Transactional Data Management for Multi-Site Systems

New Approaches and Formal Analysis

Doctoral Dissertation by

Jon Grov

Submitted to the
Faculty of Mathematics and Natural Sciences
at the University of Oslo in partial
fulfillment of the requirements for
the degree ph.d. in Computer Science

January 31, 2014
Abstract

High-impact systems, notably systems used in health care, public infrastructure, traffic control, and finance, depend on a data management facility that can tolerate many types of failure. In addition, the prevalent adoption of cloud systems increase the demand for commodity services to provide consistent and efficient data storage services. For commodity services, strong fault tolerance and scalability are necessary.

However, achieving the desired level of fault tolerance requires multi-site replication: multiple copies of data are stored at geographically distant sites. Combining such data replication with the transactional consistency guarantees provided by a traditional database system is challenging, and users are usually required to choose between consistency, performance, and fault tolerance.

Google’s Megastore is among the most mature multi-site data stores providing transactions. However, one challenge with Megastore is that it requires data to be grouped into a set of relatively fine-grained partitions, and transactional consistency is only provided within one partition. In some usage scenarios, this represents a significant disadvantage.

The main contributions of this thesis are: 1) three new approaches for transactional multi-site data management, including an extension of Megastore to provide cross-partition consistency; 2) a formal-methods-based analysis strategy using Real-Time Maude to assess both performance and correctness; and 3) a formal model of Megastore that provides the first reasonably detailed publicly available description of the Megastore approach to data management in multi-site data stores.
First and foremost: Thanks to Peter, who adopted this fading project. He has supported me tirelessly through days, weekends, nights, and holidays, and allowed me to share both his sense of humor and unmatched combination of competence, experience, and attention to detail. And thanks to Elin, who, besides making life fun, has been a single parent for weeks before deadlines. And to Ada and Herman, first of all for being you, but also for reminding me that creativity is impossible without trial and error. To all of you: We made it!

In addition to these four, this work is the result of contributions from a large group of people. First of all Mamma, who has donated several vacation days to research. Thanks also to my father, the rest of my family (including in-laws) and friends for their spiritual and practical support.

Thanks to Ellen and Stein for their kind advice and assistance, and to all my colleagues and friends in Braga. In particular, Luís, Alfrânio, Rui and Orlando: You made an impressively energetic and friendly environment for us, and I am forever grateful, both for your hospitality and for what I learned. To my former colleagues at Ifi: I enjoyed your company.

Finally, my warmest thanks to the late Ragnar Normann. With his combination of discipline and positive attitude, he will forever stay with me as a role model.
Contents

Part I Overview

1 Introduction........................................................................................................ 11

2 Background on Replicated Data Management............................................ 15
   2.1 Replicated Data and Transactions .......................................................... 15
       2.1.1 Managing Replicated Data .............................................................. 16
       2.1.2 Concurrency Control on Replicated Data ........................................ 18
       2.1.3 Ensuring Atomicity ........................................................................... 19
   2.2 Trade-offs for Multi-Site Replication ...................................................... 20
   2.3 Existing Solutions for Multi-Site Replication .......................................... 22

3 Background on Real-Time Maude ................................................................. 25
   3.1 Real-Time Maude Modeling .................................................................... 26
   3.2 Formal Analysis ....................................................................................... 31
       3.2.1 Simulation ....................................................................................... 32
       3.2.2 Model Checking ............................................................................. 32

4 Our Contributions .......................................................................................... 35
   4.1 Paper 1: A Pragmatic Protocol for Database Replication in
       Interconnected Clusters ........................................................................... 37
   4.2 Paper 2: Scalable and Fully Consistent Transactions in the Cloud
       through Hierarchical Validation ................................................................ 40
   4.3 Paper 3: Formal Modeling and Analysis of Google’s Megastore in
       Real-Time Maude .................................................................................... 42
   4.4 Paper 4: Increasing Consistency in Multi-Site Data Stores:
       Megastore-CGC and its Formal Analysis ............................................... 43

5 Discussion ....................................................................................................... 47
Part II Research Papers

Paper 1: A Pragmatic Protocol for Database Replication in Interconnected Clusters ........................................ 59

Paper 2: Scalable and Fully Consistent Transactions in the Cloud through Hierarchical Validation ..................... 69

Paper 3: Formal Modeling and Analysis of Google’s Megastore in Real-Time Maude ........................................ 85

Paper 4: Increasing Consistency in Multi-Site Data Stores: Megastore-CGC and its Formal Analysis ................. 113

Part III Appendices

A Real-Time Maude Example ........................................ 129

B Real-Time Maude Model of Megastore .............................. 133

C Real-Time Maude Model of Megastore-CGC ................. 169
Part I
Overview
Chapter 1
Introduction

The U.S. National Institute of Standards and Technology defines high-impact systems to be those systems where “the disruption of access to [...] an information system could be expected to have a severe or catastrophic adverse effect on organizational operations, organizational assets, or individuals” [45]. Such systems exist in most organizations, and in particular those involved in energy production, health care, defense, supply chain, transportation, and finance.

For high-impact systems, robust and reliable data management facilities are a crucial requirement. High-impact systems normally require replication: multiple copies of the same data must be maintained at independent servers, to reduce the probability of disruption of access, e.g., due to hardware failure or manual errors. Furthermore, since high-impact systems usually require disaster tolerance, some of these copies should be held at physically separate sites.

High-impact systems requiring fault tolerance are not the only systems benefiting from multi-site data replication: for services requiring scalability, multi-site replication allows the load to be distributed among multiple sites. Another increasingly important factor is location: users expect modern cloud services to be accessible with the same performance from Paris and from Tokyo, and cloud-based data management facilities are expected to support this.

For most non-trivial data management purposes, database systems have been the dominating approach, partly because of a standardized interface to store, retrieve, update, and search data, but also because transaction support allows robust and consistent execution of groups of operations. Databases are usually multi-user; and a transactional database system provides safe and uniform access for concurrent operations, where uncontrolled access may lead to severe inconsistency errors.

However, database products that combine multi-site replication with transactions are uncommon in practice. The main reason is that the network communication between distant data centers introduce multiple challenges, among the most important are performance and fault tolerance. Performance is reduced since the
coordination required for transaction commit must occur across slow wide area network links. Fault tolerance is complicated, since the failure scenarios in this environment are much more complex to deal with. Message delay is variable, requiring longer timeouts; there is a chance of network partitions, where several sites are available but unable to communicate with each other; and automatic failure detection consequently becomes nearly impossible [19] (since a site cannot know whether a remote site is down, or only unreachable). These challenges are popularly summarized in the CAP Theorem [5], which states that consistency, availability, and (network) partition tolerance cannot be combined; any solution must choose two among the three.

Despite these challenges, high availability combined with large scale is a requirement for many systems. This drives system developers to implement multi-site replication with reduced consistency, effectively moving the responsibility for consistency to application developers and system operators [79, 63]. Notable examples include online retail and commerce services such as Amazon [1] and eBay [17], collaborative services such as Facebook [18] and Twitter [77], and media services such as Netflix [46] and Spotify [72], but it is also increasingly relevant for other critical areas, e.g., system involved in manufacturing, traffic monitoring, and health care [44, 66].

While this strategy works in some cases, it increases complexity and requires careful analysis and testing both during design, development and operation. As stated by Michael Stonebraker [73]:

\[ \text{It is possible to build your own ACID semantics [...], given enough additional code. However, the task is so difficult, we wouldn't wish it on our worst enemy. If you need ACID semantics, you want to use a DBMS that provides them; it is much easier to deal with this at the DBMS level than at the application level.} \]

The challenges of providing consistency at the application level not only increase cost. Another, equally important, challenge is trust: data consistency errors can go undetected for a long time, possibly forever. Therefore, system consistency (also in the presence of failures) must be as transparent and well documented as possible. For some systems (such as trading systems, financial applications, payroll systems, or government registries), trust in consistency is essential, and given a trade-off between consistency, performance and availability, they always choose consistency first.

The topic of this thesis is strategies to improve transactional consistency in the presence of multi-site replication. Such strategies require fault-tolerant data management protocols for replicated data. These are highly complex artifacts, and still it is crucial to establish their performance and correctness. We advocate that to achieve this, formal methods should be used to analyze and improve such protocols.
Complex protocols are typically described using a combination of English prose and pseudo-code. A prototype of the protocol is then usually implemented in a programming language like Java. Finally, the correctness of the protocol is “proved by hand.” This methodology has a number of disadvantages, including:

1. The prose + pseudo-code description is ambiguous and imprecise, and does not make explicit critical assumptions (or lack thereof).
2. A Java prototype is an additional artifact beyond the specification of the protocol, and we must somehow ensure that this Java prototype is consistent with the specification.
3. Proofs “by hand” tend to be error-prone. Indeed, absent a precise and unambiguous definition of the protocol, there is in principle nothing that can be formally proved. Furthermore, the size and complexity of the systems that are the topic of this thesis make the possibility of a serious “hand proof” quite futile anyways.

The remaining chapters of Part I of this thesis are organized as follows. Chapter 2 gives some background on replicated data management, and in particular the challenges related to multi-site replication. It also presents an overview over existing approaches. Chapter 3 presents Real-Time Maude, the formal modeling language and analysis tool which has been fundamental for most of our work. Chapter 4 presents the contributions of the thesis, and Chapter 5 contains a discussion of advantages and disadvantages of our different approaches.

Part II of the thesis contains four research papers, each presenting and analyzing one approach to multi-site replication management, and Part III contains appendices with executable specifications for two of these approaches.
Chapter 2
Background on Replicated Data Management

This chapter first gives an overview over the requirements for managing replicated data. Section 2.2 discusses the most common trade-offs between availability, consistency, and performance in multi-site replication. Finally, Section 2.3 comments on these trade-offs in relation to currently available solutions for multi-site replication.

2.1 Replicated Data and Transactions

A database is a set of individual items used by one or more applications. Database systems provide application developers and users with operations to store, retrieve, and search data through a standardized interface. Databases are expected to provide transactions, i.e., groups of read and write operations which are executed equivalently to an atomic execution, which means that either all operations are completed successfully, or none are completed (in which case any previous operations are “rolled back”). The expected transactional features of a database system are usually summarized in the acronym ACID: atomic, consistent, isolated, and durable. In addition to atomic execution, databases are expected to support a number of consistency features, such as integrity checking according to a given schema; provide isolation for multiple users, e.g., by ensuring that concurrent transactions do not see each other’s updates; and finally, durability: once the updates are successfully applied, they should be protected against loss, e.g., due to power outages. Databases are usually accessed through standard interfaces, notably SQL.

Replication means that data items are stored at multiple locations. The purpose of data replication in a database is scalability and fault tolerance. Replication increases scalability since it allows horizontal scalability, where more than one single computer (server) is able to process requests. Maintaining copies of the same data at multiple servers is necessary to allow horizontal scalability. Since
horizontal scalability with replicated data only affects reads (write operations must still be applied at all copies), the increase in scalability from replication depends on the ratio between read and write requests.

Replication increases fault tolerance in two ways: (1) in the case of failure, time-to-recovery is decreased/eliminated since it enables failover, where requests in case of failure of one server can be processed by another server replicating the same data; and (2) the chance of data loss due to disasters (such as fire or flood) is reduced considerably if updated copies are available elsewhere.

However, replication increases complexity and introduces several new performance challenges and failure scenarios that must be resolved.

### 2.1.1 Managing Replicated Data

This section outlines the main architectures used for replicating data. We focus on architectures for systems requiring high availability, typically high-impact systems (which, as noted initially, include many systems involved in energy production systems, health care, supply chain, etc.) These systems benefit from replication across a wide area network, and we discuss the particular challenges protocols for managing replicated data (commonly denoted replica management protocols) face in this environment.

Architectures for replicating data among multiple processes can be classified as follows [56]:

- (Distributed) Shared-memory [75], where multiple processing units share the same memory. This is typically implemented in large servers, where hardware supports replicating main memory among independent processing units, e.g., to minimize time of retrieval.
- Shared-disk [2], where multiple processes share disk, typically a disk array. Compared to shared memory, this architecture allows increased horizontal scalability, but the servers must be physically co-located, which reduces fault tolerance. The commercial product Oracle RAC is an example of this architecture [54].
- Shared-nothing [9], where processes can only communicate across network links.

We only consider replication in shared-nothing architectures, and, more precisely, replication in shared-nothing architectures distributed across a wide area network – i.e., with high network latency. This is the most flexible architecture since it allows both “true” horizontal scalability across multiple data centers, and disaster recovery. However, this setting is the most complex, since it introduces challenges related to both coordination and fault-tolerance. In a wide area setting, coordination is always a performance challenge. Furthermore, as stated by
2.1 Replicated Data and Transactions

the CAP Theorem [5], full transactional consistency combined with high availability and tolerance for network partitions is impossible.

Replication in a shared-nothing, wide area environment requires a replica management protocol, and designers of such protocols face a number of significant challenges:

- Concurrency control is more difficult, especially combined with performance and scalability. In a replicated database, sites must exchange messages to prevent concurrent transactions from interfering before a transaction commits. In a wide area setting, such messages have a delay several magnitudes higher than local method calls [81]. This increases transaction latency, i.e., the time from a transaction request arrives until it is successfully completed.

