Exploring Design Alternatives for Replicated RAMP Transactions Using Maude

Lei Liang
East China Normal University

Si Liu*
ETH Zurich

***Abstract*—Developing correct, scalable, and fault-tolerant distributed databases is hard and labor-intensive. The increasing complexity of such systems under modern cloud infrastructures, e.g., geo-replicated multi-partitioned datacenters, further limits the number of design alternatives that can be explored in practice. There is therefore a need for the formal analysis of both their qualitative properties, e.g., data consistency, and their quantitative properties, e.g., latency, at an early design stage.**

In this paper we use formal modeling and both standard and statistical model checking techniques to explore the design space of replicated RAMP (Read Atomic Multi-Partition) transactions for geo-replicated databases. Specifically, we develop in Maude formal, executable specifications of three replicated RAMP designs, two by the RAMP developers and one by us, and analyze their data consistency properties. We further transform the Maude models into probabilistic rewrite theories for statistical model checking w.r.t. performance properties. Our results: (i) are consistent with the conjectures made by the RAMP developers; (ii) uncover our promising new design that not only provides all crucial data consistency guarantees but also outperforms the other design alternatives.

I. Introduction

Designing reliable and high-performance distributed systems is significantly hard and labor-intensive. Modern cloud databases that typically run in a *geo-replicated* environment for availability, scalability, and fault tolerance, have further increased this complexity. For instance, integrating a *replication* mechanism with strong consistency like Paxos into a distributed database may impose a high latency cost, while more efficient mechanisms with weaker consistency may be unable to offer strong data consistency guarantees. This provides a challenge that designers must *both assure and balance* data consistency and performance to arrive at mature designs.

To evaluate and improve a system design the only available option is to touch a large code base. This often requires intensive manual efforts, has a high risk of introducing new bugs, and is not repeatable. In practice, very few designs can be explored in this way. For example, in the case of designing replicated RAMP (Read Atomic Multi-Partition) transactions, all designs were only sketched out but not implemented [4].

Formal methods have been advocated to develop and analyze high-level designs of distributed systems in general and cloud databases in particular. One representative formal ecosystem that has shown promise in meeting the above challenges is Maude [9], a rewriting-logic-based formal framework for executable specification and a high-performance analysis toolkit for object-based distributed systems [6], [15], [16], [24], [29]. In particular, Maude's LTL model checker and the PVeStA statistical model checker [3] have been used to analyze, often for the first time, both *correctness* and *performance* of high-level designs of a wide range of state-of-the-art industrial and academic cloud databases such as Apache Cassandra, Google's Megastore, RAMP, Walter, and ROLA. Model-based performance predictions using PVeStA have shown good correspondence with implementations-based evaluations under realistic deployments for Cassandra, RAMP, Walter, and ROLA [13], [14], [19], [22].

Our Goal. We anticipate that all correct and high-performance distributed databases must undergo both qualitative and quantitative analyses to obtain desirable data consistency and performance guarantees. Moreover, these analyses should be performed *at an early design stage* to avoid design errors that may be orders of magnitude more costly to eliminate at the implementation stage. In the spirit of achieving this goal, we present the formal designs and analyses of *replicated* RAMP transactions.

Status Quo of RAMP. The RAMP transaction system, proposed by Bailis *et al.* [4], is one of the most promising databases that bridge the gap between efficiency (low latency) and strong consistency (correctness) in a distributed setting (the CAP theorem [7]). RAMP offers a correctness property called "Read Atomicity" (RA) which ensures that a given transaction's updates are either all visible or not visible at all, to other transactions.

The RAMP paper [4] briefly mentions two designs for data replication, but *without* any details or formal analysis. Given the importance of data replication in modern cloud databases and of making RAMP a production-capable system in geo-replicated environment, there is therefore a clear need for formally specifying and analyzing replicated RAMP transactions.

Our Contributions. A main contribution in this paper is to develop in Maude formal, executable specifications of three replicated RAMP designs (Section IV), two by the RAMP developers [4] and one by us (Section III). We then use Maude's LTL model checker and the CAT tool [20] to analyze these specifications against data consistency properties such as the crucial RA that RAMP was built on and the prevalent session guarantee, called *read your writes* [28]. Our model checking results show that our own design satisfies all expected consistency properties (Section V).

*Corresponding author

A second key contribution is the uncovering of our promising design with highly desirable performance compared to those proposed by the RAMP developers. To do so, we further transform the above untimed, non-probabilistic, and nondeterministic Maude models that are used for standard model checking w.r.t. logical properties into timed and purely probabilistic models for statistical model checking (SMC) [27], [31] w.r.t. performance properties. We subject the three designs to realistic transaction workloads with a wider range of experimental parameters. Our SMC analysis results shows that our own design incurs less transaction latency than the other designs under *all* experimented workloads (Section VI).

