLE)
    if SQN' := SQN + 1 /\  write-only(OPS) /\
       PIDS := prepareSites(OPS,PID,PARTITION-TABLE) .
```

The above function genPuts is defined as follows:

```
op genPuts : OperationList Address Address Nat LocalVars ReplicaTable -> Config .
op $genPuts : OperationList Address Address Nat LocalVars ReplicaTable
                                                           OperationList -> Config .
eq genPuts(OPS,PID,TID,SQN,VARS,PARTITION-TABLE)
 = $genPuts(OPS,PID,TID,SQN,VARS,PARTITION-TABLE,OPS) .
eq $genPuts((write(K,EXPR) OPS),PID,TID,SQN,VARS,PARTITION-TABLE,(OPS' write(K,EXPR) OPS''))
 = $genPuts(OPS,PID,TID,SQN,VARS,PARTITION-TABLE,(OPS' write(K,EXPR) OPS''))
      (if localReplica(K,PID,PARTITION-TABLE)
       then [ld, PID <-
           prepare(TID,version(K,eval(EXPR,VARS),ts(PID,SQN),md(OPS' OPS'')),PID)]
       else [rd, preferredSite(K,PARTITION-TABLE) <-
           prepare(TID,version(K,eval(EXPR,VARS),ts(PID,SQN),md(OPS' OPS'')),PID)]
       fi) .
eq $genPuts(((X :=read K) OPS),PID,TID,SQN,VARS,PARTITION-TABLE,OPS')
 = $genPuts(OPS,PID,TID,SQN,VARS,PARTITION-TABLE,OPS') .
eq $genPuts(nil,PID,TID,SQN,VARS,PARTITION-TABLE,OPS') = null .
```

Otherwise, if the first transaction in gotTxns is a read-only or read-write transaction, the replica updates 1stGetSites instead to keep track of the replicas from which it receives the versions from the first-round gets. The function genGets generates all get messages for the keys concerned by TID (see the executable specification available online for the definition of this, and other, functions). The expression 1stSites gives the corresponding replicas for those keys:

```
crl [start-ro-or-rw-txn] :
    {T, PID <- start}
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | gotTxns:
                            (< TID : Txn | operations: OPS, latest: empty, AS > ;; TXNS),
                        executing: noTxn,
                        1stGetSites: 1STGETS, AS' >
  =>
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | gotTxns: TXNS,
                        executing: < TID : Txn | operations: OPS, latest: vl(OPS), AS >,
                        1stGetSites: (1STGETS ; addrs(TID,PIDS)), AS' >
    genGets(OPS,PID,TID,PARTITION-TABLE)
    if (not write-only(OPS)) /\
       PIDS := 1stSites(OPS,PID,PARTITION-TABLE) .
```

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:

```
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) .
```

If instead the received `prepare` message was for a write-only transaction, the replica simply adds the received version to its local datastore, and maps the associated timestamp to the incremented sequence number (`insert(TS,SQN',TSSQN)`). Depending on whether the message sender `PID'` is the replica itself, the out-going message is equipped with a local message delay `ld` or a remote message delay `rd`. Both delays, as mentioned above, are sampled from certain distributions. Note that case (i) always considers successful preparations.

```
crl [receive-prepare-wo] :
    {T, PID <- prepare(TID,version(K,V,TS,MD),PID')}
    < PID : Partition | datastore: VS,
                        sqn: SQN,
                        tsSqn: TSSQN, AS' >
  =>
    < 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)]
    if SQN' := SQN + 1 .
```

In case (ii), the replica first determines whether the timestamp of the last `VERSION` in the local version list `VS` matches the incoming timestamp `TS'` which is the timestamp associated with the version fetched by the previous get in the same read-write transaction. If matched, the incoming version is added to the local datastore; otherwise, a negative ack (denoted by `false`) is sent back:

```
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. If the transaction has been aborted, the incoming `prepare-reply` message is simply consumed by the replica:

```
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' >
    [0.0, PID <- start] .

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

where, in the first case, a `start` message without delay is sent to the partition itself to trigger the execution of next transaction.

Receiving Acks (Lines 26–28). Upon receiving a "true" vote (`prepare-reply(...,true,...)`), the replica first checks whether all votes have now been collected. The expression `VSTS'[TID]` projects for `TID` the remaining replicas from which it is awaiting votes. If all received votes are "yes," the replica starts to commit `TID` at the associated replicas by invoking `genCommits` to generate all commit messages with the commit timestamp including the current sequence number `SQN`. The replica also adds to `commitSites` the replicas from which it is awaiting `committed` messages to commit the transaction:

```
crl [receive-prepare-reply-true-executing] :
    {T, PID <- prepare-reply(TID,true,PID')}
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | executing: < TID : Txn | operations: OPS, AS >,
                        voteSites: VSTS, sqn: SQN,
                        commitSites: CMTS, AS' >
  =>
    < TABLE : Table | table: PARTITION-TABLE >
    if VSTS'[TID] == empty  --- all votes received and all yes!
    then < PID : Partition | executing: < TID : Txn | operations: OPS, AS >,
                             voteSites: VSTS', sqn: SQN,
                             commitSites: (CMTS ; addrs(TID,PIDS)), AS' >
         genCommits(TID,SQN,PIDS,PID)
    else < PID : Partition | executing: < TID : Txn | operations: OPS, AS >,
                             voteSites: VSTS', sqn: SQN,
                             commitSites: CMTS, AS' >
    fi
    if VSTS' := remove(TID,PID',VSTS) /\
       PIDS := commitSites(OPS,PID,PARTITION-TABLE) .
```

Receiving `commit` Messages (Lines 13–16). Upon receiving a `commit` message, the partition invokes the function `cmt` to commit the transaction. `cmt` looks up `tsSqn` for the commit timestamp `TS` and the latest committed version's timestamp in `LC`, and updates the latest committed version if `TS`'s local sequence number is higher. A `committed` message is then sent back to confirm the commit:

```
rl [receive-commit] :
   {T, PID <- commit(TID,TS,PID')}
   < PID : Partition | tsSqn: TSSQN, datastore: VS, latestCommit: LC, AS' >
  =>
   < PID : Partition | tsSqn: TSSQN, datastore: VS,
                       latestCommit: cmt(LC,VS,TSSQN,TS), AS' >
   [if PID == PID' then ld else rd fi, PID' <- committed(TID,PID)] .
```

Receiving `committed` Messages. (For partitions to commit transactions locally). Upon receiving a `committed` message, the replica first checks if all committed messages have now been collected. The expression `CMTS'[TID]` projects for `TID` the remaining replicas from which it is awaiting `committed` messages. If the projection is `empty`, the replica commits the transaction:

```
crl [receive-committed] :
    {T, PID <- committed(TID,PID')}
    < PID : Partition | executing: < TID : Txn | AS >,
                        committed: TXNS, commitSites: CMTS, AS' >
  =>
    if CMTS'[TID] == empty  --- all "committed" received
    then < PID : Partition | executing: noTxn,
                             committed: (TXNS ;; < TID : Txn | AS >),
                             commitSites: CMTS', AS' >
         [0.0, PID <- start]
    else < PID : Partition | executing: < TID : Txn | AS >,
                             committed: TXNS,
                             commitSites: CMTS', AS' >
    fi
    if CMTS' := remove(TID,PID',CMTS) .
```

where a `start` message is issued out for the execution of next transaction.

Receiving get Messages (Lines 9–13 in RAMP-Fast, Appendix A). Upon receiving a `get` message, depending on the associated timestamp `TS` (if `TS` is an empty timestamp `eptTS`, the incoming message is the first-round `get`; otherwise, it is the second-round `get`), the replica replies with the corresponding version determined by the function `vmatch`. For a first-round `get`, `vmatch` looks up `LC` for the latest committed version; for the second-round `get`, `vmatch` returns the matched timestamped version of `TS`:

```
rl [receive-get] :
   {T, PID <- get(TID,K,TS,PID')}
   < PID : Partition | datastore: VS, latestCommit: LC, AS' >
 =>
   < PID : Partition | datastore: VS, latestCommit: LC, AS' >
   [if PID == PID' then ld else rd fi,
    PID' <- (if TS == eptTS then  response1(TID,vmatch(K,VS,LC),PID)
    else response2(TID,vmatch(K,VS,TS),PID) fi)] .
```

Receiving Replies to First-Round Gets (Lines 27–32 in RAMP-Fast, Appendix A). Upon receiving a returned version for the first-round get, the replica adds it to the read set, and updates `localVars` accordingly. When the replica has collected all replies to the first-round gets, it determines whether a second-round `get` is needed. The expression `gen2ndGets(TID,VL',RS',PID,PARTITION-TABLE)` generates possible second-round `get` messages based on the updated `latest`, `VL'`, and `readSet`, `RS'`. In case a second-round `get` is not needed, `gen2ndGets` generates no message (and thus the `commit-reads` message will trigger the partition to execute next transaction), and `PIDS` is an `empty` set. Note that `RS'` is needed if `TID` is a read-write transaction:

```
crl [receive-response1] :
    {T, PID <- response1(TID,version(K,V,TS,MD),PID')}
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | executing:
```

```
                             < TID : Txn | operations: (OPS (X :=read K) OPS'),
                                           readSet: RS, localVars: VARS, latest: VL, AS >,
                         1stGetSites: 1STGETS,
                         2ndGetSites: 2NDGETS, AS' >
  =>
    < TABLE : Table | table: PARTITION-TABLE >
    if 1STGETS'[TID] == empty
    then < PID : Partition | executing:
                                  < TID : Txn | operations: (OPS (X :=read K) OPS'),
                                                readSet: RS', localVars: insert(X,V,VARS),
                                                latest: VL', AS >,
                              1stGetSites: 1STGETS',
                              2ndGetSites: (2NDGETS ; addrs(TID,PIDS)), AS' >
         gen2ndGets(TID,VL',RS',PID,PARTITION-TABLE)
         [0.0, PID <- commit-reads]
    else < PID : Partition | executing:
                                  < TID : Txn | operations: (OPS (X :=read K) OPS'),
                                                readSet: RS', localVars: insert(X,V,VARS),
                                                latest: VL', AS >,
                              1stGetSites: 1STGETS',
                              2ndGetSites: 2NDGETS, AS' >
    fi
    if RS' := RS version(K,V,TS,MD) /\
       VL' := lat(VL,MD,TS) /\
       1STGETS' := remove(TID,PID',1STGETS) /\
       PIDS := 2ndSites(VL',RS',PID,PARTITION-TABLE) .
```

Receiving Replies to Second-Round Gets (Lines 32–33 in RAMP-Fast, Appendix A). Upon receiving a returned version for the second-round get, the partition simply overwrites the version fetched by the first-round get (the `readSet` is updated). It then updates the local variables `localVars` and the remaining replicas from which it is awaiting second-round `get`s:

```
rl [receive-response2] :
   {T, PID <- response2(TID,version(K,V,TS,MD),PID')}
   < PID : Partition | executing:
                            < TID : Txn | operations: (OPS (X :=read K) OPS'),
                                          readSet: (RS version(K,V',TS',MD') RS'),
                                          localVars: VARS, AS >,
                        2ndGetSites: 2NDGETS, AS' >
 =>
   < PID : Partition | executing:
                            < TID : Txn | operations: (OPS (X :=read K) OPS'),
                                          readSet: (RS version(K,V,TS,MD) RS'),
                                          localVars: insert(X,V,VARS), AS >,
                        2ndGetSites: remove(TID,PID',2NDGETS), AS' >
    [0.0, PID <- commit-reads] .
```

where a `commit-reads` message is issued out for the execution of next transaction.

Committing Reads (Lines 18–22). Upon receiving the `commit-reads` message, if the replica has no remaining replicas from which it is awaiting replies to either first-round gets or second-round gets (`1STGETS[TID] == empty` and `2NDGETS[TID] == empty`), it starts to commit the reads. There are two cases to consider: (i) a read-only transaction; or (ii) a read-write transaction.

In case (i), the replica simply puts `TID` in `committed`, and sends out a `start` message to start executing the next transaction; in case (ii), the replica further generates all prepare messages for each version concerned with newly generated timestamp including the incremented sequence number `SQN'`. The prepared versions are computed based on the previously fetched reads reflected in `VARS`, and the prepare messages also include the timestamps of the previously fetched reads in `RS`:

```
crl [commit-reads] :
    {T, PID <- commit-reads}
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | executing: < TID : Txn | operations: OPS,
                                                 localVars: VARS,
                                                 readSet: RS, AS >,
                        committed: TXNS, 1stGetSites: 1STGETS,
                        2ndGetSites: 2NDGETS, sqn: SQN, voteSites: VSTS, AS' >
 =>
   < TABLE : Table | table: PARTITION-TABLE >
   if read-only(OPS)
   then < PID : Partition | executing: noTxn,
                            committed: (TXNS ;;
                                        < TID : Txn | operations: OPS, localVars: VARS,
                                                      readSet: RS, AS >),
                            1stGetSites: 1STGETS, 2ndGetSites: 2NDGETS,
                            sqn: SQN, voteSites: VSTS, AS' >
        [0.0, PID <- start]
   else < PID : Partition | executing: < TID : Txn | operations: OPS,
                                                     localVars: VARS, readSet: RS, AS >,
                            committed: TXNS, 1stGetSites: 1STGETS,
                            2ndGetSites: 2NDGETS, sqn: SQN',
                            voteSites: (VSTS ; voteSites(TID,PIDS)), AS' >
        genPuts(OPS,PID,TID,SQN',VARS,RS,PARTITION-TABLE)
   fi
   if 1STGETS[TID] == empty /\ 2NDGETS[TID] == empty /\
      SQN' := SQN + 1 /\ PIDS := prepareSites(OPS,PID,PARTITION-TABLE) .
```

6. Model Checking Correctness Properties of ROLA

Section 4 gives an informal “proof” that ROLA guarantees RA and PLU. However, it is well known that such “hand proofs” may be erroneous or may make crucial assumptions that are not made explicit. Indeed, we have experienced that Maude model checking can uncover nontrivial errors as well as both missing and unclear assumption in a supposedly verified distributed transaction system that is less complex than ROLA [30]. To gain further confidence in the correctness of ROLA—before undertaking the hard task of formally verifying ROLA—we therefore use Maude model checking to analyze ROLA’s correctness.

This section shows how Maude can be used to formalize the RA and PLU requirements, and how Maude reachability analysis can be used to analyze whether or not ROLA guarantees RA and PLU. We also formalize and analyze whether or not ROLA satisfies CC. In particular, we add to the state a monitor object which records relevant history during a run of the protocol. We then formalize in Maude what it means that such a history satisfies RA, PLU, and CC. Finally, we use Maude reachability analysis to analyze whether all possible runs starting from four different concrete initial states satisfy the three properties.

6.1. Recording the History of a Run

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 collection of records, with one record for each transaction, where each record has the form `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. We show three examples below.

Start a Transaction. When the coordinator starts to execute a transaction, the monitor appends a new record for that transaction `TID` with the initial values `T` for the *issueTime*, `0.0` for *finishTime*, `empty` for the read and write sets, and with the committed flag set to `false`.[6] The global time `T` is obtained from the delivery time of the received `start` message:

```
crl [start-ro-or-rw-txn] :
    < M : Monitor | log: LOG >
    {T, PID <- start}
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | gotTxns: (< TID : Txn | operations: OPS,
                                                latest: empty, AS > ;; TXNS),
                        executing: noTxn,
                        1stGetSites: 1STGETS, AS' >
  =>
    < M : Monitor | log: (LOG ; record(TID,T,0.0,empty,empty,false)) >
    < TABLE : Table | table: PARTITION-TABLE >
    < PID : Partition | gotTxns: TXNS,
                        executing: < TID : Txn | operations: OPS,
                                                 latest: vl(OPS), AS >,
                        1stGetSites: (1STGETS ; addrs(TID,RIDS)), AS' >
    genGets(OPS,RID,TID,PARTITION-TABLE)
    if (not write-only(OPS)) /\
       RIDS := 1stSites(OPS,RID,PARTITION-TABLE) .
```

Commit a Transaction. 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`:

```
crl [receive-committed] :
    < M : Monitor | log: (LOG ; record(TID, T', T'', RS, WS, false) ; LOG') >
    {T, PID <- committed(TID,PID')}
    < 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) .
```

Abort a Transaction. When the coordinator receives a `false` vote, it aborts the transaction. The monitor records the abort/finish time `T` for that transaction (and the "committed" flag remains `false`):

```
rl [receive-prepare-reply-false-executing] :
   < M : Monitor | log: (LOG ; record(TID,T1,0.0,RS,WS,false) ; LOG') >
   {T, RID <- prepare-reply(TID,false,RID')}
   < PID : Partition | executing: < TID : Txn | AS >,
                       aborted: TXNS,
                       voteSites: VSTS, AS' >
 =>
   < M : Monitor | log: (LOG ; record(TID,T1, T,RS,WS,false) ; LOG') >
```

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

```
   < PID : Partition | executing: noTxn,
                       aborted: (TXNS ;; < TID : Txn | AS >),
                       voteSites: remove(TID,RID',VSTS), AS' > .
```

6.2. Formalizing and Analyzing RA, PLU, and CC

Since ROLA is terminating if only 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. That is, we use reachability analysis to check whether there exists a final state where the log shows that the run violated the desired property.

6.2.1. Read Atomicity

A system guarantees RA if it prevents fractured reads, and also prevents transactions from reading uncommitted, aborted, or intermediate data [7]. 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 $k < n$ [7].

We analyze this property by searching for a reachable *final* state (arrow `=>!`) where the property does *not* hold, which is done with the following Maude command:

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

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`. The first equation handles the case when `TID1` comes before `TID2` in the log, and the second equation treats the opposite case:

```
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] .
```

6.2.2. Prevention of Lost Updates

We analyze the PLU property by searching for a final state in which the monitor shows that an update was lost:

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

The function `lu` checks whether there are lost updates in `LOG`.

Lost updates happen when two transactions simultaneously make conditional updates to the same data item(s). Specifically, when the read sets of the two transactions overlap on data item x (meaning that both have fetched the same data of x), and x is in the write sets of the both transactions (meaning that both try to modify x), one update will be overwritten by the other, and thus an update is lost. Our specification of `lu` captures this by checking whether there are two transactions in *log* reading the same data (matched by `version(X,VX,TSX,MDX)` in the read set), and they both commit their writes on that key (matched by the key `X` in the write set):

```
op lu : Record -> Bool .
eq lu(LOG ; record(TID1,T1,T1',(version(X,VX,TSX,MDX),VS1),
                            (version(X,VX1,TSX1,MDX1),VS3),true) ; LOG' ;
            record(TID2,T2,T2',(version(X,VX,TSX,MDX),VS2),
                            (version(X,VX2,TSX2,MDX2),VS4),true) ; LOG'') = true .
eq lu(LOG) = false [owise] .
```

6.2.3. Causal Consistency

We analyze the CC property by searching for a final state in which the monitor shows a violation of causal consistency:

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

The function `notCausal` checks whether there are transactions not respecting the causal order. The transaction `T2` reads `T1`'s write (matched by `version(X,VX,TSX,MDX)`, so `T2` causally depends on `T1`. Similarly, the transaction `T3` reads `T2`'s write (matched by `version(Y,VY',TSY',MDY')`, so `T3` causally depends on `T2`. Since causality is transitive, `T3` must causally depend on `T1`. However, `T3` reads a different version `VX'`, thus violating causality:

```
 op notCausal : Record -> Bool .
ceq notCausal(LOG1 ; record(TID1,T1,T1',RS,(version(X,VX,TSX,MDX),
        version(Y,VY,TSY,MDY),VS1),true) ; LOG2 ;
            record(TID2,T2,T2',(version(X,VX,TSX,MDX),VS2),
                (version(Y,VY',TSY',MDY'),VS3),true) ; LOG3 ;
                    record(TID3,T3,T3',((version(X,VX',TSX',MDX'),
                        version(Y,VY',TSY',MDY'),VS4),WS,true) ; LOG4)
  = true if VX =/= VX' .
 eq notCausal(LOG) = false [owise] .
```

6.2.4. Model Checking Results

We have performed our analysis with 4 different initial states, with up to 8 transactions, 2 data items and 4 partitions[7], without finding a violation of RA or PLU. However, our Maude analysis from the same initial states found a violation of CC (which is not guaranteed by ROLA) from those same initial states. 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.

7. Statistical Model Checking of ROLA, Jessy and Walter

The weakest consistency models in [11, 6] guaranteeing RA and PLU are PSI and NMSI, and the main systems providing PSI and NMSI are, respectively, Walter [34] and Jessy [6]. To be an attractive design option, ROLA should outperform both Walter and Jessy. To quickly check whether ROLA indeed does so, we have also modeled Walter and Jessy—without their data replication features—in Maude (see `https://sites.google.com/site/siliunobi/fac-rola`), and have used statistical model checking with PVeStA to compare the performance of ROLA, Walter, and Jessy in terms of throughput, average transaction latency, and transaction commit rate.

[7] Two of those "partitions" will not store any data item, but those sites will serve/execute transactions.

Section 7.1 explains how we can extract the difference performance measures from the log introduced in Section 6. Section 7.2 explains how we generate workloads (transactions; which site to serve a transaction; and so on) probabilistically, and Section 7.3 summarizes the results of estimating the performance of ROLA, Walter, and Jessy using PVeStA.

7.1. 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 Section 6 we can define a number of functions on (states with) such a monitor that extract different performance metrics from this "system execution log".

Throughput. 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) .

op committedNumber : Record -> Float .
op $committedNumber : Record Float -> Float .
eq committedNumber(LOG) = $committedNumber(LOG,0.0) .
eq $committedNumber((record(TID,T1,T2,READS,WRITES,true) ; LOG),NUMBER)
 = $committedNumber(LOG,NUMBER + 1.0) .
eq $committedNumber((record(TID,T1,T2,READS,WRITES,false) ; LOG),NUMBER)
 = $committedNumber(LOG,NUMBER) .
eq $committedNumber(noRecord,NUMBER) = NUMBER .
```

Average Latency. 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` uses the auxiliary function `$totalLatency` to compute the sum of all transaction latencies (time between the issue time and the finish time of a committed transaction).

```
op totalLatency : Record -> Float .
op $totalLatency : Record Float -> Float .
eq totalLatency(LOG) = $totalLatency(LOG,0.0) .
eq $totalLatency((record(TID,T1,T2,READS,WRITES,true) ; LOG),LATENCY)
 = $totalLatency(LOG,LATENCY + T2 - T1) .
eq $totalLatency((record(TID,T1,T2,READS,WRITES,false) ; LOG),LATENCY) =
 = $totalLatency(LOG,LATENCY) .
eq $totalLatency(noRecord,LATENCY) = LATENCY .
```

Commit Rate. The function `cmtRate` computes the transaction commit rate by dividing the number of committed transactions by the total number of transactions:

```
op cmtRate : Config -> Float .
eq cmtRate(< M : Monitor | log: LOG > C) = committedNumber(LOG) / totalNumber(LOG) .
```

The function `totalNumber` returns the total number of (either committed or aborted) transactions; i.e., the number of records in the `LOG`:

```
op totalNumber : Record -> Float .
op $totalNumber : Record Float -> Float .
eq totalNumber(LOG) = $totalNumber(LOG,0.0) .
eq $totalNumber((record(TID,T1,T2,READS,WRITES,FLAG) ; LOG),NUMBER)
 = $totalNumber(LOG,NUMBER + 1.0) .
eq $totalNumber(noRecord,NUMBER) = NUMBER .
```

7.2. 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.

Each PVeStA simulation starts from `init(`*parameters*`)`, which rewrites to a *different* initial state in each simulation. The reason is that this expression involves generating certain values—such as the transactions—probabilistically.

`init` is defined by first generating the table, the scheduler, and the monitor:

```
op init : NzNat NzNat NzNat NzNat NzNat
          NzNat NzNat NzNat KeyAccessDistr -> Config .
op $init : NzNat NzNat NzNat NzNat NzNat KeyVars KeyVars
           NzNat NzNat NzNat KeyAccessDistr Config -> Config .
op $$init : NzNat NzNat NzNat NzNat NzNat KeyVars KeyVars
            NzNat NzNat NzNat KeyAccessDistr Nat Config -> Config .

eq init(RTX,WTX,RWTX,PS,KS,ROPS,WOPS,RWOPS,KAD) =
 = { 0.0 | nil }   < 0 . 2 : Monitor | log: noRecord >
   $init(RTX,WTX,RWTX,PS,PS,kvars(KS,keyVars),kvars(KS,keyVars),ROPS,WOPS,RWOPS,KAD,
         < 0 . 1 : Table | table: [emptyTable] >) .
```

where `$init` and `$$init` are two auxiliary functions which continue to generate and update other objects. `kvars` cuts out the first `KS` number of key-local var pairs, $< k_1, k_1 l >$; $< k_2, k_2 l >$; ... ; $< k_{ks}, k_{ks} l >$, from all constant key-local pairs.