Aside from reduced user experience, increased latency also increases congestion among concurrent transactions: it is well known to application developers that various system resources, such as memory and processes, are held during a transaction’s lifetime both in the database system and in connected application servers [28]. This reduces scalability. In addition, higher transaction latency affects the probability of conflict among concurrent transactions: two transactions conflict if they perform operations on the same data item, and at least one of the transactions issue a write operation. As we discuss in Section 2.1.2, conflicts among concurrent transactions require coordination to avoid inconsistency. Since the probability of conflicts among concurrent transactions increases with higher transaction latency, this represents a significant challenge in multi-site replication [21].

- Atomicity is more complex, error-prone, and with potential performance impact, especially in the presence of failures [5]. Atomic commit means that for update transactions, the updates are either applied at all replicas, or not applied at all. To achieve this, a coordination protocol is required [57]. In a non-faulty environment this is relatively simple, but combining atomicity and fault tolerance is complex, especially since wide area networks introduce a number of new challenges: reliable failure detection in a network is challenging [20], and there is potential for network partitions, where servers replicating the same data see each other as unavailable due to network errors.

Because of the challenges above, database systems combining wide area replication and consistent transaction execution are uncommon in production. Although multiple academic prototypes [71, 65, 39] and some commercial products [55, 11] aim to support wide area replication and transactional consistency, many systems requiring multi-site replication use products which reduce consistency guarantees (e.g., to the level of eventual consistency [79]) in order to maximize scalability and availability [7, 73]. Such products are sometimes denoted data stores to avoid the connotations of “traditional” database systems. Examples of data
store products are Amazon’s Dynamo [16], Cassandra [34], MongoDB [40], and Google BigTable [8]. Note that the distinction between data store products and database products is blurry, and some products, e.g., Google’s Megastore [3] (see Section 4.3), provide limited transactional features without the full feature set of a traditional database. In the following, and unless the distinction is significant, we use the term “database” also to describe a set of data managed by a data store.

2.1.2 Concurrency Control on Replicated Data

Although the application interface is usually more high level, the atomic operations of a database are read or write operations on data. Transactions are used to group these read and write operations. Often, a write operation depends on the value of a preceding read, and it is therefore important that simultaneous transactions are protected against each other. Otherwise, the result may be intolerable inconsistencies, as illustrated by the following example.

Example 2.1. Let $t_1$ and $t_2$ be two transactions where both read and write bank account $x$ to deposit $20. Without concurrency control, the following executions are both possible:

1. $t_1 : \text{read}(x) = 10; \ t_2 : \text{read}(x) = 10; \ t_1 : x := 10 + 20; \ t_2 : x := 10 + 20; \ t_1 : write(x, 30); \ t_2 : write(x, 30)$. In this situation, $t_1$’s deposit is lost.
2. $t_1 : \text{read}(x) = 10; \ t_1 : x := 10 + 20; \ t_1 : \text{write}(x, 30); \ t_2 : \text{read}(x) = 30; \ t_2 : x := 30 + 20; \ t_2 : \text{write}(x, 50); \ \text{abort}(t_1)$. Here, $t_2$ was allowed to read $t_1$’s update, which was later aborted (e.g., canceled by the user).

Methods to protect against such inconsistencies are denoted concurrency control, and such methods are often classified as pessimistic or optimistic [80]. These two classes are described briefly below.

Pessimistic concurrency control

With pessimistic concurrency control, inconsistency is prevented during transaction execution. The most well known pessimistic approach is two-phase locking, where 1) transactions must acquire read-locks or write-locks before executing operations, and 2) all necessary locks must be acquired before any locks are released [80]. Locks are managed by a lock manager, which allows read-locks to be shared while write-locks are exclusive. In a non-distributed setting, two-phase locking is a well known and efficient approach to ensure transaction isolation.
2.1 Replicated Data and Transactions

In a replicated database, two-phase locking is challenging: unless all read and write operations of a transaction are pre-declared, lock acquisition requires at least one network round-trip before every operation. In addition, locking is vulnerable to deadlocks, which are especially hard to detect in a distributed setting [21].

Another pessimistic approach to concurrency control is active replication [67], where transaction requests containing updates are ordered and distributed to all sites before execution. Each site then executes the transaction according to the decided order. This ensures consistency across sites. Active replication is used in some multi-site replica management protocols [74].

**Optimistic concurrency control**

In the optimistic approach, transactions initially execute all operations unrestricted, and before commit, a validation step is required to check the execution for consistency. If committing the transaction could cause inconsistency, the transaction is instead aborted (but may be restarted). The validation procedure is normally based on (logical) timestamps [33]: each transaction \( t \) is assigned a unique timestamp \( ts(t) \), and commit is only allowed if we, for every pair of conflicting operations \( op_i \) and \( op_j \), where \( op_i \) precedes \( op_j \), have \( ts(t_i) < ts(t_j) \). Recall that a pair of operations are conflicting if they access the same data item and at least one of them is a write. In a non-distributed, non-replicated database, timestamps are assigned in the order transactions arrive. In a replicated (and hence, distributed) database, optimistic validation requires transactions to be ordered. Ordering strategies are further discussed in Section 2.3. All four replica management approaches presented in Part II of this thesis use optimistic concurrency control.

### 2.1.3 Ensuring Atomicity

Atomic commit in a replicated database requires agreement among the participating sites, i.e., the sites replicating items updated by the committing transaction. The most well known method for atomic commitment in distributed databases is two-phase commit (2PC) [38]. As the name implies, 2PC contains two phases when committing a transaction \( t \):

1. The voting phase where one of the participating sites, the coordinator (usually the site which initially received \( t \)), sends a can-commit? request to all other sites storing data updated by \( t \).
2. The next phase depends on the outcome of the vote:
If all sites reply “Yes” in the voting phase, \( t \) enters the commit phase, and the coordinator sends the commit(\( t \)) message to all sites.

If one or more sites reply “No” (or does not respond within a given time limit) during the voting phase, the coordinator sends an abort(\( t \)) message to all sites.

Although simple and straightforward, two-phase commit is blocking if the coordinator fails after the voting phase – there is no correct way to determine if the coordinator itself allowed the transaction to commit.

Various variations of the two-phase commit protocol exists (see, e.g., [37]). There is also three-phase commit [68], in which pre-commit phase is injected between the voting phase and the commit phase of two-phase commit. This approach is not prone to blocking, but requires three network round trips to complete a transaction.

Recently, the Paxos [36] family of agreement protocols have gained popularity. Paxos is non-blocking, it is characterized by very strong fault tolerance, and it has the same message delay as two-phase commit in a non-faulty environment [22]. In outline, Paxos passes through the following phases to obtain consensus for a proposed value (in our case, “value” equals commit of transaction \( t \)):

1. Agree on a leader.
2. The leader then proposes a value to the participating sites.
3. Once the proposed value is acknowledged by at least a majority of sites, the leader informs all participants about the decision.

In the presence of failures, this may be insufficient to reach consensus, in which case a new round is initiated where another site becomes the leader. Several optimizations to Paxos exists, e.g., Fast-Paxos [35]. Of particular interest in our setting is the optimization to Paxos used by Google in Megastore [3], where the leader-election-step is included in the agreement of the previous transaction (see also Section 4.3).

### 2.2 Trade-offs for Multi-Site Replication

As the presentation so far illustrates, multi-site data management systems must balance consistency, availability, and partition tolerance. Furthermore, coordination messages in wide area networks increase transactions latency, which affects both performance and scalability (see Section 2.1.1). Therefore, many systems reduce consistency also to minimize coordination delay. Below, we discuss some common trade-offs seen in multi-site data management solutions:
2.2 Trade-offs for Multi-Site Replication

- **Update restrictions.** Concurrency control is greatly simplified if all updates are executed at one site only. *Master-slave* replication means that all updates are executed at one master site, and the changes are then propagated in commit order to the other participating sites. Non-master sites, often denoted *slave* sites, may serve all (predeclared) read-only transactions. Master-slave replication is a common setup in many commercial and free database products, notably Microsoft SQL Server [43], Oracle DB [53], MySQL [52], and PostgreSQL [62]. A major advantage of this approach is simplified concurrency control, since this can be handled locally at the master site. The main disadvantages are related to performance, since all read operations of update transactions must also be executed by the master site, and scalability, since all data must be replicated not only by the master site, but also to any slave which may become new master.

- **Partial replication.** For multi-site databases, maintaining a copy of all data items at all sites may be overly expensive, both in terms of in terms of storage space, and in terms of performance. Consequently, data are replicated at some sites only. Often, different types of data have different replication setup: data requiring very high availability may be replicated at all sites, while data used by one site only may not be replicated at all.

- **Reduced consistency for all transactions.** Given the cost of concurrency control and atomic commit, some applications accept reduced consistency, e.g., reads of stale data or temporary (non-committed) results. This approach works well for social media services such as Facebook [34] and news services such as The Guardian, which typically adopt so-called “NoSQL” data stores [7]. However, for applications requiring higher consistency, this approach incurs higher risk, both in terms of development cost, since more work is required to ensure consistency in applications, and in terms of data integrity, since responsibility for decisions related to consistency requirements is delegated to application developers (and in many cases, system administrators).

- **Consistency only within partitions.** In multi-site data management protocols targeting large-scale cloud systems, consistency is commonly provided only within partitions of the data [64, 3, 15]. This increases *throughput*, since concurrent transactions in different partitions can execute in parallel without coordination. Furthermore, since large-scale databases nearly always apply partial replication [27], it also reduces the number of sites involved in transaction coordination.

- **Reduced fault tolerance.** Even if the probability of site failures is low, replication still adds value by improving scalability for read transactions. Then, consistency and availability can be combined. One common approach for this is “lazy” master-slave replication, where update transactions are allowed to commit at the master site while replicas are updated in the background.
2.3 Existing Solutions for Multi-Site Replication

Several replica management protocols have been proposed based on *group communication middleware* [30]. In such protocols, transaction execution and commit is coordinated among sites using black-box communication primitives such as *atomic broadcast* [14]. An atomic broadcast primitive ensures both that a message is delivered to all available replicas, and that all messages sent with this primitive are delivered in a total order. This can be used to ensure concurrency control and atomic commit, as exemplified by the DataBase State Machine (DBSM) protocol [58], which works as follows\(^1\):

1. Any site may receive transaction requests, and transactions are first *optimistically* executed at their origination site.
2. Once the operations of a transaction \(t\) are completed, the origination site prepares a message \(\text{commitReq}(t)\) containing the *read set* and *read versions* for \(t\), i.e., the globally unique id of all data items read by \(t\) together with the id of the transaction creating the given version; and the *write set* and *write values* of \(t\), which contains the id of all data items written by \(t\) together with the new value.
3. The message \(\text{commitReq}(t)\) is then distributed to all sites using atomic broadcast. This ensures both that all available sites receive the message, and that they receive concurrent messages in the same order, i.e., if \(\text{commitReq}(t_1)\) arrives before \(\text{commitReq}(t_2)\) at some site \(s_k\), then \(\text{commitReq}(t_1)\) precedes \(\text{commitReq}(t_2)\) at all sites.
4. Upon receiving the message \(\text{commitReq}(t)\), each site then performs the same optimistic validation procedure: if \(t\) according to the read versions have seen the most recent value, according to the site’s local transaction log, \(t\)'s updates are applied. Otherwise, \(t\) is aborted.

In this protocol, the challenges of concurrency control and atomic commitment in replicated databases are handled by the atomic broadcast primitive. This has some benefits, since it allows a real system to choose an implementation fit for its purpose, where typical factors influencing the decision depends on the availability of true IP-multicast features, fault detection, strategy for group membership management, etc. Postgres-R [31] was the first research prototype to implement this strategy; notable adoptions are Galera [55] (an extension of MySQL) and Tungsten [11] (an add-on component which is provided both for MySQL and Oracle). Both Galera and Tungsten are open source products with commercial backing. However, although some authors present promising results [29, 39], there

\(^1\) This version of DBSM assumes *full replication* where each site maintains a full copy of the database. Partial replication with DBSM is discussed in [71]
is doubt regarding whether group communication in a wide area setting is feasible in practice [61].

The majority of recent research on multi-site replication targets cloud systems, where scalability and support for partial replication are crucial requirements due to the high “elasticity” required by such systems, i.e., the ability to manage changing and unpredictable load. Among the first research prototypes to address the requirement for elasticity was ElasTras [15]. ElasTras is a transactional overlay designed to run on top of a non-transactional data store such as Bigtable [8]. Concurrency control in ElasTras is handled by partitioning the data and assign one master site to each partition. Transactions are restricted to access one partition only, and concurrency control is handled by the master site. ElasTras provides a valuable collection of methods for fault-tolerant transaction management and dynamic partitioning, but the restriction that transaction are only allowed to access one partition is significant.

Recently, multiple systems based on Paxos have been proposed, both production systems and prototypes [64, 74, 3]. We briefly present each of these below.