II. Preliminaries

A. RAMP Transactions

To improve scalability and availability, large-scale cloud databases divide data into partitions that can be managed and accessed separately. Nonetheless, providing strong consistency guarantees for multi-partition transactions typically incurs high latency. Thus, trade-offs that combine efficiency with weaker transactional guarantees for such transactions are needed.

Bailis *et al.* [4] propose a database isolation model, called read atomicity (RA), and Read Atomic Multi-Partition (RAMP) transactions, providing efficient multi-partition operations guaranteeing RA. For instance, in a social networking scenario, if Batman and Robin become "friends" in one transaction, then other transactions should not see that Robin is a friend of Batman but that Batman is not a friend of Robin: Either both relationships are visible to others or neither is.

Three RAMP algorithms, RAMP-Fast, RAMP-Small, and RAMP-Hybrid, are proposed in [4], offering different trade-offs between the size of a message payload and performance. We focus on RAMP-Fast as both design strategies for replicating RAMP transactions (Section II-B), as well as our own design alternative (Section III), are based on it.

RAMP Writes: To guarantee that all partitions perform a transaction successfully or that none do, RAMP performs two-phase writes by using the two-phase commit protocol (2PC): In the *prepare* phase, each timestamped write is sent to its partition, which adds the write to its local database. In the *commit* phase, each partition updates an index which contains the highest-timestamped committed version of each item.

RAMP Reads: Read transactions proceed by first fetching the highest-timestamped committed *version* of each requested data item (or key) from the corresponding partition (by the first-round get message), and then deciding if they have missed any version that has been prepared but not yet committed. The timestamp and the metadata from each version read produces a mapping from items to timestamps that represent the highest-timestamped write for each transaction, appearing in the first-round read set. If the reader has a lower timestamp version than indicated in the mapping for that item, a second-round read (or get message) will be issued to fetch the missing version. Once all the missing versions have been fetched, the client can return the resulting set of versions, which include both the first-round reads as well as any missing versions fetched in the second-round reads.

B. Replicating RAMP Transactions

Replication is critical for improving data availability and system fault tolerance. However, the RAMP algorithms [4] do not take data replication into account. The paper [4] briefly discusses two design strategies for replicating RAMP transactions in a multi-datacenter/cluster setting, where database partitions are located in separate clusters in possibly geo-graphically remote regions. Nonetheless, only high-level designs are provided without any concrete algorithm or formal analysis.

Prepare-F HA RAMP (PHR): A client sets a minimum durability for its writes, i.e., the number of failures F it wishes to survive. When writing, the client issues a *prepare* request to all clusters and waits until it receives a successful response from $F+1$ partitions before starting to commit the transaction. Once all partitions have received the *prepare* request via, e.g., an asynchronous callback on the client, they commit the writes autonomously.

Sticky HA RAMP (SHR): Each client is assigned to its own cluster. Instead of issuing *prepare* and *commit* requests to all database replicas, the client issues them to its assigned cluster before forwarding them to other clusters for replication. The client can commit the transaction once it has performed the RAMP protocol only in its local cluster.

C. Rewriting Logic, Maude, and PVeStA

Rewriting Logic and Maude. Maude [9] is a rewriting-logic-based executable formal specification language and high-performance analysis tool for object-based distributed systems.

A Maude module specifies a *rewrite theory* [23] $(\Sigma, E \cup A, R)$, where:

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

Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` ("otherwise") attribute can be applied to a subterm $f(\ldots)$ only if no other equation with lefthand side $f(u_1, \ldots, u_n)$ can be applied. Maude also provides standard parameterized data types (sets, maps, etc.) that can be instantiated (and renamed). For example, `pr SET{Nat} * (sort Set{Nat} to Nats) .` defines a *set* of natural numbers of sort `Nats`.

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 $< O : C \mid att_1 : val_1, \ldots, att_n : val_n >$, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. A system state is modeled as a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages.

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule (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(O, w)` is read and consumed by an object `O` of class `C`, whose attribute `a1` 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 (e.g., `a3`) need not be mentioned in a rule.

Statistical Model Checking and PVeStA. Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [2] with rules of the form

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

where the term t' has additional new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. For a given matching instance of the variables $\overrightarrow{x}$ there can be many ways to instantiate the extra variables $\overrightarrow{y}$. The values of these variables $\overrightarrow{y}$ are sampled according to the probability distribution $\pi(\overrightarrow{x})$, which depends on the matching instance of $\overrightarrow{x}$.

Statistical model checking [27], [31] is a formal approach to analyzing probabilistic systems against temporal logic properties. It can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. In this paper we use the PVESTA tool [3] to statistically model check *purely probabilistic* systems against properties expressed by QUATEX formulas [2]. 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 a real number.

