Increasing Consistency in Multi-site Data Stores: Megastore-CGC and Its Formal Analysis*

Jon Grov[1] and Peter Csaba Ölveczky[1,2]

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

Abstract. Data stores for cloud infrastructures provide limited consistency guarantees, which restricts the applicability of the cloud for many applications with strong consistency requirements, such as financial and medical information systems. Megastore is a replicated data store used in Google's cloud infrastructure. Data are partitioned into entity groups, and consistency is only guaranteed if each transaction only accesses data from a single entity group. This paper extends Megastore to also provide consistency for transactions accessing data from multiple entity groups, thereby increasing the applicability of such cloud data stores. Our extension, Megastore-CGC, achieves this extra consistency without introducing significant additional message exchanges. We used the formal specification language and analysis tool Real-Time Maude throughout the development of Megastore-CGC. We introduce Megastore-CGC, its Real-Time Maude specification, and show how Real-Time Maude can estimate the performance of Megastore-CGC and model check Megastore-CGC.

1 Introduction

Database facilities are important for applications, such as payroll systems, stock exchange systems, banking, online auctions, and medical systems, where inconsistencies (such as lost or corrupted medication requests or money deposits) cannot be tolerated. Databases therefore usually provide *transactions*. A transaction is a sequence of read and write operations which are executed equivalently to an atomic execution, and where the concurrent execution of a set of transactions is equivalent to some sequential execution of the transactions.

The availability and performance of the database is crucial in many of the applications mentioned above, which would therefore benefit from running on a cloud infrastructure. However, there is currently limited support for transactions in cloud-based data stores. A main reason is that data must be *replicated* across multiple sites to achieve the availability and scalability expected from cloud services. Multi-site replication introduces many challenges, in particular regarding *performance*, since ensuring consistency requires costly message exchanges, and *fault tolerance*, since sites may go down or messages may be lost.

* This work was partially supported by AFOSR Grant FA8750-11-2-0084.

D. Giannakopoulou and G. Salaün (Eds.): SEFM 2014, LNCS 8702, pp. 159–174, 2014.

One of the most mature cloud-based data management systems providing some transaction support is Google's Megastore [1]. Megastore is widely used both internally at Google, backing services such as GMail and Google+, and externally through Google's Platform-as-a-Service offering Google AppEngine. Megastore is a very complex system, described informally in the overview paper [1]. To facilitate research on the Megastore approach to data management in the cloud, a precise and more detailed description is needed. We therefore define in [9] a formal model of (the) Megastore (approach) using the rewriting-logic-based Real-Time Maude formal specification language [13].

Megastore works well for many less consistency-critical applications, such as email, social media, or online newspapers, but has some limitations for more consistency-critical applications: the data must be partitioned into a set of *entity groups*, and consistency is only guaranteed if each transaction only accesses data from a single entity group. This may require a difficult (or impossible) tradeoff between scalability and consistency, as illustrated in Section 3.

In this paper, we extend Megastore to provide consistency also for transactions accessing multiple entity groups. Our extension, called Megastore-CGC ("Megastore with cross-group consistency"), achieves this additional feature without reducing Megastore's performance and fault-tolerance.

Achieving fault-tolerant transaction management is very hard [18]. We therefore formally defined Megastore-CGC in Real-Time Maude, which allowed us to use Real-Time Maude simulations and LTL model checking extensively *throughout* the development of Megastore-CGC. To the best of our knowledge, this is the first time formal methods have been used *during* the design of a cloud-based transaction protocol. We experienced that anticipating all possible behaviors of Megastore-CGC is impossible. A similar observation was made by Google's Megastore team, which implemented a pseudo-random test framework, and state that *"the tests have found many surprising problems"* [1]. Compared to such a testing framework, Real-Time Maude model checking analyzes not only a set of pseudo-random behaviors, but all possible behaviors from an initial system configuration. Furthermore, we believe that Real-Time Maude provides a more effective and low-overhead approach to testing than a real testing environment.

Several studies indicate that the test-driven development method significantly improves the quality of the resulting product [12]. In this method, a suite of tests for the planned features are written before development starts. This set of tests is then used both to give the developer quick feedback during development, and as a set of regression tests when new features are added. However, test-driven development has traditionally been considered to be unfeasible when targeting fault tolerance in complex concurrent systems due to the lack of tool support for testing large number of different scenarios. Our experience from Megastore-CGC is that with Real-Time Maude, a test-driven approach is possible also in such systems, since many complex scenarios can be quickly tested by model checking.

To summarize, the contributions of this paper are the following:

1. Section 3 defines an extension of Megastore, called Megastore-CGC, that provides consistency also for transactions accessing multiple entity groups.

2. Section 4 defines a formal model of Megastore-CGC in Real-Time Maude.
3. We use Real-Time Maude Monte Carlo simulations in Section 5 to show that the performance of Megastore-CGC is on par with that of Megastore.
4. We show in Section 6 how Real-Time Maude LTL model checking can be used to analyze the correctness of Megastore-CGC, including how such model checking can analyze the important feature *serializability* property of distributed databases: any concurrent execution of a set of transactions should produce results equivalent to a serial execution of the same transactions.