Spinnaker [64] is a prototype transactional key-value store developed by IBM, where the data is partitioned according to key ranges, and a replicated transaction log is maintained for each partition. Each partition has an elected leader which executes all update operations, and the log is synchronized by the leader using Paxos. Spinnaker does not support transactions per se, but includes a conditional write operation which takes a version number $v_n$ as argument, and allows the update only if the current version timestamp of the item written equals $v_n$. This allows protection against the “lost updates” [4] problem illustrated in Example 2.1 (in Section 2.1.2).

Calvin [74] is another Paxos-based approach to replica management. In Calvin, data are partitioned and transactions are pre-declared, i.e., the entire set of items read and written must be known before the transaction is executed. This is a significant restriction, but it allows Calvin to apply active replication, where the same updates are executed in the same order at all sites (using Paxos to ensure agreement). This allows pessimistic concurrency control, where validation aborts are avoided.

Megastore [3] is an internal system at Google, and is used by services such as GMail, Android, Google+, and Google App Engine [3, 12]. Megastore works by partitioning the set of data items, denoted entities, into a set of relatively small units denoted entity groups. For each entity group, Megastore maintains a replicated transaction log. Given the size of Megastore, partial replication is a requirement, and an entity group may be replicated at any number of sites within the system. To ensure agreement on the state of the replicated log among the sites replicating a given entity group, Megastore uses a custom coordination protocol.
built on Paxos [36]. We have used Megastore as basis for some of our work on multi-site replica management, and present Megastore further in Section 4.3.

There are other approaches, such as Microsoft’s Azure [6] and Google’s Spanner [12], which provide multi-site transaction processing. Although Azure is known to give good performance [32], publicly available details are scarce. Google's Spanner is another multi-site transactional data store developed by Google, using Paxos for synchronization. Spanner provides desirable features such as full consistency also across partitions. However, to achieve this, Spanner synchronizes time in each data center through a combination of GPS hardware and atomic clocks, which makes this approach less generic.

From the presentation above, it is clear that there is no dominating method to provide transactional consistency for data replicated across multiple sites. Given the requirements of high-impact systems, as noted initially, combined with the prevalent adoption of cloud systems and commodity services where availability and scalability is assumed to be “given”, more work in this field is needed.
Chapter 3
Background on Real-Time Maude

Real-Time Maude [51] is a formal modeling language and high-performance analysis tool for distributed real-time systems. Real-Time Maude is an extension of Maude [10], and is based on a simple, expressive, and intuitive logic called rewriting logic [41]. Real-Time Maude specifications consist of definitions of a set of data types, a set of (instantaneous) rewrite rules specifying the system’s instantaneous local transitions, and a set of tick rewrite rules that model time elapse. Real-Time Maude provides a simple and intuitive language to model and analyze distributed real-time systems in an object-oriented style. Furthermore, Real-Time Maude specifications are executable, which enables a variety of powerful formal analysis methods to be run directly on the formal specification.

My decision to formally describe and analyze multi-site replica management systems in Real-Time Maude were motivated by multiple reasons, including:

- The size and complexity of such systems require using an expressive language.
- Due to my lack of experience with formal methods, the modeling language should be simple, intuitive, and easy to quickly learn without any formal methods background.
- The high number of possible system behaviors, especially related to failure handling, requires automatic tools for fast prototyping and short feedback loops both to monitor performance and correctness.
- Strong in-house expertise was available.

In Real-Time Maude, a specification is a set of modules, where each module formally specifies a real-time rewrite theory \((\Sigma, E \cup A, IR, TR)\), where

- \(\Sigma\) is an algebraic signature; that is, a set of declarations of sorts, subsorts, and function symbols.
- \(E \cup A\) is a set of (possibly conditional) equations. \(A\) a set of equational axioms such as associativity, commutativity and identity, so deduction is performed
26 3 Background on Real-Time Maude

modulo\textsuperscript{1} the axioms $A$, $(\Sigma, E \cup A)$ is a membership equational logic theory \textsuperscript{[42]} which specifies the system’s state space.

- $IR$ is a set of labeled, and possibly conditional, rewrite rules $l : t \rightarrow t' \text{ if } cond$ specifying possible instantaneous (i.e., zero-time) local state transitions.

- $TR$ is a set of tick rules advancing time. Tick rules have the form $l : \{t\} \tau \rightarrow \{t'\}$ if $cond$, where $\tau$ denotes the duration of the rewrite and $\{\}$ is an operator that encloses the entire state.

The Real-Time Maude tool provides automated formal analysis methods such as simulation and model checking. This allows direct analysis of the formal specification, in contrast to other approaches requiring a reference implementation in some programming language.

Section 3.1 explains modeling in Real-Time Maude using a simple example, and Section 3.2 shows how a Real-Time Maude model can be formally analyzed.

### 3.1 Real-Time Maude Modeling

This section uses a simple example representing data items, transactions, and sites to show how distributed real-time systems can be specified in an object-oriented style using Real-Time Maude. A listing of the example model is included in Appendix A. We refer to \textsuperscript{[10, 49]} for more details about the syntax of Real-Time Maude.

Types are defined as sorts, and subtypes are declared using the keyword \texttt{subsort}. We declare below sorts representing items, operations (read or write) and operation lists. We use a special type \texttt{NoOp} to represent “dummy” operations.

```
subsorts Read Write < Op < NoOp < OpList .
```

Operators (or function symbols) are introduced with the \texttt{op} keyword: \texttt{op f : s\textsubscript{1}...s\textsubscript{n} \rightarrow s}. Operators can have user-definable syntax, with underbars ’\_’ marking the argument positions.

Some operators can have equational attributes, such as \texttt{assoc}, \texttt{comm}, and \texttt{id}, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are used by the Maude engine to match terms modulo the declared axioms. An operator can also be declared to be a constructor (\texttt{ctor}) that defines the elements of a sort:

```
op read : Item \rightarrow Read [ctor] .
op write : Item ItemVal \rightarrow Write [ctor] .
```

\textsuperscript{1}Informally, the term \textit{modulo} means “taking into account”.
3.1 Real-Time Maude Modeling


read($i$) denotes an operation reading an item $i$, write($i$, $v$) denotes an operation assigning the value $v$ to item $i$, and the operator noOp denotes a dummy operation. The operator _::_ allows operations to be concatenated into a list of sort OpList. For example, read($x$) :: read($y$) :: write($x$, $v$) defines an OpList containing three operations.

Equations are introduced with keyword eq, or ceq for conditional equations. The mathematical variables in such statements are declared with the keywords var and vars, or can be introduced on the fly in a statement without being declared previously, in which case they have the form var:sort. An equation $f(t_1,\ldots,t_n) = t$ with the owise (for “otherwise”) attribute can be applied to a subterm $f(...)$ only if no other equation with left-hand side $f(u_1,\ldots,u_n)$ can be applied. For example, the following function isReadOnly? checks whether a given operation list only contains read operations:

vars PRED SUCC OPLIST : OpList .
op isReadOnly? : OpList -> Bool .
eq isReadOnly?(PRED :: write(I:Item,IV:ItemVal) :: SUCC) = false .
eq isReadOnly?(OPLIST) = true [owise] .

A group of Maude declarations can be declared as a named module using the keyword mod:

(mod SETUP is
  sorts Item ItemVal .
  subsorts Read Write < Op < NoOp < OpList .
op read : Item -> Read [ctor] .
op write : Item ItemVal -> Write [ctor] .
  vars PRED SUCC OPLIST : OpList .
op isReadOnly? : OpList -> Bool .
eq isReadOnly?(PRED :: write(I:Item,IV:ItemVal) :: SUCC) = false .
eq isReadOnly?(OPLIST) = true [owise] .
endm)

An object-oriented Real-Time Maude module is declared with the keyword omod. Such modules may contain class declarations of the form

class C | att_1 : s_1, \ldots, att_n : s_n .
which declares a class $C$ with attributes $\text{att}_1$ to $\text{att}_n$ of sorts $s_1$ to $s_n$. An object of class $C$ in a given state is represented as a term $<O : C | \text{att}_1 : \text{val}_1, ..., \text{att}_n : \text{val}_n>$ of the (built-in) sort Object, where $O$, of sort Oid, is the object's identifier, and where $\text{val}_1$ to $\text{val}_n$ are the current values of the attributes $\text{att}_1$ to $\text{att}_n$. A subclass inherits all the attributes and rules of its superclasses.

A module includes types defined in another module using the syntax inc <MODULENAME>. The following module TRANSACTIONS declares a class $\text{Trans}$, denoting general transactions, with one attribute $\text{ops}$ of sort $\text{OpList}$, denoting the list of operations in the transaction. Some transactions are update transactions, that update one or more data items. Such transactions are object instances of the class $\text{UpdateTrans}$, that is a subclass of $\text{Trans}$. $\text{UpdateTrans}$ adds the attribute $\text{bufferedUpdates}$, representing locally buffered updates.

\[
\text{(omod TRANSACTION is}
\begin{align*}
\text{inc SETUP .} \\
\text{class Trans | ops : OpList .} \\
\text{class UpdateTrans | bufferedUpdates : OpList .} \\
\text{subclass UpdateTrans < Trans .} \\
\text{endom)}
\end{align*}
\]

Maude also contains built-in support for modeling messages through the sort $\text{Msg}$, where the declaration $\text{msg } m : s_1 ... s_n \rightarrow \text{Msg}$ defines the syntax of the message ($m$) and the sorts ($s_1 ... s_n$) of its parameters.

The state of a concurrent object-oriented system is a term of the sort $\text{Configuration}$, and has the structure of a multiset made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is multiset rewriting supported directly in Maude. Since a class attribute may have sort $\text{Configuration}$, we can have hierarchical objects which contain a subconfiguration of other (possibly hierarchical) objects and messages.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule.

The following module includes the TRANSACTION module defined above, and declares a class $\text{Site}$ whose attribute $\text{transactions}$ is a subconfiguration representing the currently active transactions, and $\text{numComp1}$ which is a natural number counting the set of completed transactions. The message $\text{newTrans}$ is used to inject a new transaction request with id $\text{TID}$ and operations $\text{OPLIST}$. Notice that we have specified a hierarchical object-oriented system where $\text{Site}$ objects contain subsystems, representing multisets of $\text{Trans}$-objects. Finally, the rewrite rule labeled $\text{receiveUpdateTrans}$ specifies the case where a message $\text{newTrans}(\text{TID}, \text{OPLIST})$ is received by a site $\text{SID}$. The $\text{newTrans}$ message is consumed, and as a result, site $\text{SID}$ adds a new $\text{UpdateTrans}$ object to its set of active transactions.

\[
\text{(omod SITE is}
\begin{align*}
\text{class Site | transactions : Configuration .} \\
\text{numComp1 : Natural .} \\
\text{subclass UpdateTrans < Trans .} \\
\text{subclass UpdateTrans < Site .} \\
\text{subclass Site < Site .} \\
\text{receiveUpdateTrans | receiveUpdateTrans : Trans .} \\
\text{endom)}
\end{align*}
\]
3.1 Real-Time Maude Modeling

```maude
inc TRANSACTION .
inc NAT .

var OLIST : OpList . vars SID TID : Oid . var TRANS : Configuration .

class Site | transactions : Configuration, available : Bool, numCompl : Nat .

msg newTrans : Oid OperationList -> Msg .

crl [receiveUpdateTrans] :
  newTrans(TID, OLIST)
  < SID : Site | transactions : TRANS, available : true >
=>
  < SID : Site |
  transactions : TRANS
  < TID : UpdateTrans | ops : OPLIST, bufferedWrites : noOp >
  if not isReadOnly?(OPLIST) .
endom)

By convention, attributes whose values do not change and do not affect the next state of other attributes or messages, such as numCompl, need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as available, can be omitted from right-hand sides of rules.

Real-Time Maude introduces timed modules, where a tick rule is used to advance time in the system. Tick rules are declared on the form

```maude
crl [t] : {t} => {t'} in time u if cond
```

where `{_}` is a constructor of a new sort GlobalSystem and `u` is a term of sort Time denoting the duration of the rewrite.

Tick rules in object-oriented Real-Time Maude modules are typically defined as follows:

```maude
(tomod TIMED-BEHAVIOR is
  pr TIME-DOMAIN .

  var C : Configuration . vars NEC NEC' : NEConfiguration . var T : Time .