III. THE SHR-FASTER DESIGN

We have developed an alternative design for replicating RAMP transactions, called SHR-Faster. It shares the same multi-cluster setting with SHR, i.e., the "stickiness" between clients and partitions within a cluster, as well as the methods for processing writes.

The novelty lies in how SHR-Faster processes reads: a read transaction is committed in exactly one round-trip time (RTT) instead of two RTTs required by RAMP-Fast and therefore SHR, with racing/concurrent writes. The key idea is to compute an RA-consistent snapshot of the database in terms of timestamped versions prior to the first-round reads. Specifically, the client maintains, for each key K, the last seen timestamp T_{ls} and associated timestamp T_{md} for the metadata (i.e., sibling keys). Requesting the version with the maximum of T_{ls} and all those T_{md}s where K is a sibling key always guarantees RA. The client can run a periodic stabilization protocol to update T_{ls} and T_{md} for each key, or piggyback all information via regular RAMP messages.

Since each read in SHR-Faster needs only one RTT, it should incur lower transaction latency than the other replication designs. Regarding data consistency, SHR-Faster is expected to guarantee the essential *read atomicity*, as well as an important session guarantee, called *read your writes* (RYW) [28], advocated by many industrial database applications [8]. With all these interesting properties, SHR-Faster would be an attractive option for replicating read atomic multi-partition transactions where RA, RYW, and low latency are highly desired.

IV. FORMALIZING DESIGNS

Designs are formalized in an *object-oriented* style, where the state is a multiset consisting of a number of *partition* objects, each modeling a partition of the entire database, a number of *client* objects, each executing *transactions* formalized also as objects residing inside the client, and a number of *messages* traveling between the objects. Regarding system dynamics, we focus on the designs' differences. The entire executable Maude specifications are available at [1].

A. Data Types, Objects, and Messages

Basic Data Types. A *version* is modeled as a 4-tuple <*key*`,` *value*`,` *timestamp*`,` *metadata*> (of sort `Version`) consisting of the *key*, its *value*, and the version's *timestamp* and *metadata*. A timestamp is modeled as a pair <*id*`,` *sqn*> consisting of an object identifier and a local sequence number that together uniquely identify a version. Metadata are modeled as a set of keys, indicating, for each key, the sibling keys written in the same transaction. For example, if a transaction writes keys k_1, k_2, and k_3, then the sibling keys of k_1, as the metadata, are the set $\{k_2, k_3\}$. The partition-side data structure *versions* is modeled as a set of versions denoted by `Set{Version}` that instantiates Maude's built-in parameterized container *sets* on the sort `Version`. We also use another container *maps* to define *latest* and *latestCommit* as `Map{Key,Timestamp}` mapping *keys* to *timestamps*.

Classes and Objects. A partition stores parts of the entire database. We formalize it as an object instance of the following class `Partition`:

```
class Partition | versions : Set{Version},
                  latestCommit : Map{Key,Timestamp} .
```

where the partitioned database consists of a set of versions for each key stored locally. The partition also retains the timestamp of the latest committed version for each key.

A transaction is modeled as an object instance of class `Txn`:

```
class Txn |
```

```
  operations : List{Operation}, txnSqn : Nat,
  votes : Set{Oid}, commits : Set{Oid},
  1stReads : Set{Oid}, 2ndReads : Set{Oid},
  latest : Map{Key,Timestamp},
  readset : Set{kvPair}, writeset : Set{kvPair} .
```

where the attribute `operations` denotes the transaction's operations, with each of the form `read(`*key*`)` or `write(`*key*`,`*value*`)`. The attribute `readset`, resp. `writeset`, denotes the <*key*,*value*> pairs fetched, resp. written, by the read, resp. write, operations of the transaction. The attribute `votes` stores the votes from the partitions which participate in the 2PC protocol. The remaining attributes `commits`, `1stReads`, and `2ndReads` store, respectively, the partitions from which the client is awaiting committed acks, first-round get replies, and second-round get replies.

A client, modeled as an object instance of class `Client`, is delegated to process transactions:

```
class Client |
    sqn : Nat, queue : List{Object},
    executing : Set{Object}, committed : List{Object}
    dc : Oid,  last : Map{Key,Pair} .
```

The attributes `queue`, `executing`, and `committed` store the transaction object(s) which are waiting to be executed, currently executing, and committed, respectively. Concurrently executing transactions can be modeled by multiple clients with each holding a currently executing transaction. The attribute `dc` refers to the datacenter where the client is co-located. The attribute `last`, specific to SHR-Faster, maps each key to the pair <*timestamp*,*metadata*> of its latest version that the client has seen.

The state also contains a "table" object of class `Table` mapping each key to the partitions storing the key (called the key's replicas), and each datacenter to its partitions.