2 Preliminaries

Megastore. Megastore [1] is a replicated data store developed by Google. Data are key-value pairs called *entities*. A *transaction* is a sequence of read and write operations on entities, followed by a commit request. Entities are partitioned into *entity groups*, and each entity group is replicated at different sites. A replicated transaction log is maintained for each entity group. For transactions accessing a single entity group, Megastore ensures *atomicity* and *serializability* (consistency) by only allowing one transaction to update the log at any time.

Initially, all read operations in a transaction t are executed locally at a site s, and t's updates are buffered. Each site has a *coordinator*, which is always informed about whether the local replica is up-to-date. If the local replica is not up-to-date for an entity requested by t, a majority read is performed.

Let t read and write entities from entity group eg, and let lp be the current log position in the replicated log of eg. When t requests commit, site s prepares a log entry for eg containing t's updates, and runs the following variant of the *Paxos* consensus protocol [11] to assign this entry to log position $lp + 1$:

1. Site s sends a *proposal* containing the log entry and the next leader (normally s) to the current leader site l, which was elected during the previous commit. If l accepts the entry, s sends the proposal to the other sites. If not, e.g., due to a concurrent update of the same entity group, the transaction is aborted.
2. Site s then waits for *acknowledge* responses from all sites. If some sites fail to acknowledge, s sends an *invalidate* message to these sites.
3. When each site has acknowledged either the proposal or the invalidate message, s requests all sites to apply t's updates. Each site replicating eg then appends the chosen log entry for position $lp + 1$ to the local copy of the transaction log for eg, and subsequently updates the local data store.

In the presence of failures, s may fail to achieve consensus. In this case another site may propose itself as the leader, and starts at step (1). If multiple sites propose log entries for the same log position, Paxos ensures that only one is elected, and the others are aborted.

Real-Time Maude. Real-Time Maude [13] is a formal modeling language and high-performance simulation and model checking tool for distributed real-time

systems. The modeling formalism is *expressive* and *intuitive*, allowing developers with limited formal methods experience to model complex real-time systems.

An algebraic equational specification (specifying sorts, subsorts, functions and equations defining the functions) defines the data types in a "functional programming style." Labeled rewrite rules `crl [`l`]:` t `=>` t' `if` *cond* define local transitions from state t to state t', and tick rewrite rules `crl [`l`]: {`t`} => {`t'`} in time` Δ `if` *cond* advance time in the *entire* state t by Δ time units.

A declaration `class` C `|` $att_1 : s_1, \ldots, att_n : s_n$ declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. The state is a term of sort `Configuration`, and is a *multiset* of objects and messages. Multiset union is denoted by an associative and commutative juxtaposition operator, so that rewriting is *multiset rewriting*.

Real-Time Maude specifications are executable, and the tool provides a variety of formal analysis methods. The *timed rewriting* command `(tfrew` t `in time <=` *timeLimit* `.)` simulates *one* of the system behaviors by rewriting the initial state t up to duration *timeLimit*.

Real-Time Maude's *linear temporal logic model checker* analyzes whether *each* behavior satisfies a temporal logic formula. *State propositions* are operators of sort `Prop`, and their semantics is defined by equations of the form

`eq` *statePattern* `|=` *prop* `=` b and `ceq` *statePattern* `|=` *prop* `=` b `if` *cond*

for b a term of sort `Bool`, which defines *prop* to hold in all states t where t `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The model checking command `(mc` t `|=u` *formula* `.)` checks whether the temporal logic formula *formula* holds in all behaviors starting from the initial state t.

3 Megastore-CGC

3.1 Motivation

In Megastore, the strategy for partitioning entities into entity groups depends both on application access patterns and requirements for consistency. For an application requiring consistent access to two entities A and B, A and B must belong to the same entity group. Large entity groups are therefore desired to ensure consistency for many different transactions types. However, since only one concurrent update is allowed per entity group, the system's ability to serve multiple simultaneous users depends on entity groups being relatively small. The following example illustrates that it can be hard (or impossible) to partition the entities such that the required levels of consistency and concurrency are achieved.

Example 1. Consider a hospital with thousands of employees. To enable efficient allocation of personnel to tasks (both planned and emergencies), the hospital wants to use a cloud infrastructure for a shared scheduling system used to assign each employee a status throughout the day. The system should maintain entities $\langle\langle$*employee, time slot*$\rangle$*,* *status*$\rangle$, where each employee has a set of *capabilities* (heart surgery, anesthesia, etc), and where *status* is **booked**, **available**, or **off-duty**. The scheduling system must satisfy the following constraints:

1. An employee can be **booked** for at most 12 hours during a 24-hour period.
2. Emergency preparedness requires having a certain number of **available** employees with a given capability in each time slot. There should, for example, always be an available heart surgeon to deal with emergencies.

Transactions booking personnel therefore need to inspect multiple entities before performing updates. For Constraint 1, other records for the same employee must be inspected. For Constraint 2, records of other employees must be inspected.