crl [tick] :
  {C} => {delta(C, mte(C))} in time mte(C)
```
if mte(C) > 0 \ /
  mte(C) =/= INF.

  op mte : Configuration -> TimeInf [frozen (1)] .
  eq mte(none) = INF .
  eq mte(NEC NEC') = min(mte(NEC), mte(NEC')) .

  op delta : Configuration Time -> Configuration [frozen (1)] .
  eq delta(none, T) = none .
  eq delta(NEC NEC', T) = delta(NEC, T) delta(NEC', T) .
endtom)

For each class in a Real-Time Maude specification, the equations for the \texttt{mte} and \texttt{delta} operations must then be declared as follows: \texttt{mte} returns the remaining time before some instantaneous transition must take place, and \texttt{delta} defines how the elapse of time changes the state of the object.

To account for timed behavior, we redefine the \texttt{TRANSACTION} module as follows, where the \texttt{tomod} keyword declares it as a timed module. The attribute \texttt{nextop} is used to specify the delay before a \texttt{Trans} object is ready to execute the next operation.

(tomod TRANSACTION is
  inc SETUP .
  var OPLIST : OpList .
  vars T1 T2 : Time .
  var TID : Oid .

  class Trans | ops : OplList, nextop : Time .
  class UpdateTrans | bufferedWrites : Oplist .
  subclass UpdateTrans < Trans .

  eq mte(< TID:TransId : Trans | nextop : T1 >) = T1 .
  eq delta(< TID : Trans | nextop : T1 >, T2) =
    < TID : Trans | nextop : T1 monus T2 > .
endtom)

Assuming the \texttt{newTrans} rule has been updated accordingly and a fixed delay of 10 time units per operation, we can now advance transaction execution as follows (the rule itself is assumed to take zero time):

rl [nextOperation] :
  < SID:Site | transactions : < TID:Trans | ops : OPLIST : OP, nextop : 0 > =
3.2 Formal Analysis

With a specification of the form above and some initial state consisting of objects with specific values, Real-Time Maude provides a number of powerful formal analysis methods, including:

1. System simulation up to a certain time by rewriting a given initial system configuration. This is very useful for quick feedback during development.
2. Real-Time Maude rewriting can easily be extended to Monte-Carlo simulations for quality-of-service purposes by using Maude’s built-in random function. For example, it is shown in [50] that Real-Time Maude simulations of wireless sensor networks give performance estimates on par with those provided by dedicated simulation tools.

While very useful, system simulation only analyzes one out of many possible behaviors from a given initial system configuration. Real-Time Maude also provides a number of methods to analyze all possible nondeterministic behaviors from a given initial state, including:

3. Search for (un)desired states. By specifying patterns representing states of interest, Real-Time Maude’s search command can be used to search for states that can be reached within a given time interval from the initial state.
4. Finally, LTL model checking can be used to verify that all behaviors from the initial state satisfy a given temporal logic formula $\phi$. Given that $\phi$ can test for important properties such as liveness and safety, this is a very powerful method to validate the specification, e.g., in a scenario where one or more faults are injected after a certain time. For most systems, the number of reachable states quickly becomes very large, and Real-Time Maude provides time-bounded model-checking to analyze all behaviors only up to a certain duration.

An initial state in our example can then be defined as follows

\[
\begin{align*}
\text{mod INIT is} & \\
\text{inc SITE } & \\
\text{ops t s : } & \to \text{Oid } \quad \text{ops x y : } & \to \text{Item } \quad \text{op v : } & \to \text{ItemVal } \\
\text{op initState : } & \to \text{GlobalSystem } \\
\text{eq initState } & = \\
\text{< s : Site | transactions : none, available : true, numCompl : 0 >} & \\
\text{newTrans(t, read(x) :: read(y) :: write(x, v))} & \\
\end{align*}
\]

This defines an initial state $\text{initState}$ containing one $\text{Site}$ object $s$ and a $\text{newTrans}$ message denoting an incoming transaction $t$ with three operations.
3.2.1 Simulation

The state `initState` can be simulated up to time 20 using the command

```
(tfrew initState in time <= 20 .)
```

resulting in the state

```
Result ClockedSystem :
{< s : Site | available : true,
  numCompl : 0,
  transactions :
    < t : Trans | nextop : 0, ops : noOp > >}
```

declares in time 20

where the operation list of trans t has now been reduced to `noOp`.

3.2.2 Model Checking

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

```
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), `\land`, `\lor`, `\rightarrow` (implication), `\Box` (“always”), `\diamond` (“eventually”), and `\Until` (“until”). The unbounded model checking command

```
(mc t |=u formula .)
```

checks whether the temporal logic formula `formula` holds in all behaviors starting from the initial state `t`. If the reachable state space is infinite, time-bounded LTL model checking, in which each behavior is only analyzed up to a given time bound, can be used to ensure termination of the analysis.

In our example, we can perform model checking to verify that all operations in a transaction are eventually executed in all possible behaviors. We first define the proposition `isComplete` to hold in all states where the transaction object has been created, and where its remaining operations are `noOp`.

```
op isComplete : -> Prop [ctor] .
eq {< SID:Site | transactions:< TID:Oid:Trans | ops:noOp > > SYSTEM}
  |= isComplete = true .
```
3.2 Formal Analysis

We can confirm that in all possible behaviors from the initial state `initState`, we will reach a state where all operations have been executed, using:

\[ (mc \text{initState} |=u <> \text{isComplete}. ) \]

Result `Bool`:
\[ \text{true} \]

For more complex models, model-checking is a very powerful tool, both since it allows inspection of a large number of possible states in short time, and since Real-Time Maude outputs a behavior that does not satisfy the desired property if the property does not hold. This significantly reduces development time even of fairly complex models of distributed systems, both since it simplifies debugging, and since it allows quick “regression testing” to check for bugs each time the specification is changed.

Real-Time Maude has been used to formally specify and analyze a large number of real-world systems and protocols. In our setting, relevant examples include two-phase commit [48], multi-cast protocols [60], wireless sensor network algorithms [50], and scheduling protocols [59]. The paper [48] presents a formal specification and analysis of the two-phase commit protocol (see Section 2.1.3), and by this provides a simple and easy-to-follow example of Real-Time Maude modeling for readers familiar with distributed database systems.
Chapter 4
Our Contributions

The papers presented in this thesis focus on methods to provide consistency, fault-tolerance and performance in transactional multi-site data stores. The main contributions of this thesis are:

- **New approaches for consistency and scalability in multi-site data stores.** We present three new approaches to transaction management in multi-site data stores. Our first approach, WICE [23], targets the performance challenges resulting from the high message latency in wide area networks. WICE addresses the challenge that replica management protocols based on group communication middleware, which show good performance in systems replicated across servers on the same location, do not perform equally well in multi-site replication where coordination occurs across a wide area network. WICE proposes to replace group communication middleware with an optimized, custom synchronization protocol for wide area coordination. We show through simulation studies that this indeed represents a significant improvement. The second approach, FLACS [26], is based on a new method for optimistic concurrency control based on *incremental ordering*. For systems with relatively predictable transaction access patterns, this can significantly improve performance by reducing the need for coordination among distant sites. Our third and final approach, Megastore-CGC [25], is an extension of Google’s Megastore. Megastore only supports consistency within partitions of the data set, and Megastore-CGC extends Megastore to provide consistent transactions also for transactions reading items from different partitions. An important feature of Megastore-CGC is that Megastore’s strong fault tolerance is preserved, and no additional messages are required during normal operation.

- **New modeling techniques and analysis methods to formally analyze and verify replica management protocols.** Three of our proposed approaches are formally specified in Real-Time Maude. We also provide new modeling techniques for Real-Time Maude, e.g., for network infrastructure and fault handling in distributed data stores, together with new methods for formally
analyzing serializability. This enables rigorous testing of important properties such as how the system deals with failures during fault recovery, and whether serializable execution is ensured. We are not aware of previous work using formal methods to model and analyze transaction processing systems of this size and complexity.

- **A formalization of Google’s Megastore approach.** Google’s Megastore is among the largest real-world deployments of a transactional multi-site data store. However, it is an internal system at Google, with only a short, informal description of its design publicly available [3]. Given its success internally at Google, we believe that the Megastore approach can also be successfully applied to other data management systems. To facilitate further research and development of the Megastore approach, we provide a formal specification of a replica management system based on the description in [3]. Furthermore, we provide an in-depth analysis of its performance and correctness, using Real-Time Maude’s simulation and model checking capabilities.

### List of papers


Each paper presents one specific approach to replica management in wide area networks, and then presents an analysis of the protocol. In the following sections, we summarize the contributions of each paper.
4.1 Paper 1: A Pragmatic Protocol for Database Replication in Interconnected Clusters

The paper [23] introduces the WICE (Wide area and Cluster Enabled) replica management protocol. WICE is a new optimistic protocol for transactional multi-site replica management. The main idea behind WICE is that the group communication-based approach used by several successful replica management protocols, notably Postgres-R [31] and DBSM [58], is too costly in a wide area network. In this approach, an optimistic validation procedure is combined with a black-box atomic broadcast primitive. The main issue is that the atomic broadcast primitive “hides” incoming updates from the data management system until the update becomes stable, i.e., when all sites agree to apply the update.

WICE is based on the assumption that in a wide area network with high message latency, remote updates should instead be delivered as soon as possible, and transactions should be allowed to read unstable data. This is an optimization, as it reduces the chance that other transactions are aborted due to reading old versions. Note that consistency is still preserved as long as transactions are blocked until the updates read are stable.

WICE is an optimistic protocol, used to manage a set of data replicated among a set of sites. Each site is assumed to contain one cluster with a number of servers, and WICE uses a custom protocol for communication among clusters (sites). Servers within the same cluster use group communication middleware, which provides attractive features such as automatic group membership management and reliable multicast.

Clients submit transaction requests to servers within each cluster. Any client can connect to any server. One of the clusters, the primary cluster, is responsible for validating transactions before commit. The other clusters are denoted secondary clusters. Each cluster has a delegate server which acts as a proxy to the other clusters.

The steps for executing an update transaction $t$ in WICE are as follows:

1. When local execution is complete, the receiving server requests $t$ to be ordered and validated. Ordering and validation is performed by one server within the primary cluster, denoted the certifier. The validation request is sent to the certifier as a message containing $t$’s updates together with its read set, containing the identifier of all items read by $t$.

---

1 A note on terminology: in [23], we use the term site to represent individual database servers within each cluster. Both in this overview and the remaining papers, a site is used to represent one geographically separate site (which may contain multiple servers), and the terms site and cluster are equivalent. According to this, the correct interpretation of a “site” in [23] is a server.
2. The certifier then orders $t$ against all other transactions executing in the system. This is the validation procedure.
3. Assuming $t$ is successfully validated, the certifier then distributes the updates to each cluster’s delegate, which in turn broadcasts them to all servers within the cluster.
4. A server receiving $t$’s updates acknowledges to the sender, and then applies the updates. At this point, the updates are *unstable*, in which case local transactions are allowed to read, but they are blocked until all read operations have seen (transitively) stable data.
5. Eventually, $t$ is acknowledged by all servers and becomes *stable*.

To demonstrate the effectiveness of the approach, a prototype of WICE was developed in Java and then benchmarked against the atomic-broadcast-based DBSM protocol. The experiments were performed using a simulator developed and hosted by the Distributed Systems Research Group at the University of Minho. In this simulator, real Java implementations of the replication protocol and communication middleware are executed within a simulated model of database software, operating system, network, and hardware components such as CPU and disk. A workload generator, based on the well-known TPC-C benchmark [76], is used to generate transaction requests.

The simulator provides a detailed model of a real database server, where the simulator is calibrated through profiling real system’s CPU usage for similar operations, such as database queries and updates. These data are used to include precise estimates of CPU delay during simulation, as well as to model contention: when executing a given operation, the simulator blocks the simulated CPU the corresponding (measured) time. The protocol prototypes deployed in the simulator are implemented and executed against an abstraction layer which provides job scheduling, clock access, and a simplified network interface [69]. This simulator has been used to compare several atomic-broadcast based replica protocols [29, 47, 70, 13, 78].

Our experiment setup was six servers organized in two clusters of three servers each. Within each cluster we assumed a local area network, while the network link between clusters emulated an inter-continental satellite link (with message delay at 400ms). Simulated clients injected different transaction requests according to a distribution of transaction types and “think times” (delay between requests) given by TPC-C [76]. Our experiments varied the number of clients from 60 to 6000, with clients evenly distributed among servers. As a baseline, the same measurements were performed on the DBSM protocol with all six servers in the same cluster. The DBSM protocol has previously shown good performance compared to a number of other protocols, including Postgres-R, using the same simulator [29].

The results are shown in Figure 4.1. *Throughput* is the number of committed transactions per minute, *latency* is the time between the transaction request
is submitted until it transaction terminates, and *abort rate* is the fraction of received transaction requests that are aborted (due to validation failure). As our results show, the abort rate of WICE is significantly lower than in DBSM, and in particular, in the primary cluster. The study also shows that in a medium-loaded system, the observed latency decreases by 50% in WICE compared to DBSM. In an optimistic protocol, congestion shows up primarily as higher abort rate. Lower transaction latency does not only improve user experience, it also reduces congestion.

Fig. 4.1 Performance results for WICE.
4.2 Paper 2: Scalable and Fully Consistent Transactions in the Cloud through Hierarchical Validation

Paper [26] presents FLACS (Flexible, Location-Aware Consistency), a replica management protocol designed for optimal scalability in multi-site replicated data stores. We assume a set of data items replicated among a set of sites, and FLACS then facilitates combining full consistency with performance: the sites are organized in a (logical) tree structure, and a custom, incremental ordering protocol allows transactions to be validated and committed near (or at) their origination site. This may provide a significant advantage, especially in a multi-site environment with high network latency.

To achieve this advantage, FLACS depends on a certain level of locality in transaction access, i.e., there must be a pattern in the set of items typically requested at a given site. In many real-world systems there is significant such locality, as shown in the following example:

**Example 4.1.** Assume we have a service for online booking of hotel rooms where the bookings are replicated between two sites: one in New York and one in Paris. Then, the majority of bookings seen at the New York site can be expected to request American hotel rooms, while the majority of bookings received in Paris request French hotel rooms.