Then `$init` uniformly assigns each key to a partition; `assignKey` also updates the table with the assigned key and its partition:

```
eq $init(RTX,WTX,RWTX,0,REPLS2,(< K,VAR > ; KVARS),KVARS',ROPS,WOPS,RWOPS,KAD,C)
 = $init(RTX,WTX,RWTX,0,REPLS2,KVARS,KVARS',ROPS,WOPS,RWOPS,KAD,
         assignKey(K,sampleUniWithInt(REPLS2) + 1,C)) .

op assignKey : Key Address Config -> Config .
eq assignKey(K,PID,< PID : Partition | datastore: VS, AS >
               < TB : Table | table: [KEYREPLICAS] > C)
 = < PID : Partition | datastore: (VS version(K,[0],eptTS,empty)), AS >
   < TB : Table | table: [replicatingSites(K,PID) ;; KEYREPLICAS] > C .
```

where `eptTS` is the default timestamp.

The function `$init` then generates transactions when all keys have been assigned (denoted by `noKeyVar`). The following case shows when there are non-zero[8] read, write, and read-write transactions left to generate:

```
eq $init(s RTX,s WTX,s RWTX,0,PS',noKeyVar,KVARS', ROPS,WOPS,RWOPS,KAD,C)
 = $$init(s RTX,s WTX,s RWTX,0,PS',noKeyVar,KVARS',ROPS,WOPS,RWOPS,
          KAD,sampleUniWithInt(s RTX + s WTX + s RWTX),C) .
```

[8] '`s`' denotes the successor function on natural numbers, and there are `s RTX`, `s WTX`, and `s RWTX` transactions of the three different types left to generate.

```
eq $$init(s RTX, s WTX, s RWTX, 0, PS', noKeyVar, KVARS', ROPS, WOPS, RWOPS, KAD, R-OR-W-OR-RW, C)
 = if R-OR-W-OR-RW < s RTX  --- generate read-only txns
   then $init(RTX, s WTX, s RWTX, 0, PS', noKeyVar, KVARS', ROPS, WOPS, RWOPS, KAD,
              addRTxn(sampleUniWithInt(PS') + 1, ROPS, KVARS', KAD, C))
   else if s RTX <= R-OR-W-OR-RW and R-OR-W-OR-RW < s RTX + s WTX
        --- generate write-only txns
        then $init(s RTX, WTX, s RWTX, 0, PS', noKeyVar, KVARS', ROPS,
                   WOPS, RWOPS, KAD, addWTxn(sampleUniWithInt(PS') + 1, WOPS, KVARS', KAD, C))
        else   --- generate read-write txns
        $init(s RTX, s WTX, RWTX, 0, PS', noKeyVar, KVARS', ROPS, WOPS, RWOPS, KAD,
              addRWTxn(sampleUniWithInt(PS') + 1, RWOPS, KVARS', KAD, C))
        fi fi .
```

This equation first probabilistically decides whether the next transaction is a read-only, a write-only, or a read-write transaction. Since the probability of picking a read transaction should be $\frac{\#\textit{readsLeft}}{\#\textit{txnLeft}}$, it uniformly picks a value `R-OR-W-OR-RW` from $[0, \ldots, \#\textit{txnLeft} - 1]$ (the number of transactions left to generate is `s RTX + s WTX + s RWTX`) using the expression `sampleUniWithInt(s RTX + s WTX + s RWTX)`. If the value picked is in $[0, \ldots, \#\textit{readsLeft} - 1]$ (`< s RTX`), we generate a new read-only transaction next (`then` branch); otherwise, in a similar way, we generate a new write-only transaction (`else-if-then` branch), or a new read-write transaction (`else-if-else` branch). But which partition should get the transaction? The partitions have identities `1`, `2`, ..., `n`, where `n` is the number of partitions (`PS'`). The expression `sampleUniWithInt(PS') + 1` gives us the partition, sampled uniformly from $[1, \ldots, n]$.

Similarly, we treat other cases based on the type(s) of the remaining transactions. We omit the details.

The following defines the functions `addRTxn` which generates a new read-only transaction:

```
op addRTxn : Address Nat KeyVars KeyAccessDistr Config -> Config .
op $addRTxn : Config -> Config .  --- generate local variables

--- if this is the first read-only txn to generate
eq addRTxn(RID, ROPS, KVARS, KAD,
          < PID : Partition | gotTxns: emptyTxnList, AS > C)
 = $addRTxn(< PID : Partition |
                gotTxns: < PID . 1 : Txn | operations: addReads(ROPS, KVARS, KAD),
                                           readSet: nil, latest: empty,
                                           localVars: empty, txnSQN: 0 >, AS >) C .

--- if there is already some txn(s) generated
eq addRTxn(RID, ROPS, KVARS, KAD,
          < PID : Partition | gotTxns: (TXNS ;; < PID . N : Txn | AS' >), AS > C)
 = $addRTxn(< PID : Partition |
                gotTxns: (TXNS ;; < PID . N : Txn | AS' > ;;
                         < PID . (N + 1) : Txn | operations: addReads(ROPS,KVARS,KAD),
                                                 readSet: nil, latest: empty,
                                                 localVars: empty, txnSQN: 0 >), AS >) C .

--- pair local variables
eq $addRTxn(< PID : Partition |
               gotTxns: (TXNS ;;
                         < PID . N : Txn | operations: OPS, readSet: nil, latest: empty,
                                           localVars: empty, txnSQN: 0 >), AS >)
 = < PID : Partition |
                gotTxns: (TXNS ;;
                         < PID . N : Txn | operations: OPS, readSet: nil, latest: empty,
                                           localVars: lvars(OPS), txnSQN: 0 >), AS > .
```

where `lvars` generates the local variables by projecting the associated local variable from each key-local variable pair. We do not show the cases of `addWTxn` and `addRWTxn`.

When there are no more transactions to generate, `$init` returns the generated objects:

```
eq $init(0,0,0,0,PS',noKeyVar,KVARS',ROPS,WOPS,RWOPS,KAD,C) = C .
```

7.3. Statistical Model Checking Results

We performed our PVeStA experiments with different configurations, with 200 transactions, 2/4 operations per read-only/-write transaction, up to 150 data items and up to 50 partitions, with lognormal message delay distributions, and with uniform and Zipfian data item access distributions. Regarding lognormal's parameters, local delays use $\mu = 0$ and $\sigma = 1$, while remote delays use $\mu = 3$ and $\sigma = 2$.

The plots in Fig. 2 show the *throughput* as a function of the percentage of read-only transactions, and number of keys (data items), sometimes with both uniform and Zipfian distributions. The plots show that ROLA outperforms Jessy, which itself outperforms Walter, for all parameter combinations. As the number of keys increases, the throughput of all three protocols increase. In particular, with 100 and more keys, ROLA with uniform distribution has significant incremental throughput. We also learn from the plots that more reads give higher throughput, since read-only transactions in all three protocols can commit locally without certification. We only plot the results under uniform key access distribution for the top plot; these results are consistent with the results using Zipfian distributions.