The question is how to group the records into entity groups. Grouping all entities into the same entity group would make simultaneous assignments (by different operators) impossible, which is unacceptable. Grouping all entities belonging to one employee into the same entity group allows us to enforce Constraint 1 but not Constraint 2: Let H1 and H2 be the only two **available** heart surgeons at time slot τ, and let two concurrent transactions *Book-H1* and *Book-H2* attempt to book H1 and H2, respectively, at time τ. If H1 and H2 belong to different entity groups, Megastore cannot ensure consistency across H1 and H2. Then, both *Book-H1* and *Book-H2* could see the other heart surgeon as **available**, leading to the violation of Constraint 2.

3.2 Megastore-CGC

In Megastore, the data is a set E of *entities* replicated across a set S of *sites*. E is partitioned into a set $EG = \{eg_1, \ldots, eg_n\}$ of non-empty *entity groups*. A function $R : S \to \mathcal{P}(EG)$ assigns to a site the entity groups it replicates.

In Megastore-CGC, the set of entity groups is partitioned into a set OC of *ordering classes*. A number of entity groups should belong to the same ordering class if consistent transactions across these entity groups are required. Furthermore, for each ordering class, there must be at least one site replicating all entity groups in the ordering class ($\forall oc \in OC \quad \exists s \in S \quad oc \subseteq R(s)$). One of the sites replicating all the entity groups in an ordering class oc is the *ordering site* of oc.

A key observation is that, in Megastore, a site replicating a set of entity groups participates in all updates on these entity groups, and should therefore be able to maintain an ordering on these updates. The idea behind Megastore-CGC is that with this ordering, one site, the ordering site, can validate transactions.

Example 2. The status of heart surgeon h at time slot τ is represented by the entity h_τ, which is part of the entity group e_h representing all time slots of h.

Let t be a transaction, initiated at site s_t, that wants to book h_τ. Since there must always be at least one heart surgeon available, t also reads the status of the

other heart surgeons at time τ. These entities belong to different entity groups. t completes by changing the availability status of h_τ to **booked**, if possible.

Using Megastore, Constraint 2 could be violated if some concurrent transaction t', executing at site $s_{t'}$, attempts to book the only other available heart surgeon h' at time slot τ:

1. t reads the value of h_τ and h'_τ at s_t.
2. t' reads the value of h_τ and h'_τ at $s_{t'}$.
3. t books h_τ. This update is distributed by s_t and applied at all sites replicating h_τ, including $s_{t'}$.
4. t' books h'_τ. This update is distributed by $s_{t'}$ and applied at all replicating sites, including s_t.

This execution, which books both heart surgeons and leaves no heart surgeon for emergencies, is not serializable. Megastore-CGC can ensure also Constraint 2 if we group the entity groups for all employees with a given expertise into the same ordering class: The ordering site of the ordering class *HS* of all heart surgeons orders t and t', and then validates t and t' by checking whether all read operations have seen the most recent updates (according to the given order). In the above scenario, either t or t' would fail this test and be aborted.

Since Megastore-CGC makes explicit and uses the implicit ordering of updates during Megastore commits, Megastore-CGC is essentially piggybacked onto Megastore's commit protocol, which has the following advantages:

- Performance on par with Megastore, as Megastore-CGC does not introduce additional coordination messages or blocking.
- For transactions requiring the consistency level provided by Megastore, fault tolerance is identical to that of Megastore.

3.3 Megastore-CGC Without Error Handling

This section explains the behavior of Megastore-CGC without its fault-tolerance features; i.e., assuming that messages are not lost and that sites never fail.

Megastore-CGC maintains the following additional information:

- A mapping $os : OC \to S$, which assigns to each ordering class oc its *ordering site* $os(oc)$ such that $oc \subseteq R(os(oc))$ for each ordering class $oc \in OC$.
- A function $ol : OC \to Orderlist$, assigning to each ordering class its *ordering list*. Each entry in the ordering list for oc contains the updates on entity groups in oc, together with the updating transaction.

We can select any Megastore site replicating all entity groups in an ordering class oc as the ordering site for oc. The ordering list $ol(oc)$ is replicated, with each site maintaining a projection of $ol(oc)$ of updates to locally replicated entity groups.

The mapping os is stored as a special entity group eg_{os} replicated at all sites. This ensures a consistent view among all participating sites, since the ordering site of an ordering class oc may change when an ordering site fails.

When a transaction t accessing entity group(s) in ordering class oc commits, an entry for t is appended to the list $ol(oc)$ by $os(oc)$. This represents the *ordering* of t in oc, and t can then be validated: its execution is valid if and only if all read operations have seen the most recent update according to $ol(oc)$.

Let t be a transaction with ordering class oc. Megastore-CGC then extends Megastore's commit protocol (see Section 2) as follows:

- In Step 1, t is ordered once the ordering site $os(oc)$ receives t's updates. After ordering, $os(oc)$ *validates* t, using the read set of t as input (the read set is included with the log entry proposal for t, and contains the id of all entities read by t, together with the log position of the version read by t).
- If validation at $os(oc)$ is successful, the updated order is included in the apply-request of Step 3.
- If validation is not successful, the apply-step is replaced by a rollback-step, requesting all participating sites to abort t.