The following shows an example initial state of sort `Configuration` (with some parts replaced by '...') with two clients, `c1` and `c2`, queuing, respectively, two and one transactions. Read transaction `t1` has two read operations on, respectively, two keys `k1` and `k2`. Write transaction `t2` consists of two write operations, with each writing a value (e.g., `"TASE"`) to a key (e.g., `k2`). Transaction `t3` is a transaction of a single read on key `k4`. The key space is split by two partitions `p1` (resp. `p3`) and `p2` (resp. `p4`) and replicated in two datacenters, e.g., key `k1` is replicated at partitions `p1` and `p4` that are located in datacenters `d1` and `d2`, respectively. Each partition is initialized accordingly; in particular, for each key, the value is the empty string, the timestamp is `null`, and the metadata is an `empty` set:

```
op init : -> Configuration .
eq init =
  < c1 : Client | queue :
      (< t1 : Txn | operations : (read(k1) read(k2)),
                    txnSqn : 0, latest : empty,
                    votes : empty,  commits : empty,
                    1stReads : empty, 2ndReads : empty,
                    readset : empty, writeset : empty >
       < t2 : Txn | operations : (write(k2,"TASE")
                                  write(k3,"ECNU")), ... >),
    sqn : 0, executing : empty, committed : nil,
    dc : d1, last : empty >
  < c2 : Client | queue :
       < t3 : Txn | operations : read(k4), ... >, ... >
  < p1 : Partition | versions : (< k1, "", null, empty >,
                                 < k4, "", null, empty >),
                     latest : empty >
  < p2 : Partition | ... > < p3 : Partition | ... >
  < p4 : Partition | ... >
  < t : Table |
    replicas: (k1 |-> (p1, p4), k2 |-> (p2, p3),
               k3 |-> (p2, p4), k4 |-> (p1, p3)),
    clusters: (d1 |-> (p1, p2), d2 |-> (p3, p4)) >
```

Messages. A message has the form `(to` *receiver* `from` *sender* `:` *mc*`)`. The terms *sender* and *receiver* are object identifiers. The term *mc* is the message content, having the form of either `get(`*txn*`,`*key*`,`*ts*`,`*dc*`)`, a get message, sent to the corresponding partition located in datacenter *dc*, requesting *key*'s version with the timestamp *ts*; `reply(`*txn*`,`*key*`,`*val*`,`*ts*`,`*md*`)`, the returned version from the partition for first-round get request; `reply2(`*txn*`,`*key*`,`*val*`,`*ts*`,`*md*`)`, the respond to second-round get request; `prepare(`*txn*`,`*key*`,`*val*`,`*ts*`,`*md*`,`*dc*`)`, the version sent from a write transaction to the corresponding partition in datacenter *dc*; `vote(`*txn*`,`*vote*`)`, the reply to the corresponding prepare message where *vote* denotes the partition's decision on whether it can commit the transaction; `commit(`*txn*`,`*ts*`,`*dc*`)`, the demand for committing the version with timestamp *ts*; or `ack(`*txn*`)`, the acknowledgment of committing the transaction from the partition. Note that SHR-Faster only use `reply` messages for transaction reads as no second-round read is needed.

B. Formalizing System Dynamics

We have formalized the dynamic behaviors of all three designs using rewrite rules. In this section we focus on the core difference (denoted by the colored parts in italics in a rule), that typically lies in the client-side methods, between PHR and SHR w.r.t. processing write transactions, and between SHR and SHR-Faster w.r.t. processing reads.[1] We exemplify each case with one rule, and refer to [1] for the remaining rules including the partition-side methods.

1) Processing Writes: The following rule shows how an SHR client proceeds once the *prepare* phase ends (checked by the `all` predicate if all `vote` messages have been received):

```
crl [rcv-vote-shr] :
  (from O' to O : vote(TID,yes))
  < T : Table | replicas : RS, cluster : DS >
  < O : Client | dc : DC, executing :
      < TID : Txn | operations : OPS, txnSqn : N,
                    votes : VS, commits : CS > >
=>
  < T : Table | >
  < O : Client | executing :
      < TID : Txn | votes : VS', commits : CS,
          (if all(VS') then
              addCommitsOneDc(OPS,RS,DS,DC)
                      else empty fi) > >
  (if all(VS') then
     genCommitsOneDc(O,TID,N,OPS,RS,DS,DC)
                 else none fi)
  if VS' := remove(O',VS) .
```

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

The client starts to commit the executing transaction by generating the commit messages for all the participants involved in the 2PC protocol in its datacenter (by genCommitsOneDc). The addCommitsOneDc function computes the set of those participants' identifiers that are added to commits. The remove function updates the client's waiting list for votes.