In this example, an attractive optimization is to allow European bookings to be validated in Paris, while American bookings are validated in New York. Due to the long message transmission time between New York and Paris, this gives a significant reduction in transaction delay.

A naïve implementation of this optimization is to partition the data into two separate databases, and use an independent validator site at each continent (this could, e.g., be implemented using WICE). However, some transactions will access hotel rooms in both Europe and America; to provide consistency for these, partitioning does not work.

FLACS is designed to solve this problem, by providing full consistency for all subsets of data items while allowing “local” validation. This is accomplished as follows:

Sites are structured in a logical validation hierarchy. Data is logically partitioned, and each partition is associated with a set of observers. The observers receive the initial commit request for an update transaction (after local execution at the origination site). The commit request is then propagated upwards in the validation hierarchy through FIFO channels. Throughout this propagation, transactions are incrementally ordered, and validation and commit can occur as soon as the committing transaction has been ordered against all (potentially) conflicting transactions.
4.2 Scalable and Fully Consistent Trans. in the Cloud through Hierarchical Validation

An important requirement is that the validation hierarchy is designed according to the locality patterns, such that for a majority of transactions, validation and commit happen as soon as possible. This improves user experience, but it also improves the abort rate: similarly to the idea behind WICE, reducing the delay between a transaction’s execution and its commit also reduces the chance of validation failures, simply because other transactions are allowed to see the updates sooner.

Note that WICE can be seen as a special case of FLACS where the validation hierarchy is designed such that the root site orders and validates all transaction.

We defined FLACS a Real-Time Maude model. Using this model, we performed a simulation study where FLACS was compared to a WICE-like model where one site was the only observer for all items. Our experiment setup has four sites (London, Paris, New York, and Los Angeles), and the model was calibrated to model real message delays between these cities. The test data was hotel rooms, with multiple clients booking hotel rooms in one or more of the cities.

![Fig. 4.2 Experiment validation hierarchies.](image)

![Fig. 4.3 FLACS vs WICE.](image)

We compared a setup where one site (New York) acted as the “master” observer for all rooms to a setup where rooms were allocated to sites according to the expected locality, as shown in Figure 4.2. Our results, shown in Figure 4.3,
Our Contributions indicate that FLACS represents a significant improvement. However, we observed during our experiments that choosing the wrong validation hierarchy significantly impacts performance.

4.3 Paper 3: Formal Modeling and Analysis of Google’s Megastore in Real-Time Maude

Google’s Megastore is probably the largest existing multi-site transactional data management system. It handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of data across many global data centers [3]. It is used both for Google’s own services such as GMail, Android and Google+ [3, 12], and by Google’s clients through the Platform-as-a-Service offering Google App Engine. Given the size of Megastore, partial replication is a requirement, and data may be replicated at any number of sites within the system. As discussed in Section 2.3, Megastore works by partitioning the data into entity groups and maintaining a replicated transaction log using a custom coordination protocol built on Paxos [36]. Transaction execution is optimistic: a site $s$ receiving a transaction request $t$, updating entities within entity group $eg$, initially executes $t$’s operations locally at site $s$. After $t$’s operations are executed locally, $s$ then proposes $t$ as the next entry for $eg$’s replicated log. The Paxos-based coordination protocol provides concurrency control, by ensuring that if multiple sites propose different transactions for the same log position, only one is chosen (and the other transactions aborted). Moreover, by using Paxos, the coordination protocol ensures agreement even in the presence of complex fault scenarios, e.g., involving multiple sites. This ensures atomic commit.

For practical usage, Megastore provides very strong fault tolerance and a robust design where transactions are allowed to commit even in scenarios with multiple sites failing. Its major disadvantage is that consistency is only provided within partitions. Another possible disadvantage is that Megastore has relatively low performance, especially compared to “performance-focused” multi-site replication protocols such as WICE and FLACS.

Since Megastore is an internal system at Google and has previously only been described informally, we chose to define a formal Real-Time Maude model of Megastore, both to obtain a more detailed understanding of its underlying principles, and to be able to analyze its behavior in different scenarios. In our paper [24], we provide three contributions:

- We present a Real-Time Maude model of (our interpretation of) Megastore, as given in [3]. This model (containing 56 rewrite rules) facilitates further research on the Megastore approach through a clear, unambiguous, and
4.4 Incr. Cons. in Multi-Site Data Stores: Megastore-CGC and its Formal Analysis

thoroughly analyzed specification of Megastore’s features for replica management. A version of the specification allowing timed model checking is provided in Appendix B. The executable model can also be downloaded at http://folk.uio.no/jongr/megastore/maude.html.

- Our model of Megastore is the first publicly available Real-Time Maude model of a distributed data store system, and we have developed novel techniques to model specific characteristics such as optimized message broadcast with variable network delay, and nondeterministic site failures, in Real-Time Maude. In addition, we provide techniques to analyze correctness and serializability using Real-Time Maude.
- Together with the model, we present analysis results of Megastore’s behavior both in the non-faulty scenario and in relatively complex site failure scenarios, using both model checking and simulation.

Often, protocol design flaws are discovered during Real-Time Maude modeling and the following analysis. Since the only available informal description of Megastore is at a high level and does not provide a sufficient level of detail, it was impossible to map flaws found during Real-Time Maude model checking back to the informal description. However, Real-Time Maude’s strong support for exploring protocol behavior in various scenarios has significantly improved the correctness of our Megastore model, and therefore, also significantly improved its value as a contribution to the research community.

4.4 Paper 4: Increasing Consistency in Multi-Site Data Stores: Megastore-CGC and its Formal Analysis

Megastore’s value is proven by its success as an important part of Google’s infrastructure. However, since consistency is only provided within entity groups and only one running update is allowed per entity group, the partitioning of entities into entity groups requires care and effort to avoid reducing either scalability and consistency. On the one hand, if entity groups are too large, scalability is reduced through a higher number of conflicting, concurrent transactions. On the other hand, if entity groups are too small, the probability of inconsistency is increased due to a higher chance of conflicting transactions accessing multiple entity groups.2 This increases maintenance cost and risk for practical deployments, and for some systems, finding the right balance may be impossible.

2 In detail, the trade-off also involves user access patterns and whether the entities have a “natural” grouping. For an email service such as GMail, a partitioning scheme based on user is a natural grouping, while for an online retail service, the partitioning scheme for inventory is less obvious.
In [25], we present an extension of Megastore, denoted Megastore-CGC (Megastore with Cross-Group Consistency). Megastore-CGC is based on the key observation that, in Megastore, a site replicating a set of entity groups participates in all updates on these entity groups. Therefore, this site implicitly has an ordering on these updates. Making this ordering explicit makes it possible to validate the transactions, to ensure that only transactions with a consistent view across multiple entity groups are allowed to commit.

Megastore-CGC allows a set of entity groups to be combined into an ordering class, and consistency is ensured among transactions accessing multiple entity groups if all entity groups belong to the same ordering class. Any set of entity groups may be combined into an ordering class, given that at least one site replicates all entity groups in the set.

This is a significant improvement compared to Megastore, and it is provided without impacting either fault tolerance or performance by piggybacking an ordering and validation protocol onto Megastore's coordination protocol. In outline, Megastore-CGC works as follows:

- For each ordering class, one site is designated the ordering site. As the ordering site receives notification of transactions updating entity groups in the ordering class, it orders the updating transaction, and then performs a validation step to verify that the transaction’s read set is correct according to the given order. The role of the ordering site (within the ordering class) is similar to the certifier of WICE, but since Megastore’s coordination protocol requires each site to vote before a commit decision is made, the validation outcome is included in the ordering site’s vote. This allows us to extend Megastore with cross-entity group consistency without introducing additional messages.

- For “Megastore-friendly” transactions accessing one entity group only, i.e., not requiring validation, Megastore’s fault tolerance features and performance are fully preserved. In addition, we provide fault tolerance for the ordering and validation extension: if the ordering site for an ordering class becomes unavailable, a failover protocol is initiated to elect a new ordering site (among the other sites replicating all entity groups in the ordering class). Transactions requiring validation are preventively aborted until a new ordering site is active.

Megastore-CGC is formally specified in Real-Time Maude, extending the Megastore model we present in Section 4.3. We performed several experiments to analyze both performance and correctness (details are given in [25]). Here, we present the simulations to verify that Megastore-CGC does not reduce performance compared to Megastore. We compared Megastore-CGC with Megastore in two simulation experiments, each simulating 1,000 seconds with 2.5 transactions per second. Both experiments involve three sites Site 1, Site 2, and RSite, where Site 1 and Site 2 are assumed to be located in the same area while RSite
is at a more remote location. In Experiment 1, some of the transactions require cross-entity group validation since they access multiple entity groups. In Experiment 2, all transactions are “Megastore-friendly”, i.e., accessing only one entity group (in which case Megastore and Megastore-CGC should perform equally).

The tables below show the results, with *Comm.* representing the transactions successfully committed, *Abs.* is the number of transactions aborted due to conflict, and *Avg.lat.* is the average transaction latency of committed transactions. For Megastore-CGC, we also show the number of transactions aborted due to validation failures (*Val.abs.*).

### Experiment 1: Cross-entity group transactions

<table>
<thead>
<tr>
<th></th>
<th>Megastore</th>
<th>Megastore-CGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>652</td>
<td>152</td>
</tr>
<tr>
<td>Site 2</td>
<td>704</td>
<td>100</td>
</tr>
<tr>
<td>RSite</td>
<td>640</td>
<td>172</td>
</tr>
</tbody>
</table>

### Experiment 2: Single entity group transactions

<table>
<thead>
<tr>
<th></th>
<th>Megastore</th>
<th>Megastore-CGC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>684</td>
<td>120</td>
</tr>
<tr>
<td>RSite</td>
<td>674</td>
<td>138</td>
</tr>
<tr>
<td>Site 2</td>
<td>693</td>
<td>111</td>
</tr>
</tbody>
</table>

We see that in Experiment 1, the number of commits in Megastore-CGC at Site 1 and RSite is lower than Megastore’s due to validation failures. In Megastore, these transactions are committed, but possibly after an inconsistent execution. Experiment 2 confirms that for transactions accessing one entity group, there are no validation aborts in Megastore-CGC, and the performance of Megastore-CGC equals Megastore.

Our Real-Time Maude model of Megastore-CGC is included in Appendix C. The entire executable model can be downloaded at [http://folk.uio.no/jongr/mcgc/](http://folk.uio.no/jongr/mcgc/).
Chapter 5
Discussion

There is no “silver bullet” approach to transactional multi-site data management. Instead, different use cases require different approaches. This is reflected in the four replica management protocols presented in this thesis: WICE and FLACS focus mainly on performance, while Megastore-CGC and Megastore focus on fault tolerance. In Figure 5.1, we informally compare these approaches in the three dimensions performance, fault tolerance, and consistency.

![Fig. 5.1 Protocol characteristics informally compared.](image)

- **FLACS** is designed for performance and consistency through a validation protocol which ensures both, given proper configuration. However, FLACS requires a trade-off between performance and fault tolerance. In addition, its fault tolerance features are immature and only superficially described. In an environment with frequent network or site failures, the commit protocol of FLACS is vulnerable to blocking.
- **WICE** represents a “compromise” between performance and fault tolerance. Its strongest features are full consistency combined with a simple design and good performance through an efficient optimistic commit protocol. Its weakest point is that the Paxos-based commit protocol of Megastore/Megastore-CGC offer better fault tolerance.

47
• **Megastore**’s strongest advantage is a robust, well-tested protocol to ensure atomic commit even in the presence of major failures. Its main disadvantages are consistency only within partitions, combined with a three-step commit protocol which also restricts updates to one running update per partition.

• **Megastore-CGC** is an extension of Megastore, with similar fault tolerance and performance features. Megastore-CGC has stronger consistency features than Megastore due to its support for cross-entity group validation.

Among the three new approaches presented here, I believe Megastore-CGC has the highest potential as a general purpose approach: it is based on Megastore, which has already proven its success, its features are thoroughly developed and tested using Real-Time Maude, and there is a huge demand for stable, scalable and consistent data management systems. Still, I believe there exist use cases where WICE and FLACS may be a good fit: the WICE approach could be used in interactive planning systems where both highly consistent updates and multisite replication are highly useful features, but where performance of updates is critical. FLACS could be a viable option for applications where data have different SLA requirements, e.g., in e-commerce, where data with a specific “owner” (such as the inventory of a local store) can be combined with order tracking data, and these different data types can have different performance and fault tolerance characteristics based on configuration.

In addition, the ordering and validation features of Megastore-CGC were inspired by FLACS and WICE: the ordering and validation features of WICE are essentially piggybacked on a two-phase commit protocol in a similar manner as Megastore-CGC’s ordering and validation is piggybacked onto Paxos, and the tree structure of FLACS inspired Megastore-CGC’s grouping of entity groups into ordering classes.

Finally, I would like to briefly share my experiences with Real-Time Maude, both for specification and analysis. Initially, I had doubts whether a formal specification in Real-Time Maude adds value over a textual specification combined with a reference implementation written in some programming language (say Java). This is partly because competence in Java is far more common, and partly because development tools are more mature.