A more detailed description of these steps is given in Appendix A.

3.4 Failure Handling in Megastore-CGC

The transaction ordering must be consistent even when the ordering site fails and/or messages containing ordering information are lost. Our key ideas are:

- Transactions *not* requiring the additional consistency features provided by Megastore-CGC are treated as in Megastore: they are committed regardless of whether Megastore-CGC's validation features are available.
- A new ordering site is chosen if the current ordering site may be unavailable.

The commit protocol of a transaction t may be completed without t being ordered (and validated) by the ordering site. This can happen for several reasons:

1. The ordering site is down (or recovering from failure).
2. The ordering site did not receive the message containing t's updates.
3. The acknowledgment from the ordering site was lost.
4. The site executing t crashed after sending t's updates, and some other site completed the commit protocol for t (this is a feature provided by Paxos).

In this scenario, the apply message for t in Step 3 is sent without the ordering information. The next step depends on the validation requirements of t:

- If t only reads entities from one entity group, recipients of the message register t as *awaiting order* before applying t's updates.
- If t accesses multiple entity groups, t cannot be safely committed, and its updates will be replaced by an empty list of operations.

If the ordering site fails, Megastore-CGC provides a method to reinstate ordering if there is another site replicating all entity groups of the ordering class. The steps of this *ordering site failover* are:

- Let t be a transaction with ordering class oc. If the ordering site $os(oc)$ fails to order t during t's commit, s_t (the original site executing t) initiates an ordering site failover for ordering class oc.
- s_t selects the new ordering site s' from the sites replicating all entity groups in oc. If no such site (except $os(oc)$) exists, the failover procedure is canceled.
- If a new ordering site is available, s_t prepares an update to the special entity group eg_{os}, which contains the current ordering site for each ordering class.
- Once this update is accepted by a majority of sites, the new ordering site s' is elected. The mapping os is updated to $os[oc \mapsto s']$.
- Once elected, s' orders all transactions registered as *awaiting order*. This ordering is included in the *apply* message for the next transaction t'.

4 Formalizing Megastore-CGC

This section presents our formal Real-Time Maude model of Megastore-CGC, which extends and modifies our model of Megastore in [9]. The entire executable formal specification is available at `http://folk.uio.no/jongr/mcgc/`.

We model Megastore-CGC in an object-oriented way, where the state consists of a multiset of site objects and messages traveling between them. Each site is modeled as an object instance of the following class:

```
class Site | entityGroups : Configuration,        localTransactions : Configuration,
             coordinator : EntGroupLogPosPairSet, egOrderings : OrderClassUpdates,
             awaitingOrder : EntGroupUpdateList .
```

The attribute `entityGroups` contains one `EntityGroup` object for each entity group replicated at the site; `localTransactions` contains one `Transaction` object for each active transaction originating at the site; `coordinator` denotes the local coordinator state for each entity group; `egOrderings` contains a list of entries (t, eg, lp) for each ordering class oc, representing $ol(oc)$, where lp is the *log position* of t's update in the transaction log for entity group eg; and `awaitingOrder` is a set of entries (oc, t, eg, lp), used during failures for transactions requiring ordering later.

Each site's copy of an entity group is modeled as an object of the class

```
class EntityGroup | entitiesState : EntitySet,         transactionLog : LogEntryList,
                    replicas : EntityGroupReplicaSet, proposals : PaxosProposalSet,
                    pendingWrites : PendingWriteList .
```

`entitiesState` stores the local version of each entity. `transactionLog` denotes the local copy of the replicated transaction log. A log entry `(`t lp s ol`)` contains the identity t of the originating transaction, the log position lp, the leader site s for the *next* log entry, and the list ol of write operations executed by t. `replicas` denotes the set of sites replicating this entity group; `proposals` denotes the local state in ongoing Paxos processes involving this entity group; and `pendingWrites` maintains a list of write operations waiting to be applied to the `entitiesState`.

A transaction request is a list of current read operations `cr(`e`)` and write operations `w(`e`,`v`)`. Executing transactions are modeled as objects of the class

```
class Transaction | operations : OperationList,      status : TransStatus,
                    reads : EntitySet,              readState : ReadStateSet,
                    writes : OperationList,         paxosState : PaxosStateSet .
```

The attribute `operations` contains the remaining operations in the transaction; `reads` stores the values fetched during read operations; write operations are buffered in `writes`; `status` holds the current transaction status; and `readState` and `paxosState` store transient data during execution.

We assume that the sites are connected by a wide-area network, and we therefore do not assume FIFO delivery between the same pair of nodes.

The dynamic behavior of Megastore-CGC is defined by 72 rewrite rules.

5 Performance Estimation

This section shows how randomized Real-Time Maude simulations can estimate the following performance parameters of Megastore-CGC:

- Average time, per committed transaction, between the request arrives and the response is sent.
- Number of commits, conflict aborts, and validation aborts at each site.