Modifications for PHR. Instead of all votes in the co-located cluster, a PHR client collects a *majority* of responses (checked by the quorum predicate) from all replicas, including remote ones, before propagating the commit messages (by the genCommitsAllDc function):

```
crl [rcv-vote-phr] :
  (from O' to O : vote(TID,yes))
  < T : Table | replicas : RS, cluster : DS >
  < O : Client | executing :
      < TID : Txn | operations : OPS, txnSqn : N,
                    votes : VS, commits : CS >
                  committed : TXNS, dc : DC >
=>
  < T : Table | >
  (if quorum(VS')
     then < O : Client | executing : empty,
                         committed : (TXNS
       < TID : Txn | votes : VS', commits : (CS,
            addCommitsAllDc(OPS,RS,DS,DC)) >) >
          genCommitsAllDc(O,TID,N,OPS,RS,DS,DC)
     else < O : Client | executing :
       < TID : Txn | votes : VS' > > fi)
  if VS' := remove(O',VS) .
```

The corresponding replicas are added to commits, from which the client awaits the ack messages. Notice that, unlike SHR, PHR then commits the transaction by moving it from executing to committed; the client can therefore issue the next transaction in the queue.

2) Processing Reads: The key difference between SHR and its optimization SHR-Faster lies in how a client reacts to the first-round reads. In SHR, upon receiving a reply, the client checks whether all first-round responses have been collected (by the all predicate): the client then issues second-round get messages (by gen2ndReads) if any writes are missing; otherwise, it continues to wait for the remaining first-round reply messages:

```
crl [rcv-1st-reply-shr] :
  (from O' to O : reply(TID,K,V,TS,MD))
  < T : Table | replicas : RS, cluster : DS >
  < O : Client | dc : DC, executing :
      < TID : Txn | operations : OPS, readset : RS,
                    latest : VL, 1stReads : 1ST,
                    2ndReads : 2ND > >
=>
  < T : Table | >
  < O : Client  | executing :
      < TID : Txn | readSet : RS', latest : VL',
                    1stReads: 1ST', 2ndReads : 2ND,
            (if all(1ST') then
                2ndPars(VL',RS',RS,DS,DC)
                          else empty fi) > >
  (if all(1ST') then
      gen2ndReads(O,TID,VL',RS',RS,DS,DC)
                else none fi)
  if RS' := RS, < K,V,TS,MD > /\ VL' := lat(VL,MD,TS)
     /\ 1ST' := remove(O',1ST) .
```

The 2ndPars function computes the corresponding partitions (possibly none) involved in the second-round communication. The lat function updates the current key-timestamp mappings VL with the latest committed timestamp received (i.e., TS), which are then used to compute the missing writes. The client also updates its readset accordingly (i.e., RS').

The SHR client commits a read transaction (moved from executing to committed) if both 1stReads and 2ndReads are empty, indicating that all first- and second-round replies have been received:

```
rl [rtxn-committed-shr] :
  < O : Client | committed : TXNS, executing :
      < TID : Txn | 1stReads : empty,
                    2ndReads : empty > >
=>
  < O : Client | committed : (TXNS < TID : Txn | >),
                 executing : empty > .
```

Modifications for SHR-Faster. A read transaction in SHR-Faster is committed in one RTT if all first-round reply messages have been collected:

```
rl [rcv-1st-reply-shr-faster] :
  (from O' to O : reply(TID,K,V,TS,MD))
  < O : Client | last : KP, committed : TXNS,
                 executing :
      < TID : Txn | readset : RS, 1stReads : 1ST > >
=>
  if all(1ST') then
   < O : Client | last : last(KP,< K,V,TS,MD >),
                  committed : (TXNS
     < TID : Txn | readset : (RS, < K,V,TS,MD >),
                   1stReads : remove(O',1ST) >),
                  executing : empty >
               else
   < O : Client | executing :
     < TID : Txn | readset : (RS, < K,V,TS,MD >),
                   1stReads : remove(O',1ST) > > fi .
```

where, unlike SHR, no second-round get request is generated. The client also updates last with the received version for computing RA-consistent snapshots of the database (see [1] for details).

V. Model Checking Correctness Properties

This section shows our formal analysis of all three designs against three data consistency properties using the CAT tool [20] and Maude's built-in model checker. In particular, we formalize in Maude the *read your writes* session guarantee which CAT cannot handle.

A. The CAT Model Checker

CAT [20] is a Maude-based tool for model checking data consistency properties of distributed transaction systems like SHR specified in Maude. It currently supports nine transactional consistency properties, including RA and the weaker consistency property *read committed* (RC) used as the default isolation level by almost all SQL databases [11].