The plots in Fig. 3 show the *average transaction latency* as a function of the same parameters as the plots for throughput. Again, we see that ROLA outperforms Jessy and Walter in all settings. In particular, this difference between ROLA/Jessy and Walter 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 all three protocols can commit locally without certification), but ROLA still has noticeable lower latency than the other two protocols.

The plots in Fig. 4 present the *transaction commit rate* as a function of the same parameters as the above. The plots show that Walter has overall higher commit rate than ROLA, which itself has higher commit rate than Jessy, because Walter trades latency for more committed transactions. As the number of keys increases, the commit rate of all three protocols increase. In particular, with 100 or more keys, ROLA has significantly higher commit rate than Jessy. We also learn from the plots that more reads give higher commit rate, as read-only transactions in all three protocols can commit directly. We only plot the results under uniform key access distribution for some parameter combinations, which are consistent with the results using Zipfian distributions.

Our Maude specifications of ROLA, Walter, and Jessy have approximately 850, 1200 and 900 LOC, respectively, all excluding approximate 300 shared LOC related to the scheduler and sampler, and 350 shared LOC related to the initial-states generator. Our Maude specifications of ROLA, Walter and Jessy have 15, 27, and 16 rewrite rules, respectively. Computing the probabilities took a day (worst case) on 20 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.

8. 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 [25, 19, 20], different versions of RAMP [22, 21], Walter [24], P-Store [30], and Google's Megastore [14, 15]. In contrast to these papers, our paper uses formal methods to develop and validate an entirely new design, ROLA, for a new consistency model.

We are not aware of other work on formal model-based performance analysis of globally-distributed transactional databases. This might be because the most popular formal tools supporting probabilistic/statistical model checking are based on automata (e.g., Uppaal SMC [2] and Prism [1]), and it seems hard to model state-of-the-art distributed transactional systems using automata. Maude provides the expressiveness and modeling convenience that makes it possible to model such complex systems with reasonable effort.

Concerning formal methods for distributed data stores, engineers at Amazon have used TLA+ and

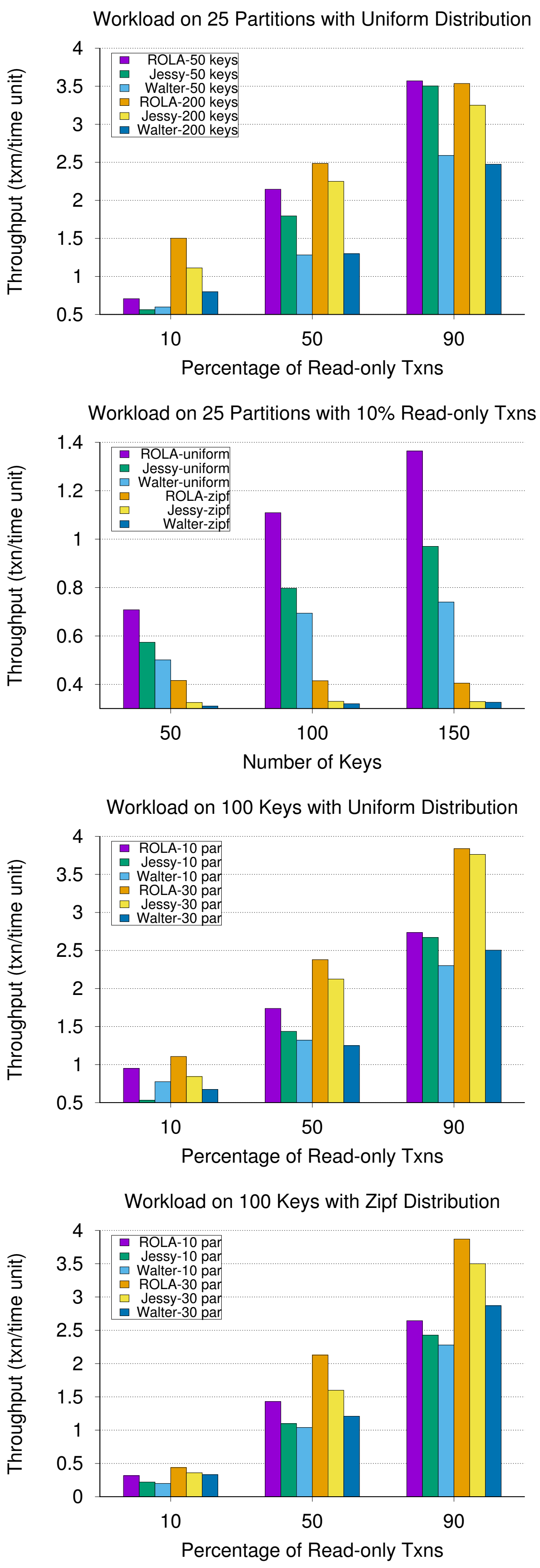

Figure 2. Throughput comparison under different workload conditions.