We compare the performance of (our models of) Megastore-CGC and Megastore. With the right system parameters, Real-Time Maude simulations should provide realistic performance estimates. For example, it is shown in [14] that Real-Time Maude simulations of wireless sensor networks give as good performance estimates as dedicated simulation tools. Our system parameters are:

- Frequency and distribution of transaction requests.
- Number of sites.
- Number and size of entity groups and ordering classes.
- Network delay distribution between each pair of sites.
- Network and site failure rates.
- Initial values of the seeds for the random function.

We can easily change these parameters by modifying the initial state in Fig. 1. We use a scenario with three sites, four entities, two entity groups, one ordering class (containing both entity groups), and a set of transaction types reading and writing these entity groups. A local read operation requires 10 ms to complete, according to real-world measurements in [1]. After commit, we assume a delay of 100 ms for each write operation before the new value is available. Two sites, Site 1 and Site 2, are located in the same area, with the third site (RSite) at a more remote location. The probability distribution of the network delays is:

	30%	**30%**	**30%**	**10%**
Site 1 ↔ Site 2	10	15	20	50
Site 1 ↔ RSite	30	35	40	100
Site 2 ↔ RSite	30	35	40	100

Transaction requests are generated randomly at each site according to the following frequency distribution (where "Book $H1_A$" is a transaction that also reads the entity $H2_A$ ("heart surgeon H2 in the afternoon") before possibly booking (heart surgeon) H1 in the afternoon):

```
eq initState(N) =
{< RSite : Site |
    awaitingOrder : noAwaitingOrderSet, coordinator : ..., egOrderings : ...,
    entityGroups :
      (< H1 : EntityGroup | pendingWrites : emptyPWList, proposals : emptyProposalSet,
                            replicas : ..., entitiesState : ..., transactionLog : ... >
       < H2 : EntityGroup | ... >
       < OrderSites : EntityGroup | ... >), --- special entity group representing the map OS
    localTransactions : none, seqGen : 0 >
 < Site1 : Site | ... >
 < Site2 : Site | ... >
 < NWRK : NetworkDelays |
    connections : (conn(Site1 <-> RSite,< 1 ; 30 ; 30 > ... < 91 ; 100 ; 100 >, true) ;
                   conn(RSite <-> Site2, ... , true) ; conn(Site1 <-> Site2, ... , true)) >
 < rnd : Random | seed : N >
 < stats(Site1): SiteStatistics | avgLatency : 0, commits : 0,
                                  conflictAborts : 0, validationAborts : 0, ... >
 < stats(RSite): SiteStatistics | ... >       < stats(Site2): SiteStatistics | ... >
 < transGen(RSite): PoissonTransGen | idCounter : 1, status : waiting(10),
    workload : < 1 ; 25 ; update-H1-M > ... < 76 ; 100 ; book-H1-A > >
 < transGen(Site1): PoissonTransGen | ... >    < transGen(Site2): PoissonTransGen | ... > >}
```

Fig. 1. An initial state in our simulations (with parts of the term replaced by '...').

Site 1	Site 2	Remote site
Update $H1_M$ 50%	Update $H1_M$ 25%	Update $H1_M$ 25%
Update $H1_A$ 50%	Update $H1_A$ 25%	Update $H1_A$ 25%
	Update $H1_M$ 25%	Update $H2_A$ 25%
	Book $H2_A$ 25%	Book $H1_A$ 25%

We add "record" objects that record events during the simulation, using techniques in [14]. The initial state `initState`, shown in Figure 1, is then a multiset containing: one `Site` object for each site; one `NetworkDelays` object containing the network delay distributions; one `Random` object with the seed used to randomly select a network delay when a message is sent; one `SiteStatistics` object for each site recording statistics during simulation; and a `PoissonTransGen` object for each site, which generates transactions randomly according to the given distribution.

We simulate the system up to 1,000,000 ms using the command

```
(tfrew initState(10) in time <= 1000000 .)
```

which returns the term (with parts of the term are replaced by '...')

```
{< stats(RSite): SiteStatistics | avgLatency :  94579/631, commitCount : 631,
                                  conflictAborts : 171, validationAborts : 10, ... > ... }
```

in 145,957ms cpu time on a Pentium Intel Core i7 2,6 GHz.

We have also run these experiments on our model of Megastore, and show the result when the average (overall) transaction rate is 2.5 TPS (transactions per second). The following table shows the number of transactions successfully committed (Comm.), and aborted due to conflict (Abs.), and the average transaction latency (Avg.lat). For Megastore-CGC, we also show the number of transactions aborted due to validation failures (Val.abs), since the transactions `book-H1-A` and `book-H2-A` access multiple entity groups and could see an inconsistent read set.