B. Formalizing Read Your Writes

Read your writes [28] guarantees that the effects of all writes performed by a client are visible to its subsequent reads. We define RYW in Maude as a predicate over states as follows:

```
op ryw : Configuration -> Bool .

*** case 1: read even earlier write in the same session
eq ryw(< O : Client | committed : (TXNS1
      < TID1 | writeset : (< K,V,TS,MD >, WS) > TXNS2
      < TID2 | writeset : (< K,V',TS',MD' >, WS') > TXNS3
      < TID3 | readset : (< K,V,TS'',MD'' >, RS) > TXNS4) >
        CONF) = false .

*** case 2: read earlier write from a different session
ceq ryw(< O : Client | committed : (TXNS1
      < TID1 | writeset : (< K,V,TS,MD >, WS) > TXNS2) >
        < O' : Client | committed : (TXNS3
      < TID2 | writeset : (< K,V',TS',MD' >, WS') > TXNS4
      < TID3 | readset : (< K,V,TS'',MD'' >, RS) > TXNS5) >
        CONF) = false if TS < TS' .

*** case 3: read initial value
eq ryw(< O : Client | committed : (TXNS1
      < TID1 | writeset : (< K,V,TS,MD >, WS) > TXNS2
      < TID2 | readset : (< K,"",TS',MD' >, RS) > TXNS3) >
        CONF) = false .

*** remaining cases
eq ryw(CONF) = true [owise] .
```

The predicate `ryw` analyzes RYW by checking (the state) whether there was a "bad situation" in which a committed transaction had fetched an RYW-*inconsistent* value. More specifically, we consider three such cases:

1) Instead of returning the previous write in the same session, a committed transaction `TID3` reads an even earlier write (by matching the key `K` and the value `V`) in the same session (the first equation).
2) A committed transaction `TID3` fetches a write from a different session, which has an earlier version than `TID3`'s previous write in the same session (the second conditional equation).
3) Despite a previous write in the same session, a committed transaction `TID2` returns the initial value (the third equation).

The fourth equation covers all the remaining (RYW-consistent) cases. Note that `CONF` (of sort `Configuration`) denotes the rest of the entire configuration (or state) in the first three equations.

C. Model Checking Analysis

We have applied CAT and Maude's built-in model checker to three Maude models of the designs against RA, RC, and RYW, respectively.

TABLE I
MODEL CHECKING RESULTS FOR CONSISTENCY PROPERTIES. "✓", RESP. "×", REFERS TO SATISFYING, RESP. VIOLATING THE PROPERTY.

Maude Model	LOC	# rules	Consistency Property		
			RC	RA	RYW
PHR	845	14	✓	✓	×
SHR	894	14	✓	✓	✓
SHR-Faster	887	12	✓	✓	✓

Table I summarizes our model checking results. All three designs satisfy RA, as well as RC. Regarding RYW, our analysis shows that only PHR does not satisfy the property. The counterexample obtained by analyzing the initial state where two clients `c1` and `c2`, each issuing two transactions,

```
c1:  (write(k1,24) write(k2,73))
     (read(k1) read(k2))
c2:  (write(k1,11) write(k2,54))
     (write(k3,3) write(k1,32))
```

shows that the read operations from `c1` return the values written by `c2`'s first write transaction `write(k1,11) write(k2,54)` that was issued before `c1`'s first write transaction. The reason is that a PHR read transaction could access the versions written before its last write transaction commits; PHR enables "fast" commit of writes once the client collects a majority of responses from all the participants.

We have run CAT with automatically generated *different initial states* [20], with up to 6 transactions, 2 operations per transaction, 3 clients, 6 partitions, 3 datacenters, 3 keys, and 3 replicas per key. We have also employed these initial states in our analysis of RYW with Maude's built-in model checker. Each analysis took about one hour (worst case) to execute on a 3.3 GHz Dual-Core Intel Core i7 CPU with 16GB memory.

VI. Quantitative Analysis Using SMC

Our qualitative analysis in Section V investigates the desired data consistency guarantees—SHR-Faster, as well as SHR, satisfies all three properties.

SHR-Faster must therefore outperform SHR, as well as PHR, to be an attractive design. In this section we use statistical model checking with PVeStA to compare the performance of these designs in terms of average transaction latency.

A. Probabilistic Models

The models in Section IV are untimed, non-probabilistic, and nondeterministic, which can be analyzed w.r.t. their logical properties using model checking. To estimate the system performance, we need to enrich the models with time and probabilities and eliminate any nondeterminism. The transformed models are therefore *timed* and *purely probabilistic*, which can be subjected to SMC.

We follow the methodology in [2] by assigning to each message a delay sampled from a continuous probability distribution so that the probability that two messages have the same delay is 0. The framework [2] then guarantees no unquantified nondeterminism by firing a rewrite rule according to the arrival of a message. The elapse of message delays, as well as advancing the global clock, is maintained by a scheduler object in the framework.