However, after some experience with Real-Time Maude, I have no doubts anymore: while a Java implementation must make a lot of explicit assumptions regarding, e.g., control flow, which are not really part of the specification, Real-Time Maude is a declarative language which allows clear, unambiguous statements specifying protocol behavior only. As a result, both the initial development and maintenance of a Real-Time Maude specification is less error-prone and more intuitive. Once past the initial confusion, I also found Real-Time Maude features, such as simulation and model checking, very powerful when working with distributed protocols. This was mainly due to their simplicity and efficiency, but also because
system state is represented \textit{explicitly}. This allows simple recording and inspection of individual states, in addition to traces representing the entire behavior up to a certain (undesired) state. A particular advantage was that Real-Time Maude provides powerful tools to model and analyze complex systems without any previous formal method background, and I believe Real-Time Maude can significantly improve both the development process and result for most organizations working with data management protocols.

References


REFERENCES


REFERENCES


[77] Twitter, Inc. Twitter. URL: http://twitter.com/.


Part II
Research Papers
Paper 1: A Pragmatic Protocol for Database Replication in Interconnected Clusters
A Pragmatic Protocol for Database Replication in Interconnected Clusters


Abstract

Multi-master update everywhere database replication, as achieved by protocols based on group communication such as DBSM and Postgres-R, addresses both performance and availability. By scaling it to wide area networks, one could save costly bandwidth and avoid large round-trips to a distant master server. Also, by ensuring that updates are safely stored at a remote site within transaction boundaries, disaster recovery is guaranteed. Unfortunately, scaling existing cluster based replication protocols is troublesome.

In this paper we present a database replication protocol based on group communication that targets interconnected clusters. In contrast with previous proposals, it uses a separate multicast group for each cluster and thus does not impose any additional requirements on group communication, easing implementation and deployment in a real setting. Nonetheless, the protocol ensures one-copy equivalence while allowing all sites to execute update transactions. Experimental evaluation using the workload of the industry standard TPC-C benchmark confirms the advantages of the approach.

1. Introduction

Database replication is an attractive concept both to increase fault tolerance and to improve scalability by enabling several database sites to serve the same queries. The main challenge of such systems is that coordinating updates among the participating servers inevitably delays the execution of update-transactions. A particularly promising approach is taken by replication protocols based on group communication such as DBSM[12, 7] and Postgres-R[10, 21]. By taking advantage of optimistic concurrency control allowed by transactional semantics and of atomic multicast provided by group communication, it provides performance and scalability even in face of demanding workloads such as the industry standard TPC-C benchmark[17].

Unfortunately, scaling existing cluster based replication protocols to a wide area network is troublesome. Notably, the latency of uniform atomic (or safe) delivery required to ensure fault tolerance has a profound impact in optimistic concurrency protocols leading to increased abort rate[6]. This wastes resources and endangers the ability to commit long lived transactions in a busy server. Although optimistic delivery can mitigate this limitation[16], using it requires an in-depth rewrite of existing protocol implementations. In fact, the only generally available group communication toolkit supporting it is Appia[11, 14].

Furthermore, although research has been addressing group communication in wide area networks for a long time, industrial deployment is far more common in clusters. Therefore one should expect wide area features to be far less tested and optimized, if implemented at all. The overhead of maintaining automatic management of membership spanning multiple geographically apart sites is also not negligible. Finally, the practicality of group communication over wide area networks is also compromised by network configuration and security issues, such as firewalls, tunnels and NAT gateways. In particular, using true multicast for efficiency is often not an option.

In this paper we present WICE, a protocol targeted at multiple clusters interconnected by a wide area network. In contrast with lazy replication protocols, such as Oracle Streams[20], WICE ensures that no globally committed transaction (i.e. which has been acknowledged to clients) is lost. On the other hand, by allowing all replicas to be fully on-line and executing update transactions, it improves resource efficiency and performance when compared to volume replication[18], often the only choice for disaster recovery in mission critical applications.

In detail, the contributions of this paper are the following: (i) introduces the protocol providing 1-copy equivalence of the native database consistency criterion, even in the presence of faults, while confining group communication within LANs and improving practicality, (ii) takes advantage of directly implementing updates stabilization across wide-area directly on TCP/IP to greatly reduce the likelihood of a transaction being aborted during the certification phase, which is the single greatest obstacle to the scalability of previous proposals[6], and (iii) provides an experimental evaluation of the protocol applied to a multi-version database when running the workload of the industry standard TPC-C benchmark[19], thus verifying the previ-
2. System Model

We assume the page model for a database [2]: A collection of named data items which have a value. The combined values of the data items at any given moment is the database state. We do not make any assumptions on the granularity of data items.

A database site is modeled as a sequential process. In detail, the execution of each site is modeled as a sequence of steps that may change the site’s state. Namely, the database state is manipulated by executing \( \text{READ}(x) \) and \( \text{WRITE}(x) \) steps, where \( x \) represents a database tuple. A transaction is a sequence of read and write operations followed by a \( \text{COMMIT}(t) \) or \( \text{ABORT}(t) \) operation. Each site contains a complete copy of the database and is responsible for ensuring local concurrency control.

We consider a finite set of database sites that communicate through a fully connected network. Both computation and communication are asynchronous. Sites may fail only by crashing and do not recover, thus stopping to execute database operations, or send or deliver further messages.

Database sites are organized in clusters. Within a cluster we assume a primary component group membership service that provides current and consistent views of the sites believed to be up [4]. This service is intended to allow, at any moment, the deterministic identification of a distinguished site as the cluster’s delegate as well as providing a view-synchronous multicast primitive (Section 2.2). The availability of a primary component group membership service implicitly assumes that consensus is solvable in our system model [8]. The assumed failure patterns and failure detection capabilities of our model are thus indirectly determined by the actual solution adopted for consensus.

Among clusters, we assume that the failure of an entire cluster is reliably detected at the other sites. That is, if all sites in a cluster fail then the fact is eventually noticed by the other clusters’ delegates. Otherwise, the cluster is never suspected to have failed.\(^1\) At each cluster, the set of clusters believed to be up is given by a function \( \text{remoteClusters}() \).

2.1. Database Interface

The replication protocol presented in Section 3 uses a replication interface with the database engine that is part of the API being defined in the context of the GORDA project [5]. The interface has been implemented in a number of DBMS, notably in PostgreSQL [9] and Derby [1]. The interested reader can find its detailed definition in [13].

Basically, it allows the inspection of a transaction’s execution at three specific points: (1) at the beginning of the transaction’s execution, (2) at the end of the transaction’s execution, just before it starts committing updates or rolls back, and (3) when the local database system has committed the transaction and is ready to reply to the client. Furthermore, the database engine provides an update function executed with priority over any other running transactions that allows to update the values of a given set of items.

More precisely, we assume that the replicated database engine allows to register four callback functions as follows:

- \( \text{onExecuting}(\text{tid}) \) invoked before a transaction is about to enter the executing state, i.e., before it starts execution.
- \( \text{onCommitting}(\text{tid, rs, ws, wv}) \) invoked when the transaction \( \text{tid} \) succeeds and is about to enter the commit phase. The database provides the set of tuples read (rs) and written (ws) by the transaction, as well as the written values (wv). At this point the transaction has all its updates buffered and all write locks still acquired.
- \( \text{onAborting}(\text{tid}) \) invoked when the transaction \( \text{tid} \) fails and is about to abort.
- \( \text{onCommitted}(\text{tid}) \) invoked after the transaction has completed making all updates persistent, released locks, entered the committed state and is ready to reply to the client.

When it invokes any of the above functions, the database engine suspends the execution of the transaction until the protocol replies by invoking the database functions \( \text{continueExecuting}(\text{tid}) \), \( \text{continueCommitting}(\text{tid}) \), \( \text{continueAborting}(\text{tid}) \) and \( \text{continueCommitted}(\text{tid}) \), respectively.

Replica updates are submitted to the database using the \( \text{dbUpdate}(\text{tid, ws, wv}) \) function which applies the values in wv to the tuples in ws by means of a high priority transaction. A transaction submitted through \( \text{dbUpdate} \) only triggers the \( \text{onCommitted}(\text{tid}) \) event. High priority means that any regular (i.e., non high priority) transaction holding locks on any item in ws will be aborted. Moreover, high priority transactions are serialized when requesting locks and then executed concurrently.

2.2. Communication Primitives

Among sites within the same cluster, a group communication toolkit is available providing reliable point-to-point communication and FIFO uniform view-synchronous multicast [4]. Uniform view-synchronous multicast is defined through primitives \( u_{\text{vs}}\text{cast} \) and \( u_{\text{vs}}\text{deliver} \). FIFO uniform view-synchronous multicast is invoked through primitive \( \text{fifo}_{\text{vs}}\text{cast} \). Point-to-point reliable communication is defined by two primitives \( r_{\text{send}} \) and \( r_{\text{deliver}} \). These primitives rely on the existence of a (primary component) group membership service that tracks the membership of the cluster. Among clusters, messages are exchanged...
The WICE protocol adopts an optimistic concurrency control policy. Transactions are executed optimistically at any site and then, just before commit, certified against concurrent transactions. WICE borrows from protocols such as Postgres-R [10] and DBSM [12] often called certification based protocols. These protocols share two fundamental characteristics: (1) each database site is assumed to store the whole database and transactions can be executed at any site, and (2) all update transactions are certified and, if valid, committed in the same order at all sites.

WICE does not make use of a total order communication primitive, instead ordering is explicitly handled by the protocol. In WICE, one of the sites plays the role of certifier, it totally orders and certifies all transactions. Each valid transaction is then broadcast together with its commit order and updates. This allows to leverage the knowledge about the system’s topology and to make optimizations that would not be possible otherwise.

The WICE algorithm is exemplified in Figure 1. In a nutshell, the handling of a transaction proceeds as follows. Consider a system consisting of two clusters A and B. Each cluster has a designated delegate responsible for handling the communication with the other cluster. The delegate of cluster A, site $s_2$ is also responsible for certifying all executed transactions. When an update transaction $T$ is submitted to site $s_1$ ($T$’s initiator), it is readily executed and sent to the certifier. If it succeeds, then the certifier propagates $T$’s updates and commit order, both locally and to cluster’s B delegate. The latter, in turn, propagates $T$ locally. Once a delegate is certain that all sites in its cluster delivered $T$’s data it acknowledges the fact to the other cluster’s delegate. This acknowledgement is multicast locally by each delegate. Once a database site knows $T$’s data has been delivered everywhere and all previous transactions had been committed or aborted, then it commits $T$. The initiator of $T$ can then reply to its client.

Note that the algorithm discussed here only applies to update transactions, as read-only transactions do not need such a validation. Nevertheless we cannot allow any transaction to read and expose updates before the updating transactions become stable, i.e., committed. For clarity, we omit this from the protocol and assume it to be handled by the local DBMS by blocking the commit of a read-only transaction until all updaters from which it has read from become stable.

### 3.1. Algorithm

We now consider the protocol algorithm in detail (Figure 2). It is composed by a set of handlers that deal with events triggered by the database engine (“Events at the initiator” and “Transaction commit”) and with message delivery. We assume that every database site knows the current system’s certifier through a function certifier(). The local concurrency control strategy of a given site, which we admit to be either snapshot isolation (SI) or strict two-phase locking (S2PL), is given by the function localCC(). Each cluster delegate can find the other participating clusters through a function remoteClusters() as well as identifying some delegate’s cluster through function cluster(). Further, the function delegate() is used to determine whether the current site is the delegate of its cluster or not.

**Global site variables** Each database site manages four sets containing transactions known to be certified, locally updated, locally commited and remotely stable. It keeps track of the number of locally executed transactions in variable lts. The certifier keeps track of the number of certified transactions in variable gts.

**Events at the initiator** Before a transaction tid executes its first operation, the onExecuting handler is invoked. The version of the database seen by tid is required for the validation procedure, and for sites running snapshot isolation, this is equal to the number of committed transactions when tid begins execution. For sites using two-phase locking, the version must instead be recorded at the end of the execution, i.e., in the onCommitting handler.

If the transaction at any time aborts locally, onAborting() is invoked and the transaction is simply forgotten by the protocol. On the contrary, if tid succeeds execution then onCommitting() is invoked. If local consistency is S2PL, the database version is recorded here. Then, tid’s read set, write set and written values (rs, ws and wv) provided by the database are reliably sent to the certifier along with the version of the database on which the transaction executed. The transaction’s execution is left suspended until it is certified and its outcome known. If tid ends up committing then continueCommitting(tid) will be called, otherwise the initiator receives a (ABORT, tid) message from the certifier and forces the transaction to abort locally.
Certification Upon delivering an update transaction to certify — (CERTIFY, tid, ts, rs, ws, wv) — from some initiator site the certifier performs the certification of tid against its concurrent transactions. For every certified transaction (but not necessarily committed yet) ctid with timestamp equal or greater than tid’s, a certification function is called with ctid’s write set and tid’s read and write sets. When preserving 1-SR the certification function checks tid’s read and write sets against ctid’s write set. If 1-SI is the adopted consistency criterion then only the write sets of both transactions are compared. In both cases, if there is a non-empty intersection then the certification fails and an abort message is sent back to tid’s initiator.