	Megastore			Megastore-CGC			
	Comm.	Abs.	Avg.lat	Comm.	Abs.	Val.abs.	Avg.lat
Site 1	652	152	126	660	144	0	123
Site 2	704	100	118	674	115	15	118
RSite	640	172	151	631	171	10	150

We have also compared the performance on "Megastore-friendly" transactions where each transaction only accesses a single entity group. The performance of Megastore and Megastore-CGC is virtually the same in this experiment:

	Megastore			Megastore-CGC			
	Comm.	Abs.	Avg.lat	Comm.	Abs.	Val.abs.	Avg.lat
Site 1	684	120	122	679	125	0	120
RSite	674	138	132	677	135	0	130
Site 2	693	111	110	691	113	0	113

We also used simulations *during* the development of Megastore-CGC to estimate the performance of different design choices. For example, our experiments showed that aggressive failure detection may increase the number of validation aborts, since ordering may be quicker re-established in case of small transient errors (such as message losses) than if a failover is required.

6 Model Checking Verification

We use *model checking* to explore *all possible* behaviors of Megastore-CGC that can happen nondeterministically from a given initial system configuration. In addition to verifying desired properties, model checking is invaluable *during* the design process, and helped us discover many subtle bugs in (earlier versions of) Megastore-CGC that were not uncovered during extensive simulation.

We analyze the original nondeterministic model (not the randomized one used for performance estimation). For the model checking analysis to terminate, we analyze scenarios with a limited number of transactions, and restrict the message delays, transaction start times, site and communication failures, etc.

With a finite number of transactions, the system should satisfy the property that in all states from some point on:

1. All transactions have finished their execution.
2. All replicas of an entity have the same value or the coordinator of diverging site(s) is invalidated.
3. All logs for an entity group contain the same entries, unless a coordinator is invalidated.
4. The execution was serializable; i.e., it gives the same result as some execution in which the transactions are executed one after the other.

This property can be formalized as the following temporal logic formula Φ:

```
<> [] (allTransFinished /\ entityGroupsEqualOrInvalid
      /\ transLogsEqualOrInvalid /\ isSerializable)
```

`allTransFinished` is a state proposition that is `true` in a state if all transactions have finished; `entityGroupsEqualOrInvalid` is a state proposition that is `true` in all states where all replicas of each entity have the same value, unless the coordinator has been invalidated; and `transLogsEqualOrInvalid` is `true` when all transitions logs for each entity group are equal (unless a coordinator has been invalidated). The last of these propositions is defined as follows:

```
op transLogsEqualOrInvalid : -> Prop [ctor] .
ceq {REST
     < S1 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                   entityGroups : < EG1 : EntityGroup | transactionLog : LOG1 > ... >
     < S2 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                   entityGroups : < EG1 : EntityGroup | transactionLog : LOG2 > ... >}
    |= transLogsEqual = false if LOG1 =/= LOG2 .
eq {SYSTEM} |= transLogsEqualOrInvalid = true [owise] .
```

We first characterize the states where `transLogsEqualOrInvalid` does *not* hold, namely, the states with two sites with valid coordinators and where some entity group `EG1` has different values. The last equation, with the `owise` ("otherwise") attribute, defines `transLogsEqualOrInvalid` to be `true` in all other states.

To analyze serializability, we use the technique in [9]. The *serialization graph* for an execution of a set of committed transactions is a directed graph where each transaction is represented by a node, and where there is an edge from a node t_1 to another node t_2 iff the transaction t_1 has executed an operation on entity e occurs *before* transaction t_2 executed an operation on the same entity, and at least one of the operations was a write operation. An execution of multiple transactions is serializable if and only if its serialization graph is acyclic [20].

In a multi-versioned replicated data store like Megastore-CGC, we need a *version order* $<<$ on the written entity values to decide the *before* relation when constructing the serialization graph. For example: a write operation `w(`e`,`v`)` which creates a version k of entity e occurs *before* a current read `cr(`e`)` iff `cr(`e`)` reads a version l where $k << l$ according to the selected version order. Since every committed transaction is assigned a unique log position for each entity group it updates, we use log positions for the version order. This means that if, for example, t_i reads from log position lp and t_k commits an update at log position lp', then $t_i \rightarrow t_k$ in the serialization graph iff $lp < lp'$.

When an update transaction t_i commits, it produces a message containing:

- the log position and value of each entity it has read; and
- the set of entities written, all of them have the log position assigned to t_i.

We add a `TransactionHistory` object containing the current serialization graph. When a transaction commits, this object reads the above message and updates its serialization graph. The proposition `isSerializable` is then defined

```
op isSerializable : -> Prop [ctor] .
eq {< th : TransactionHistory | graph : GRAPH > REST}
          |= isSerializable = not hasCycle(GRAPH) .
```

We have model checked the temporal logic formula Φ with a number of different system parameters. For example, we have executed the command without

site and communication failures, where the message delay is either 30 or 80, with 5 transactions, in the following setup:

Site	Transaction	Operations	Start time
Site 1	update-H1-A	read H1-A; write(H1-A, $Avail_1$)	150
RSite	update-H2-A	read H2-A; write(H2-A, $Avail_2$)	150
Site 2	update-H2-A	read H2-A; write(H2-A, $Avail_3$)	150
RSite	book-H2-A	read H1-A; read H2-A; write(H2-A, $Booked_1$)	$\{180, 210\}$
Site 2	book-H1-A	read H2-A; read H1-A; write(H1-A, $Booked_2$)	$\{180, 210\}$

We then use the following command to check whether each behavior satisfies the desired properties in Megastore-CGC:

```
(mc init1 |=u Φ .)
```

which returned `true` in 124 seconds cpu time. The number of different states reachable from the initial state is 108,279.