We exemplify with a (partition-side) rule how we have transformed the untimed non-probabilistic rewrite rules to the timed and probabilistic setting. In the transformed probabilistic rule below, upon receiving the `prepare` message at global time `GT`, the partition stores the associated version `< K,V,TS,MD >` in its local database, and sends its vote `yes` back with message delay `D` sampled from probability distribution `distr(`$\overrightarrow{x}$`)`:

```
rl [rcv-prep-prob] :
  {GT, (to O from O' : prepare(TID,K,V,TS,MD,DC))}
  < O : Partition | versions : VS >
=>
```

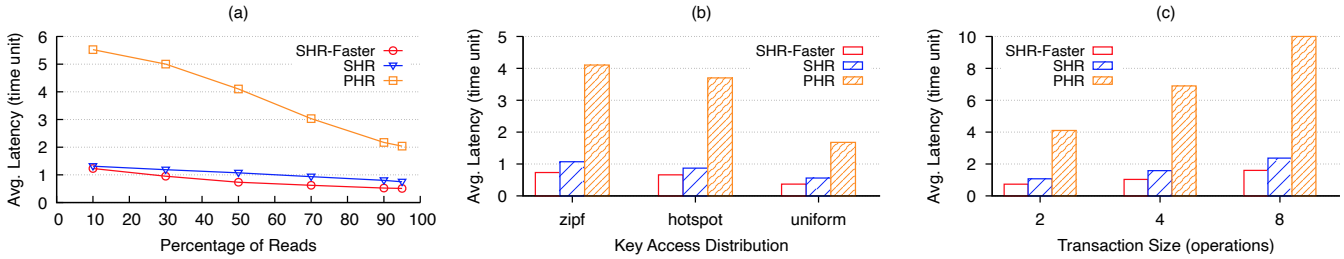

Fig. 1. Statistical model checking analysis results for the comparison of three designs.

```
< O : Partition | versions : (VS, < K,V,TS,MD >) >
[D, (to O' from O : vote(TID,yes))]
with probability D := distr(x⃗) .
```

where `distr` is parametric on $\vec{x}$, the corresponding variables in the lefthand side of the rule (see [20] for details). The outgoing `vote` must then be consumed by receiver `O'` when time `D` elapses, i.e., at global time `GT + D`.

B. Extracting Transaction Latency Measures

PVeStA estimates the expected value of a QuaTEx expression on a run, up to a desired statistical confidence. Hence, the key to performing SMC is to define a measure on system runs. We then define a function on states that extracts the *latency* metric from the system execution log recorded by a monitor object (similar to the monitor defined in CAT [20]; see Section V). Specifically, the monitor stores important information about each transaction, i.e., its ID, issue time, and commit time. Thanks to the transformed rules, we can extract timing information, e.g., the current global time, directly from an incoming message (see rule `rcv-prep-prob` above). We have modified the probabilistic models by updating the monitor object whenever needed. See [1] for details.

C. Benchmark and Experimental Setup

To statistically analyze the designs' performance, we implement in Maude a parametric workload generator that probabilistically generates a different initial state for each simulation in an SMC analysis. The parameters are: #clients (default: 50), #clusters (default: 3), #partitions/cluster (default: 5), #keys (default: 1k), #transactions (default: 400), #ops/transaction (default: 2), %reads (default: 50), and key-access distribution[2] (default: Zipfian [10]).

To mimic the real-world multi-datacenter environment, we use the lognormal distribution for probabilistically sampling message delays [5], which are further modulated linearly according to the distance between the sender and receiver. We employed 20 d430 Emulab machines [30], each with two 2.4 GHz 64-bit 8-Core E5-2630 processors, to parallelize SMC with the PVeStA tool [3]. We set the statistical confidence level to 95% and the size parameter to 0.01 for all our experiments.

[2]Key-access distribution is the probability that a read/write operation accesses a certain data item.

D. SMC Analysis Results

We have subjected the transformed purely probabilistic models to the above workload generator and PVeStA, and performed three sets of experiments on measuring average latency. Each SMC analysis took up to two hours (worst case) to terminate. Each point in the plots represents the average of three SMC results.

Fig. 1 depicts the average latency as a function of percentage of reads, key-access distribution, and transaction size, respectively. The plots show that our SHR-Faster design outperforms SHR, which in turn outperforms PHR, for *all* parameter combinations.

As shown in Fig. 1-(a), the latency for PHR tends to decrease significantly, starting from 10% reads, with increasing reads in the mixed workload. The reason is that transactional writing must wait for $F+1$ (in this case 2) responses from the corresponding partitions, including at least one in the remote cluster, while reads need not to due to the weak consistency guarantee of read atomicity. The latency difference between SHR-Faster and SHR becomes larger with more concurrent writes (i.e., 50%–70% reads) as more second-round reads were issued to fetched the missing writes in SHR.