When tid’s passes the certification test then the certifier’s sequence number is incremented and tid added to its set of certified transactions. The transaction’s id, commit order, write set and written values are then sent to all other replicas. Locally, tid is sent using the FIFO uniform view-synchronous multicast primitive as a (UPDATE_LOC, tid, gts, ws, wv) message. Remotely, it is sent using the FIFO reliable point-to-point primitive to each remote cluster as a (UPDATE_REM, tid, gts, ws, wv) message.

Remote delivery of updates Once a cluster delegate delivers a transaction from the certifier it simply forwards the message to the local replicas using the FIFO uniform view-synchronous multicast primitive.

Local delivery of updates When a replica delivers a transaction tid it signals the fact adding it to its set of updated transactions. The use of a uniform primitive ensures that once the transaction is delivered at the current replica it is eventually delivered at all non-faulty replicas in the cluster. Therefore, if the replica is a cluster delegate it acknowledges the fact that tid became stable at the cluster to all clusters. The just delivered updates are applied. If the replica is the tid’s initiator then it just needs to proceed with continueCommitting(tid). Although tid does not hold high priority locks at the initiator, the fact that it passed certification means that between its execution and the given commit order, no other certified transaction conflicted with it, and consequently, tid will not be aborted by another transaction requesting high-priority locks at tid’s initiator. For all other sites, db_update is invoked.

Delivery of remote acks Each time a delegate delivers a stability acknowledgment for transaction tid from some cluster, the pair (tid, cluster) is added to its acks set. When tid has been acknowledged by all remote clusters, then the delegate locally declares the transaction remotely stable using the (non-uniform) view-synchronous multicast primitive — (STABLE_REM, tid). When this message is delivered each replica adds tid to its remotestable set.

Transaction commit Here, each site handles the onCommitted callback. When onCommitted (tid) is invoked the site just increments its local database version lts and adds tid to its committed set. Since all tid locks have been released then any new transaction can read from tid and therefore from a more recent version of the database. When tid is known to be committed locally and stable everywhere the database is then allowed to reply to the client, which happens after continueCommitted(tid).

3.2. Failure Handling

The WICE algorithm tolerates both the failure of single database sites as well as the failure of whole clusters. In this section we present and explain the recovery procedures in both cases.

Locally, each cluster is governed by a group membership service and local communication rests on view-synchronous multicast primitives. This definitely eases failure handling locally. In the event of a site being expelled from the group (because it was taken down, has failed, became unreachable, etc.) every other site in the group eventually becomes aware of the fact by installing a new view of the group. This allows each site to deterministically determine the cluster’s delegate should the former failed. Moreover, view-synchrony ensures that all sites surviving the previous view delivered the same set of messages, thus not requiring any synchronization among them. As a result, no particular procedure is required on the failure or an ordinary site. In the next two sections we examine the failures of a cluster’s delegate and of the system’s certifier. Then, we consider the failure of an entire cluster. For the sake of simplicity and lack of space, we assume that no sites are added to a cluster and that once a site is expelled from the group, whatever was the reason for this, it is no longer readmitted.

3.2.1 Delegate Failover

In Figure 3a, we sketch a protocol for recovering from a site failure when this site was the cluster’s delegate. On a view change, site d becomes aware it is the new cluster’s delegate. To ensure that no transactions are blocked, d must re-run all transaction updates and acknowledgements received from remote clusters that may have been incompletely processed by the previous delegate.

New delegate: Synchronization request When initialized, the new delegate d sends a message (DELEGATE.Sync, lts) to the certifier in order to ensure that all transactions certified since lts are delivered in its local cluster. The lts value corresponds to the latest transactions updated in d’s cluster. The new delegate also contacts each remote cluster with (ACK.Sync, lts, TRUE) acknowledging the local stability of all transactions up to lts, requesting similar action from the recipients (argument TRUE of the message).
Global site variables
1  local = ts = []
2  certified = updated = ()
3  commited = remotestable = acks = {}
4  gts = lts = ...

Events at the initiator
5  upon onSending(tid)
6    if localCC() == S2PL
7      then if local[tid] = lts
8        then continueSending(tid)
9      end
10   upon onSending(tid)
11    if certified[tid, id] = lts
12      then continueSending(tid)
13    end
14   end

(1) Certification
15  upon delivered(CERTIFY, tid, ts, ws, wv)
16    from initiator
17    for each (ctid, cts, cws, cwv) in certified
18      if cts ≥ ts and certification(cws, ws, wv)
19        then continueCommitting(tid)
20      else continueAborting(tid)
21    end
22    gts = gts + 1
23    add (tid, cluster) to acks
24    for each cluster in remoteClusters()
25      enqueue (tid, gts, ws, wv) to cluster
26    end
27  end

(2) Remote delivery of updates
28  upon delivered(UPDATE, tid, ts, ws, wv)
29    from certifier
30    for each (ctid, cts, cws, cwv) in certified
31      if (ctid, cts, cws, cwv) in committed
32        then add c to acked
33      else continueCommitting(tid)
34      end
35    end
36    lts = lts + 1
37    remoteClusters()
38      for each cluster in remoteClusters()
39        if ts > lts
40          then continueCommitting(tid)
41        end
42      do
43          if certified
44            then remoteClusters()
45          end
46          then if remoteCluster()
47            then continueCommitting(tid)
48            do
49              if remoteCluster()
50                then continueCommitting(tid)
51                else continueCommitting(tid)
52                end
53            end
54            do
55              if remoteCluster()
56                then continueCommitting(tid)
57                else continueCommitting(tid)
58                end
59            do
60            end
61            do
62            do
63            end

(4 and 5) Delivery of remote acks
64  upon delivered(ACK_REM, tid from cluster
65    do
66      if aclst from aclst
67        then aclst from aclst
68      else aclst from aclst
69      end
70      aclst from aclst
71      aclst from aclst
72      aclst from aclst
73      aclst from aclst
74      aclst from aclst
75      aclst from aclst
76      aclst from aclst
77      aclst from aclst
78      aclst from aclst
79      aclst from aclst
80      aclst from aclst
81      aclst from aclst
82      aclst from aclst
83      aclst from aclst
84      aclst from aclst
85      aclst from aclst
86      aclst from aclst
87      aclst from aclst
88      aclst from aclst
89      aclst from aclst
90      aclst from aclst
91      aclst from aclst
92      aclst from aclst
93      aclst from aclst
94      aclst from aclst
95      aclst from aclst
96      aclst from aclst
97      aclst from aclst
98      aclst from aclst
99      aclst from aclst
100     aclst from aclst
101     aclst from aclst
102     aclst from aclst
103     aclst from aclst
104     aclst from aclst
105     aclst from aclst
106     aclst from aclst
107     aclst from aclst
108     aclst from aclst
109     aclst from aclst
110    do
111    end
112    end
113    end
114    end
115    end
116    end
117    end
118    end
119    end
120    end

Figure 2: WICE protocol

Certifier: Handle synchronization request When delivering this message, the certifier resends (in order) each certified transaction with a certification timestamp larger than d’s lts value.

All delegates: Synchronize ACK’s When the message (ACK, clts, tid, reply) from a cluster is delivered to a remote cluster C, the delegate of C regards all its updated transactions with ts <= clts as acknowledged by cluster. It then just checks whether these transactions become stable in every cluster and proceeds accordingly. If reply was set to TRUE a similar message (now with reply set to FALSE) is sent back to the initializing delegate (just elected) so it can also update the respective acknowledgements.

3.2.2 Certifier Failover
The most serious single server failure is when the current system’s certifier becomes unavailable. When initialized, the new certifier advertises itself to all delegates. There may be previously certified transactions not yet known to new certifier so a state synchronization is due. Figure 3b shows our synchronizaon protocol in pseudocode. The code assumes two existing functions, blockCertification() and unblockCertification(). Their implementation is not shown, but they state all arriving certification requests should be buffered, awaiting the synchronization protocol to finish.

New certifier: Synchronization request The new certifier c starts by invoking blockCertification() and requesting from each cluster all the transactions they might have delivered and updated after the last one updated by c.

Each delegate: Send missing transactions When a (CERTSYNC_REQUEST, clts) is received by the delegate of a cluster C, it replies with a list of its updated transactions (tid, ts, ws, wv) such that ts > clts, that is, transactions not yet seen by the new certifier.

Certifier: Missing updates When processing a (CERTSYNC_REPLY, clts, missing) from remote cluster C, the new certifier c then checks each member of the missing list whether it has already received this transaction from another cluster. This will happen if two or more remote clusters both know about a transaction which is unkown to c. If not, the transaction is enqueued in c’s certified queue. As soon as all replies from remoteCluster() are delivered, c sets the certifiers counter gts to lts and starts distributing from its certified queue (1) locally transactions with ts <= lts and (2) remotely according to each cluster’s last updated transaction. The certifier’s gts counter is updated for each transaction distributed locally. Finished the update, certification is unblocked.

3.2.3 Multiple Failures
The WICE protocol shall tolerate situations where multiple servers or entire clusters can fail abruptly. Most failure scenarios can be handled using a combination of the procedure for single servers. To avoid blocking during synchronization, we assume that all running synchronization routines are restarted if a delegate fails.

The only scenario which requires special treatment is the loss of an entire cluster. In that case, the other clusters must be informed as soon as possible to allow blocking current and future transactions to become stable. A handler for this event is illustrated in Figure 3c.

4. Evaluation
In replication protocols that rely on a system-wide uniform atomic broadcast, updates cannot be applied before
their carrier message has been delivered (and acknowledged) by all sites. This means that a full round-trip to the most distant site $2 \cdot t_{max}$ is required before updates can be installed, regardless of the location of the initiator. As the probability of two concurrent transactions conflicting depends on the latency, this has a profound impact in the abort rate of certification based protocols such as DBSM and Postgres-R [6].

In WICE, and considering two clusters $C_A$ and $C_B$, total ordering of messages is performed using a sequencer sited, say, in cluster $C_A$, also referred to as the primary cluster. The updates of each update transaction can be installed as soon as the certification result is known but they are made visible to clients only after stabilization. Thus, it makes sense to distinguish between $\text{interval}$ and $\text{commit-interval}$.

Commit-interval denotes the time elapsed from the end of execution until the transaction gets committed at the originating site and is still lower bounded by $2 \cdot t_{max}$. The install-interval is the time elapsed from the moment the transaction finishes its optimistic execution until some site installs the incoming updates. Ignoring intra-cluster latency, and considering transactions originated at $C_A$, the install-interval is negligible for servers in cluster $C_A$ and close to $t_{max}$ in cluster $C_B$. On the other hand, for transactions originating in cluster $C_B$, the install-interval will be close $t_{max}$ and $2 \cdot t_{max}$ for $C_A$ and $C_B$ respectively.

The most significant advantage of the WICE protocol when compared to DBSM in a wide area network should therefore be its impact on the abort rate due to early delivery of updates. In this section, we experimentally verify this claim.

4.1. Experimental Environment

Experimental evaluation is conducted by running an actual implementation of the protocol within a simulated environment. By profiling real components with CPU cycle counters, the technique captures the actual overhead introduced by protocols [15]. By fine tuning the simulation components to accurately reproduce real components, it realistically reproduces results of real distributed systems [17]. When compared to testing in a real setting, this allows a tight control over experimental conditions, with advantages in repeatability and observability. The approach has been previously used to evaluate database replication protocols both in LANs and WANs [6]. In detail, we use simulated database clients, database engines and networks, and real implementations of replication and group communication protocols.

The workload generator is configured according to the industry standard on-line transaction processing benchmark TPC-C [19]. Briefly, a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. This workload is update intensive, as 92% of the transactions perform updates. It is also varied, as the $\text{delivery}$ transaction takes a considerable amount of CPU time and has a very large read-set. The $\text{payment}$ transaction is likely to produce Write-Write conflicts. The $\text{neworder}$ transaction is short-lived and with higher locality.

The results thus vary according to the platform used for calibration of the simulated environment [17]. Results presented in this paper refer to the following hardware configuration: Each server has a single CPU AMD Opteron 250 running at 2.4GHz, 4GB RAM and a RAID 5 SATA disk array with fibre attachment. Transaction processing engines and overheads are configured according to Post-
Transaction Name Distribution Empirical Estimators
Delivery normal mean=143.70 sd=2.33
Neworder uniform min=6.45 max=16.83
don't include contention, as when blocked waiting for a resource processes are not scheduled. Also according to PostgreSQL 8.0, transaction processing engines use a multi-version concurrency control approach.

In our target scenario, 3 database servers are positioned at each of two different sites, as shown in Figure 4. The network simulator is configured as a pair of switched 1Gbps Ethernet local area networks, connected by a dedicated T3 link (45Mbps) with 400ms round-trip latency, representative of an inter-continental satellite link. As a baseline, we present also results obtained when configuring all 6 servers within the same local area network.

In all scenarios, we vary the number of simulated clients from 60 to 6000, equally spread by all servers. We also take advantage of the locality in TPC-C: Clients associated with the same warehouse are connected to the same server to exploit locality, as suggested by the TPC-C specification. Note however, that with a small probability any client updates records associated with any warehouse.

4.2. Performance Results
The performance of the WICE protocol is evaluated by observing the throughput, latency and abort rate achieved when compared with plain DBSM. As a baseline, we present results obtained by grouping all 6 servers in the same cluster (DBSM CLUSTER). The results, obtained with Write-Write conflict certification (achieving 1-SI), are presented in Figure 5. Results are presented separately for each cluster.