Performing the exact same model checking in Megastore returns the following counterexample, in which there is both an edge from `book-H1-A` to `book-H2-A` and from `book-H2-A` to `book-H1-A` in the serialization graph:

```
Result ModelCheckResult :  counterexample({initTransactions
  ...
< th : TransactionHistory | graph : < book-H2-A ; book-H1-A > ; < book-H1-A ; book-H2-A > ; ...>})
```

Real-Time Maude outputs a behavior invalidating Φ when model checking fails; this allowed us to easily identify the (often subtle) issues causing problems.

We have also successfully model checked Megastore-CGC in a number of other scenarios, including:

- Three transactions, two possible start times, one site failure and fixed message delay (1,874,946 reachable states, model checked in 6,311 seconds).
- Three transactions, two possible start times, fixed message delay and one message failure (265,410 reachable states, model checked in 858 seconds).

7 Related Work

Data stores such as Amazon's Dynamo [7], Google's BigTable [3], and Cassandra [10] are widely used due to their combination of high availability and scalability. However, given their lack of transaction features, several data stores with (limited) transaction support have emerged to address the need for strong consistency in many real-world applications. In addition to Megastore, ElasTraS [6], Spinnaker [16], Calvin [19], and Microsoft's Azure [2] achieve high availability and scalability by partitioning the data, and provide consistency *within* each partition. Both Megastore, Spinnaker, and Calvin use Paxos to distribute updates among sites. We are not aware of any generic method for transactional consistency *across* partitions besides Megastore-CGC. Google's Spanner [5] provides both high availability, scalability, and transactional consistency across partitions, but is less generic since it demands a complex infrastructure involving GPS hardware and atomic clocks.

We have not seen any other work on formalizing and verifying transactional data stores using formal verification tools. In [15] the authors assert the need for formal analysis of replication and concurrency control in transactional cloud data stores, and they analyze a prose-and-pseudo-code description of a Paxos-based concurrency control protocol. In contrast to our work, this description is not amenable to model checking and simulation.

A prerequisite for extending Megastore is to have detailed knowledge of it, which is a challenging task, since Megastore is an internal system at Google that is publicly described only in an informal way in [1]. In [9] we therefore develop a fairly detailed Real-Time Maude model of Megastore. The value of using Maude [4] (the "untimed" version of Real-Time Maude) for formally analyzing other cloud systems is demonstrated in [17], where the authors point out possible bottlenecks in a naïve implementation of ZooKeeper for key distribution, and in [8], where the authors analyze denial-of-service prevention mechanisms.

8 Concluding Remarks

We have used Real-Time Maude to develop an extension of Megastore, denoted Megastore-CGC, which provides consistency also for transactions that access multiple entity groups.

The main idea behind Megastore-CGC is that in Megastore, sites replicating multiple entity groups implicitly observe an ordering of updates *across* this set of partitions. We make this ordering explicit by defining *ordering sites*. An important advantage of Megastore-CGC is that ordering and validation is piggybacked onto the existing message interactions of Megastore's commit protocol, allowing Megastore-CGC to provide these features without introducing new messages or waiting. This is also reflected in our Monte Carlo simulations, which indicate that the performance of Megastore-CGC is virtually the same as that of Megastore.

The Megastore-CGC approach might be applicable to other Paxos-based transactional data stores such as Spinnaker [16] and Calvin [19]. However, one key assumption in Megastore is that each site has a *coordinator* which knows whether the local site has received all updates. Without this feature, changing the ordering site (in case of failure) becomes significantly more complex.

Designing and validating a sophisticated protocol like Megastore-CGC is very challenging. Real-Time Maude's intuitive and expressive formalism allowed a domain expert (the first author) to define both a precise, formal description and an executable prototype in a single artifact. Simulating and model checking this prototype automatically provided quick feedback about both the performance and the correctness of different design choices, even for very complex scenarios. Model checking was especially helpful, both to verify properties and to find subtle "corner case" design errors that were not found during extensive simulations.