Fig. 1-(b) plots the comparison under various key-access distributions. With more frequent key accesses, i.e., Zipfian and hotspot (80% operations on 20% keys [10]), the latency difference is quite large as more second-round reads were sent to possibly remote partitions. The same argument applies to Fig. 1-(c) where increasing transaction size (i.e., #ops/transaction) leads to more remote communications.

VII. Related Work

Maude for RAMP. In [17] we formalized RAMP and its variants and used Maude model checking to analyze their correctness properties. In [14] we modeled RAMP and its extensions in probabilistic rewrite theories and employed SMC to compare their performance. None of these RAMP-family algorithms considers data replication that is indispensable in modern distributed database systems. In contrast, this paper focuses on replicated RAMP transactions and also introduces our most promising replication design SHR-Faster. Moreover, this paper follows a more general methodology based on formal modeling with rewrite rules that are further transformed into probabilistic rewrite rules so that both qualitative and quantitative properties are investigated.

Maude for Distributed Databases. Existing work [12], [18], [19], [26] specifies and model checks various distributed transaction systems that provide other types of data consistency properties, as opposed to *read atomicity* and *read your writes* considered in this paper. The paper [12] estimate Google's Megastore using randomized Real-Time Maude simulations. However, this methodology cannot offer any measure of statistical confidence in the estimations. The papers [13], [21] show how the NoSQL key-value store Cassandra has been formally analyzed for both data consistency and performance properties using Maude and PVeStA. Cassandra only supports single read/write operation, while transactional consistency properties such as RA are more complex to be analyzed.

Formal Methods for Distributed Databases. Other formal methods have been used in the design of distributed databases. One representative is TLA+ and its model checker TLC that have been used in both industry, e.g., Amazon's cloud computing infrastructure [25], and academia, e.g., the TAPIR protocol [32]. In contrast to our work, they employ formal methods only for correctness checking but could not run any performance measurement as complained in [25].

VIII. Conclusion

We have explored in this paper three designs for replicated RAMP transactions by formal modeling with Maude rewrite rules and probabilistic rewrite theories, and by the analysis with standard model checking for data consistency properties and PVeStA statistical model checker for performance measurements, respectively. Substantial knowledge has been obtained about the design space. In particular, we have identified a promising design with sticky clients and one round-trip communication for transactional reads. We have shown that our design not only provides the crucial RA and RYW consistency guarantees, but also outperforms the other design alternatives w.r.t. transaction latency under various workloads.

A natural next step is to confirm our model-based SMC predictions of the designs with their implementation-based evaluations under the actual deployment in the cloud.

References

[1] Maude specifications of PHR, SHR, and SHR-Faster. https://github.com/siliunobi/replicated-ramp.

[2] 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.

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

[4] 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.

[5] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] 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 *Assured Cloud Computing*, chapter 2, pages 10–48. Wiley-IEEE Computer Society Press, 2018.

[7] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, page 7, 2000.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, et al. TAO: facebook's distributed data store for the social graph. In *USENIX ATC*. USENIX Association, 2013.

[9] 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.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[11] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *PODC*. ACM, 2017.

[12] 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.

[13] 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.

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

[15] Si Liu, Peter Csaba Ölveczky, and José Meseguer. Formal analysis of leader election in manets using real-time maude. In *Software, Services, and Systems*, volume 8950 of *LNCS*, pages 231–252. Springer, 2015.

[16] Si Liu, Peter Csaba Ölveczky, and José Meseguer. Modeling and analyzing mobile ad hoc networks in Real-Time Maude. *J. Log. Algebraic Methods Program.*, 85(1):34–66, 2016.

[17] 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*. ACM, 2016.

[18] 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*. Springer, 2018.

[19] Si Liu, Peter Csaba Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. *Formal Asp. Comput.*, 31(5), 2019.

[20] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. Automatic analysis of consistency properties of distributed transaction systems in Maude. In *TACAS*, volume 11428 of *LNCS*. Springer, 2019.

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

[22] Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. Generating correct-by-construction distributed implementations from formal Maude designs. In *NFM*, LNCS. Springer, 2020.

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

[24] José Meseguer. Twenty years of rewriting logic. *J. Algebraic and Logic Programming*, 81:721–781, 2012.

[25] 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.

[26] Peter Csaba Ölveczky. Formalizing and validating the p-store replicated data store in maude. In *WADT*, volume 10644 of *LNCS*. Springer, 2016.

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

[28] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*. IEEE Computer Society, 1994.

[29] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the attack surface of trigger-action IoT platforms. In *CCS*, pages 1439–1453. ACM, 2019.

[30] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, et al. An integrated experimental environment for distributed systems and networks. In *OSDI*. USENIX Association, 2002.

[31] 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.

[32] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *SOSP 2015*. ACM, 2015.