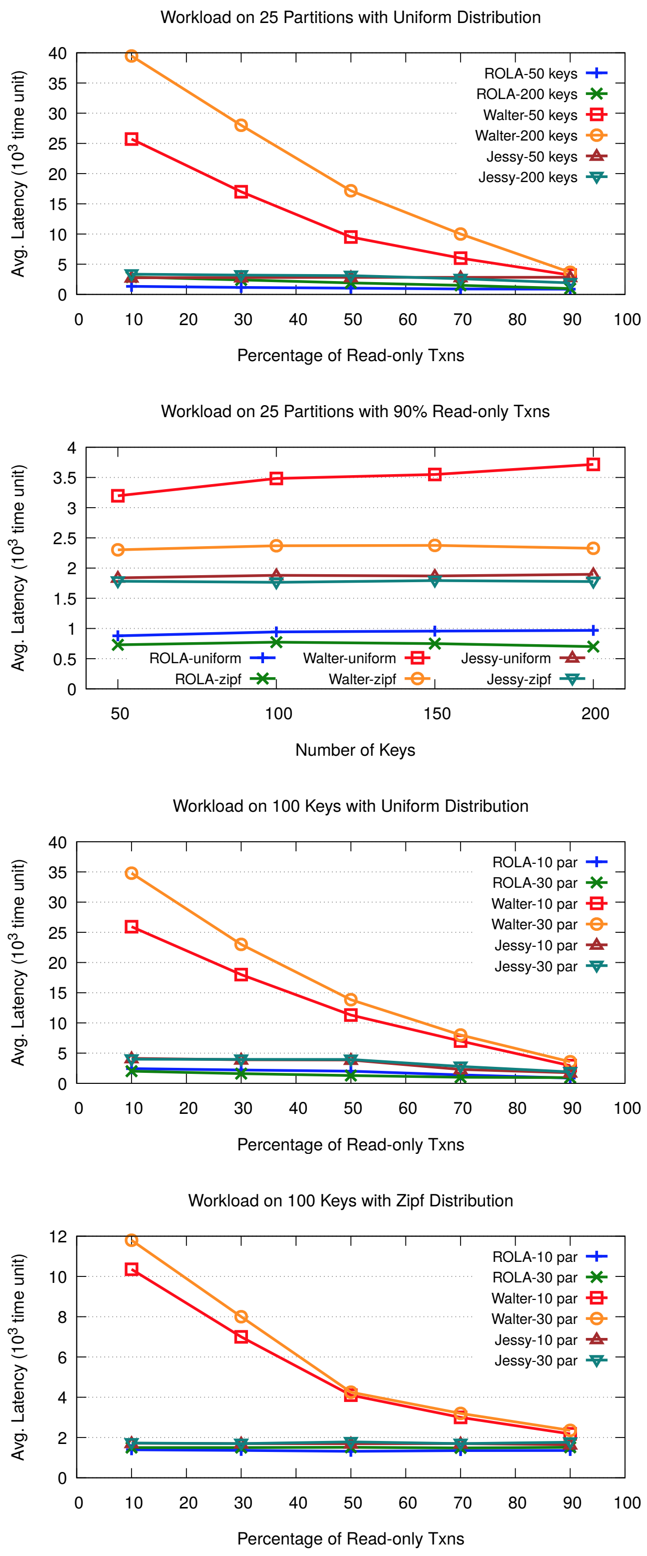

Figure 3. Average latency comparison across varying workload conditions.

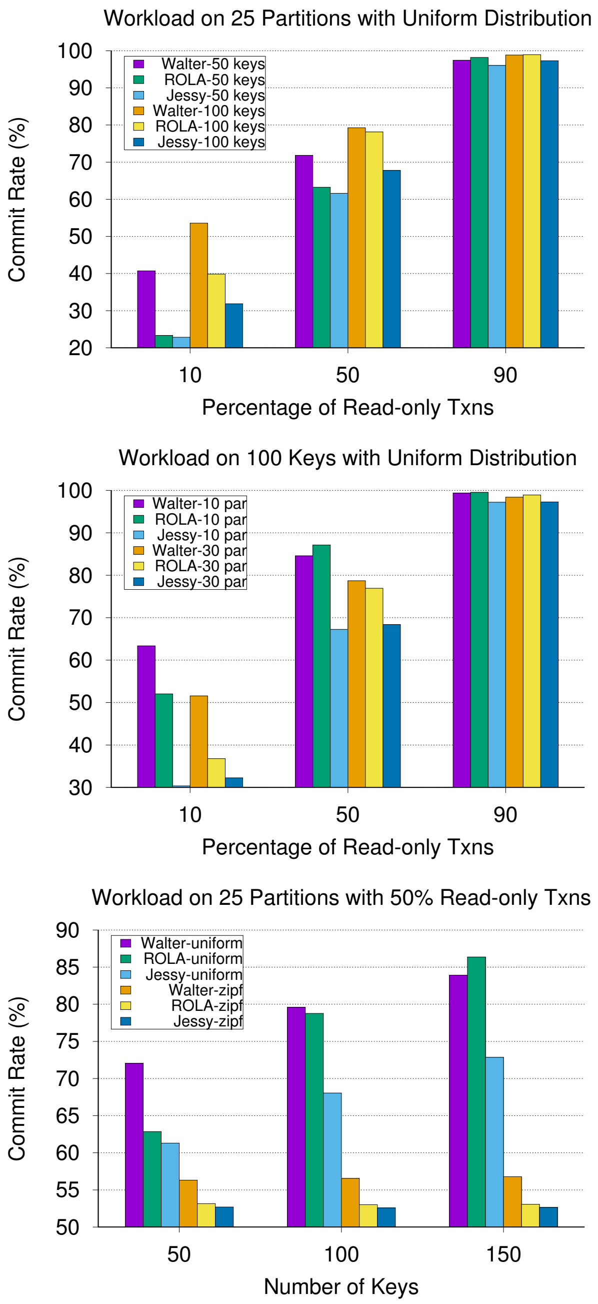

Figure 4. Transaction commit rate comparison across varying workload conditions.

its model checker TLC to model and analyze the correctness of key parts of Amazon's celebrated cloud computing infrastructure [29]. 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+ [37].

In contrast to our work, whose aim is to develop and analyze high-level formal models to quickly explore different design choices and finding bugs early, other approaches [18, 35] use distributed model checkers to analyze the *implementation* of cloud storage systems. Verifying both protocols and code is the goal of the IronFleet framework at Microsoft Research [16]. Their verification methodology includes a wide range of methods and tools, and requires (in contrast to our method) "considerable assistance from the developer."

9. 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. [11]. 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 and Jessy 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. This of course does not exclude the additional information and improvements that will be gained by *implementing* ROLA; but it substantially reduces the effort required in reaching a mature design. 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 [17], RAMP [7], and Walter [34], the performance estimates obtained by formal analysis and those obtained by experimenting with the real system were basically in agreement with each other [19, 21, 24]. This confirmed the useful predictive power of the formal analyses. Our future research will investigate the existence of a similar agreement for ROLA.

This work is part of a long-term research effort in which we have been using Maude to both meet the challenges and exploit the opportunities of modular design and analysis for cloud-based transaction systems (see [10, 28] for surveys). As part of this effort, we have formally specified in Maude and analyzed both the consistency and the performance properties of the following systems: Apache Cassandra [17], Google's Megastore [9] and its *Megastore-CGC* extension [15], RAMP [8], P-Store [31] Walter [34], and (in this paper and [23]) ROLA and its comparison with both Walter [34] and Jessy [6]. From the analysis of all these systems, which span different points in the consistency vs. performance spectrum for distributed transaction systems, a better, more modular understanding of the different algorithms that need to be combined to achieve the different designs and their relationships has been gained. An important next step in the near future is to develop a library of formally specified components and show how system designs such as those for ROLA, Walter, Jessy, the other systems described above, and other future systems can be obtained as *modular compositions* out of such a library of components. We should then use such components to extend ROLA to deal with data replication.