References

1. Baker, J., et al.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR (2011), `http://www.cidrdb.org`

2. Campbell, D.G., Kakivaya, G., Ellis, N.: Extreme scale with full SQL language support in Microsoft SQL Azure. In: SIGMOD 2010, pp. 1021–1024. ACM (2010)
3. Chang, F., et al.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2), 4:1–4:26 (2008)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Corbett, J.C., et al.: Spanner: Google's globally-distributed database. In: OSDI 2012. USENIX (2012)
6. Das, S., Agrawal, D., Abbadi, A.E.: ElasTraS: An elastic transactional data store in the cloud. In: USENIX HotCloud. USENIX (2009)
7. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41, 205–220 (2007)
8. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 78–93. Springer, Heidelberg (2012)
9. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Futatsugi Festschrift. LNCS, vol. 8373, pp. 494–519. Springer, Heidelberg (2014)
10. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40 (2010)
11. Lamport, L.: Paxos made simple. ACM Sigact News 32(4), 18–25 (2001)
12. Munir, H., Moayyed, M., Petersen, K.: Considering rigor and relevance when evaluating test driven development: A systematic review. Inform. Softw. Techn. (2014)
13. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
14. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410(2-3), 254–280 (2009)
15. Patterson, S., et al.: Serializability, not serial: concurrency control and availability in multi-datacenter datastores. Proc. VLDB 5(11), 1459–1470 (2012)
16. Rao, J., Shekita, E.J., Tata, S.: Using Paxos to build a scalable, consistent, and highly available datastore. Proc. VLDB 4(4), 243–254 (2011)
17. Skeirik, S., Bobba, R.B., Meseguer, J.: Formal analysis of fault-tolerant group key management using ZooKeeper. In: Proc. CCGRID. IEEE (2013)
18. Stonebraker, M., Cattell, R.: 10 rules for scalable performance in 'simple operation' datastores. Commun. ACM 54(6), 72–80 (2011)
19. Thomson, A., et al.: Calvin: Fast distributed transactions for partitioned database systems. In: Proc. SIGMOD 2012. ACM (2012), http://doi.acm.org/10.1145/2213836.2213838
20. Weikum, G., Vossen, G.: Concurrency Control and Recovery in Database Systems. Morgan Kaufman (2001)

A Transaction Commit in Megastore-CGC

Let t be a transaction executing at site s_t, reading a set of entity groups EG and updating an entity group $eg \in EG$. All entity groups in EG belong to ordering class oc. The table below summarizes the steps of committing t in Megastore-CGC, and distinguishes the features of Megastore from the features of our CGC extension. In the table, R_{eg} denotes all sites replicating eg.

Step	Site(s)	Megastore	CGC extension
1a	s_t	Send an *acceptLeader* request to the leader s_l for the current log position.	If $s_l = os(oc)$, include t's read set and request ordering and validation from s_l.
1b	s_l	Receive *acceptLeader* request. If there are no conflicting updates within eg, send accept to s_t. Otherwise, request s_t to abort t.	If $s_l = os(oc)$ and there are no conflicting updates in eg, order and validate t by appending t's updates to $ol(oc)$ and then verifying that t has seen the most recent update for each member of EG. If validation is successful, $ol(oc)$ is included in the accept message. If validation is unsuccessful, request s_t to abort t.
1c	s_t	Receive response from s_l. If s_l requests abort, t is aborted. Otherwise, multicast an *accept* request for t to all sites replicating entity group eg, except s_t and s_l.	If $s_l \neq os(oc)$ and $s_t = os(oc)$, order and validate t. If validation is successful, s_t requests accept from the other sites. Otherwise, t is aborted. If $s_l \neq os(oc)$ and $s_t \neq os(oc)$: include t's read set in the *accept* request for $os(oc)$.
2	$R_{eg} \setminus \{os(oc), s_t, s_l\}$	Receive and store the *accept* request, send acknowledgment to s_t.	
2'	$os(oc)$ if $os(oc) \neq s_t \wedge os(oc) \neq s_l$	Receive and store the *accept* request, send acknowledgment to s_t.	Order and validate t. If validation is successful, include $ol(oc)$ in the acknowledgment message. If validation is unsuccessful, the acknowledgment is sent without including the ordering.
3	s_t	Multicast *apply* message containing t's updates.	If t was successfully ordered and validated, include $ol(oc)$ in this message. Otherwise, replace t's updates with an empty list of operations (effectively aborting t).
3'	R_{eg}	Apply t's updates to local transaction log and replicated entity store.	If the apply message contains $ol(oc)$, update the local copy of $ol(oc)$.

Some further comments on the CGC extension:

- t is ordered when the ordering site $os(oc)$ *accepts* t. If $os(oc)$ is the leader for this log position, this occurs at Step 1b. Otherwise, it occurs at Step 2'.
- After ordering, $os(oc)$ validates t, using the read set of t as input. The read set is included in the accept-request for $os(oc)$, and contains the id of all entities read by t together with the version seen (represented by the log position). The validation procedure ensures that for any pair of transactions in a read-write conflict (i.e., one is reading and the other is writing the same entity), one of the transactions is aborted unless the conflicting operations occur according to the order $ol(oc)$. Assuming transactions access entity groups within one ordering class only, this is sufficient to verify that the serialization graph [20] for any schedule is acyclic.[1]
- If validation at $os(oc)$ is successful, site s_t distributes the updated order to all sites replicating eg as part of the apply message for t. If validation is not successful, the apply-step is replaced by an empty operation list, effectively aborting t (Step 3).

[1] Megastore is a multi-version data store where write-write conflicts do not occur.