Acknowledgments. We thank Keshav Santhanam, 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] PRISM. `http://www.prismmodelchecker.org/`.

[2] Uppaal SMC. `http://people.cs.aau.dk/~adavid/smc/`.

[3] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, MIT, 1999.

[4] Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. *Electr. Notes Theor. Comput. Sci.*, 153(2), 2006.

[5] Musab AlTurki and José Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In *CALCO'11*, volume 6859 of *LNCS*. Springer, 2011.

[6] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, pages 163–172. IEEE Computer Society, 2013.

[7] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 41(3):15:1–15:45, 2016.

[8] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proc. SIGMOD'14*. ACM, 2014.

[9] Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR'11*. www.cidrdb.org, 2011.

[10] Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter Csaba Ölveczky, and Stephen Skeirik. Survivability: Design, formal modeling, and validation of cloud storage systems using Maude. In Roy H. Campbell, Charles A. Kamhoua, and Kevin A. Kwiat, editors, *Assured Cloud Computing*. Wiley–IEEE Computer Society Press, 2018.

[11] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[12] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[13] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Statistical model checking for composite actor systems. In *WADT'12*, volume 7841 of *LNCS*. Springer, 2013.

[14] Jon Grov and Peter Csaba Ölveczky. Formal modeling and analysis of Google's Megastore in Real-Time Maude. In *Specification, Algebra, and Software*, volume 8373 of *LNCS*. Springer, 2014.

[15] Jon Grov and Peter Csaba Ölveczky. Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In *SEFM*, volume 8702 of *LNCS*. Springer, 2014.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *Proc. 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, 2015.

[17] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, 2010.

[18] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 2014.

[19] Si Liu, Jatin Ganhotra, Muntasir Rahman, Son Nguyen, Indranil Gupta, and José Meseguer. Quantitative analysis of consistency in NoSQL key-value stores. *Leibniz Transactions on Embedded Systems*, 4(1):03:1–03:26, 2017.

[20] Si Liu, Son Nguyen, Jatin Ganhotra, Muntasir Raihan Rahman, Indranil Gupta, and José Meseguer. Quantitative analysis of consistency in NoSQL key-value stores. In *QEST*, pages 228–243, 2015.

[21] Si Liu, Peter Csaba Ölveczky, Jatin Ganhotra, Indranil Gupta, and José Meseguer. Exploring design alternatives for RAMP transactions through statistical model checking. In *Proc. ICFEM'17*, volume 10610 of *LNCS*. Springer, 2017.

[22] Si Liu, Peter Csaba Ölveczky, Muntasir Raihan Rahman, Jatin Ganhotra, Indranil Gupta, and José Meseguer. Formal modeling and analysis of RAMP transaction systems. In *SAC'16*. ACM, 2016.

[23] Si Liu, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, Indranil Gupta, and José Meseguer. ROLA: A new distributed transaction protocol and its formal analysis. In *FASE*, volume 10802 of *LNCS*, pages 77–93. Springer, 2018.

[24] Si Liu, Peter Csaba Ölveczky, Qi Wang, and José Meseguer. Formal modeling and analysis of the Walter transactional data store. In *WRLA*, volume 11152 of *LNCS*. Springer, 2018.

[25] Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. Formal modeling and analysis of Cassandra in Maude. In *ICFEM'14*, volume 8829 of *LNCS*. Springer, 2014.

[26] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[27] José Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. WADT'97*, volume 1376 of *LNCS*. Springer, 1998.

[28] José Meseguer. Formal design of cloud computing systems in maude. Technical report, University of Illinois at Urbana-Champaign, 2018.

[29] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Mark Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[30] Peter Csaba Ölveczky. Formalizing and validating the P-Store replicated data store in Maude. In *Proc. WADT'16*, volume 10644 of *Lecture Notes in Computer Science*. Springer, 2017.

[31] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine partial replication in wide area networks. In *Proc. SRDS'10*. IEEE Computer Society, 2010.

[32] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In *CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[33] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. VESTA: A statistical model-checker and analyzer for probabilistic systems. In *QEST'05*. IEEE Computer Society, 2005.

[34] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP 2011*. ACM, 2011.

[35] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proc. 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, pages 213–228. USENIX Association, 2009.

[36] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, 2006.

[37] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. Symposium on Operating Systems Principles, (SOSP'15)*. ACM, 2015.

ALGORITHM 1: RAMP-Fast

```
Server-side Data Structures
$versions$: set of versions $\langle item, value$, timestamp $ts_v$, metadata $md\rangle$
$lastCommit[i]$: last committed timestamp for item $i$

Server-side Methods
procedure PREPARE($v$ : version)
    $versions$.add($v$)
    return

procedure COMMIT($ts_c$ : timestamp)
    $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
    $\forall i \in I_{ts}, lastCommit[i] \leftarrow \max(lastCommit[i], ts_c)$

procedure GET($i$ : item, $ts_{req}$ : timestamp)
    if $ts_{req} = \emptyset$ then
        return $v \in versions : v.item = i \wedge v.ts_v = lastCommit[item]$
    else
        return $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$

Client-side Methods
procedure PUT_ALL($W$ : set of $\langle item, value\rangle$)
    $ts_{tx} \leftarrow$ generate new timestamp
    $I_{tx} \leftarrow$ set of items in $W$
    parallel-for $\langle i, v\rangle \in W$
        $w \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\})\rangle$
        invoke PREPARE($w$) on respective server (i.e., partition)
    parallel-for server $s$ : $s$ contains an item in $W$
        invoke COMMIT($ts_{tx}$) on $s$

procedure GET_ALL($I$ : set of items)
    $ret \leftarrow \{\}$
    parallel-for $i \in I$
        $ret[i] \leftarrow$ GET($i, \emptyset$)
    $v_{latest} \leftarrow \{\}$ (default value: $-1$)
    for response $r \in ret$ do
        for $i_{tx} \in r.md$ do
            $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
    parallel-for item $i \in I$
        if $v_{latest}[i] > ret[i].ts_v$ then
            $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
    return $ret$
```

Figure 5. The RAMP-Fast algorithm as described in [7].

A. The RAMP-Fast Algorithm as Given in [7]

Figure 5 shows the RAMP-Fast algorithm as it is described in [7].