The first interesting observation from the baseline protocol (DBSM CLUSTER) is that the capacity of the system is exhausted with 6000 clients. This shows up as throughput peaking (Figure 5(a)), increasing latency due to queuing (Figure 5(c)), and abort rate due to increased concurrency (Figure 5(e)). By examining resource usage logs one concludes that this is due to saturation of available CPU time. We should thus focus on system behavior up to 4000 clients, as a properly configured system will perform flow control to ensure operation in that range. Throughput grows linearly, latency is approximately constant and the abort rate negligible.

Then, we turn our attention to DBSM in the target scenario. Although throughput scalability is apparently close to linear, it is misleading as it corresponds to a high abort rate and a linearly increasing latency, in particular in cluster $C_B$ (Figures 5(d) and 5(f)). Both are explained by the same phenomenon: As locks are withheld during wide area stabilization, queuing delays arise, thus proportionally increasing the probability of later being aborted. Aborted transactions have to be resubmitted by the application, thus further loading the system. It is also important to underline that, as expected, latency and abort rate impact both clusters equally as both suffer with the same $2 \cdot t_{\text{max}}$ commit-interval.

As expected, the WICE protocol improves the performance at the primary cluster without negatively impacting secondary clusters. Namely, in the primary cluster the abort rate is negligible (Figure 5(e)), comparable only with the DBSM CLUSTER scenario. The latency is also approximately constant in the safe operating range (i.e., up to 4000 clients), although impacted by the round-trip over the wide area link (Figure 5(c)). Note however that such impact is very close to the absolute minimum of $2 \cdot t_{\text{max}}$ at 400 ms.

Also as expected, the abort rate of transactions initiated in the second cluster, which are impacted by a $t_{\text{max}}$ to $2 \cdot t_{\text{max}}$ commit-interval, is not negligible although still
offering a substantial improvement on DBSM. In the next section, we discuss the impact of this in the expected usage scenario of WICE.

4.3. Discussion

The workload assignment used in the previous section deserves some additional comments. The WICE protocol targets the global enterprise where the goal of replication is twofold. First, by providing a cluster for each region of the globe one avoids having to route all queries to a central location and thus avoid imposing the large latency on clients when no updates are performed, while at the same time balancing the load. Second, it improves availability as even catastrophic disasters can only impact the computing or communication infrastructure at a single location. One has therefore to consider clusters located in different time-zones, having distinct peak utilization periods.

This means that the evaluation scenario in the previous section, in which traffic in both clusters is exactly the same, is the worst case scenario for the proposed protocol. In reality, one should be able to migrate the centralized sequencer to the currently most loaded cluster. The additional abort rate at other locations can then be easily solved by resubmission, as these clusters are off peak and thus with under-utilized resources.

We also have not assumed that resubmission can be done automatically by the database management system. However, this is true for many workloads, especially in current multi-tiered applications. By taking advantage of such option one could thus completely mask the abort rate at secondary clusters.

5. Conclusion

Eager update-everywhere database replication optimized for interconnected clusters in wide area networks is a valuable contribution to the infrastructure of the global enterprise. By providing the ability to locally serve clients it improves performance and by allowing failover ensures disaster recovery with no data loss. This is a hard problem, which existing commercial solutions address either by admitting some data loss or by centralizing update processing.

The proposed WICE protocol shows how to scale replication protocols based on group communication to a wide area setting with increased performance, while at the same time increasing their practicality. This is achieved by restricting group communication within clusters and using a simple peer protocol over long distance links. The evaluation performed in a realistic platform illustrates the advantages of the approach, namely, linear throughput scalability, up to 2 times less latency and a negligible abort rate at the cluster supporting the region currently generating the most traffic.

References

Paper 2: Scalable and Fully Consistent Transactions in the Cloud through Hierarchical Validation
Scalable and Fully Consistent Transactions in the Cloud through Hierarchical Validation

Jon Grov\textsuperscript{1,2} and Peter Csaba Ölveczky\textsuperscript{1,3}
\textsuperscript{1} University of Oslo
\textsuperscript{2} Bekk Consulting AS
\textsuperscript{3} University of Illinois at Urbana-Champaign

Abstract. Cloud-based systems are expected to provide both high availability and low latency regardless of location. For data management, this requires replication. However, transaction management on replicated data poses a number of challenges. One of the most important is isolation: Coordinating simultaneous transactions in a local system is relatively straightforward, but for databases distributed across multiple geographical sites, this requires costly message exchange. Due to the resulting performance impact, available solutions for scalable data management in the cloud work either by reducing consistency standards (e.g., to eventual consistency), or by partitioning the data set and providing consistent execution only within each partition. In both cases, application development is more costly and error-prone, and for critical applications where consistency is crucial, e.g., stock trading, it may seriously limit the possibility to adopt a cloud infrastructure. In this paper, we propose a new method for coordinating transactions on replicated data. We target cloud systems with distribution across a wide-area network. Our approach is based on partitioning data to allow efficient local coordination while providing full consistency through a hierarchical validation procedure across partitions. We also present results from an experimental evaluation using Real-Time Maude simulations.

1 Introduction

Cloud-based systems are expected to provide good performance combined with high availability and ubiquitous access, regardless of physical location and system load. Data management services in the cloud also need database features such as transactions, which allow users to execute groups of operations atomically and consistently. For many applications, including payroll management, banking, resource booking (e.g., tickets), shared calendars, and stock trading, a database providing consistency through transactions is crucial to enable cloud adoption.

To achieve high availability and ubiquitous access, cloud-based databases require data replication. Replication improves availability, since data are accessible even if a server fails, and ubiquitous access, since copies of data can be placed

\textsuperscript{*} This work was partially supported by AFOSR Grant FA8750-11-2-0084.
near the users. Replication may also increase scalability as the workload can be distributed among multiple hosts. Unfortunately, transaction management on replicated data is hard. Managing concurrent access on replicated data requires coordination, and if copies are separated by slow network links, this may increase transaction latency beyond acceptable bounds.

These challenges have made most cloud-based databases relax consistency. Several applications use data stores, which abandon transaction support to reduce latency and increase availability. Notable examples of such data stores are Amazon’s Dynamo [1], Cassandra [2], and Google BigTable [3]. A recent trend is data stores with transactional capabilities within partitions of the data set. Examples include ElaStraS [4], Spinnaker [5] and Google’s Megastore [6]. All of these provide high availability, but the transaction support is limited as there is no isolation among transactions accessing different partitions. This imposes strict limits on how to partition the data, and reduce the general applicability.

Managing consistency in applications without transaction support is difficult and expensive [7]. Furthermore, inconsistencies related to concurrent transactions can potentially go undetected for a long time. Google’s Spanner [8] combines full consistency with scalability, availability, and low latency in a system replicated across a large geographical area (both sides of the US). However, Spanner is deployed on a complex infrastructure based on GPS and atomic clocks, which limits its applicability as a general-purpose solution.

In this paper, we propose a method for managing replicated data which provides low latency, transaction support, and scalability, without requiring specific infrastructure. Our approach, FLACS (Flexible, Location-Aware Consistency), is based on the observation that in cloud systems, transactions accessing the same data often originate in the same area. In a world wide online bookstore, the chance is high that most transactions from Spain access Spanish books, while German customers buy German book. For this, partitioning the database according to language would work with traditional methods. However, since we also need to support customers purchasing books both in Spanish and in German, a more sophisticated solution is needed.

FLACS provides full consistency across partitions by organizing the sites in a tree structure, and allow transactions to be validated and committed as near their originating site as possible. To facilitate this, we propose an incremental ordering protocol which allows validation without full view of concurrent transactions. For many usage patterns, this allows the majority of transactions to execute with minimal delay.

We have formally specified the FLACS protocol as a real-time rewrite theory [9], and have used Real-Time Maude [9] simulations to compare the performance of FLACS to a classical approach with a master-site for validation.

The rest of the paper is structured as follows. Section 2 defines our system model. Section 3 gives an overview of the FLACS protocol. Section 4 explains the protocol in more detail. Section 5 presents our simulation results. Finally, Section 6 discusses related work and Section 7 gives some concluding remarks.
2 System Model

We formalize a system for storing and accessing replicated data as a tuple 
\((P, U, I, O, T, Q, D, lb)\) where:

\- \(P\) is a finite set (of process identifiers representing a set of real-world processes, typically a set of network hosts).
\- \(U\) is a set (representing possible data values).
\- \(I\) is a set (of identifiers for logical data items).
\- \(O \subseteq (\{\text{read}\} \times I) \cup (\{\text{write}\} \times I \times U)\) is a set of possible operations on items.
\- \(T\) is a set (of transaction identifiers).
\- \(Q\) is a set of transactions of the form \((t, p, O_{t,p}, <_{t,p})\), where \(t \in T\) is the transaction identifier, \(p \in P\) is the process hosting transaction, \(O_{t,p} \subseteq O\) is the set of operations executed by \(t\) on \(p\) and \(<_{t,p}\) is a partial order on \(O_{t,p}\).
\- \(D \subseteq I \times U \times P\) is a set (with \((i, u, p)\) a replica of \(i\) with value \(u\) at \(p\)).
\- \(lb\) is a function \(lb : P \times P \rightarrow \mathbb{N}\) (denoting the lower bound on the message transmission time from \(p\) to \(p'\)).

The read set of a transaction \((t, p, O_{t,p}, <_{t,p})\) is the set \(RS(t) = \{i \in I \mid \{\text{read}, i\} \in O_{t,p}\}\), and the write set of \(t\) is \(WS(t) = \{i \in I \mid \{\text{write}, i\} \in O_{t,p}\}\). A pair of transactions \(t, t'\) are in conflict if \(WS(t) \cap (RS(t') \cup WS(t')) \neq \emptyset\), or vice versa.

A read-only transaction is a transaction \(t\) where \(WS(t) = \emptyset\). Managing read-only transactions is relatively easy. Therefore, by the term transaction we will mean a transaction \(t\) with \(WS(t) \neq \emptyset\) unless stated otherwise. The treatment of read-only transactions is discussed in Section 3.4.

We assume that processes communicate by message passing, and that each pair \((p, p')\) of processes is connected by a link with minimum message transmission time \(lb(p, p')\). We also assume that the underlying infrastructure provides the following operations for inter-process communication:

\- unicast\((m, p, p')\), where \(m\) is some message, \(p\) is the sender and \(p'\) is the receiver. Unicast does not guarantee any upper bound on message delivery times nor that messages are delivered in the order in which they were sent.
\- fifoUnicast\((m, p, p')\). Similar to unicast, but guarantees that messages between two processes are delivered in the order in which they were sent.

We use simple utility functions for multicast and broadcast built on unicast, and do not assume access to sophisticated group communication middleware.

3 Overview of the FLACS Protocol

State-of-the-art database replication protocols, such as Postgres-R [10] or DBSM [11], provide serializability through optimistic validation combined with atomic broadcast to order all transactions before commit. FLACS is an optimistic protocol following a similar approach with one notable exception: FLACS does not require a total order on all transactions before validation. Instead, a transaction \(t\) is executed as follows:
1. Execute all operations at the process receiving $t$ (denoted $t$’s initiator).

2. Ordering: A set of processes denoted observers are asked to order $t$ against all conflicting transactions. The observers for $t$ are given by $RS(t)$ and $WS(t)$.

3. Validation: Once $t$ is ordered against all conflicting transactions, it is ready for validation. The validating process $p$ is determined by the observers. $t$ is granted commit if and only if for each member $i$ of $RS(t)$, $t$ has read the most recent version of $i$ according to the local order of $p$.

4. If $t$ is committed, updates are applied according to the order seen by the validator. Otherwise, an abort-message is sent to participating processes.

The purpose of FLACS is to reduce validation delay since coordination among the observers usually requires fewer messages than an atomic broadcast.

### 3.1 Observers

An observer’s task is to serialize updates on its observed items. Formally, an observer function $obs: I \rightarrow P^+(P)$ maps each item $i$ to its observer(s) $obs(i)$. The idea is to choose as observers processes physically near the most frequent users, and assign items commonly accessed by conflicting transactions to the same observer(s). The observers for a transaction $t$ is the union of the observers for all items in $WS(t)$.

**Example 1.** Consider a hotel reservation service. Since most reservations are local, *rooms in France* should map to observers physically located in Paris, while *rooms in Germany* are observed by processes in Berlin. As explained below, this allows transactions accessing rooms only in France to commit locally in Paris.

### 3.2 Ordering

The FLACS validation procedure dictates that a transaction $t$ is granted commit if and only if $t$ has read the most recent version of each $i \in RS(t)$. Since there is no common time among processes, we need to define “most recent.” For protocols where transactions are included in a total order before validation, the definition of most recent is simple: it is the most recent according to the total order.

FLACS does not include transactions in a total order before validation. Instead, FLACS uses an incremental ordering and validates a transaction $t$ as soon as it is ordered against all conflicting transactions. Each process $p$ maintains a local, strict partial order $\prec_p$ on the (update) transactions seen so far. Intuitively, $\prec_p$ must order any pair of transactions $t, t'$ known by $p$ to be in conflict. However, the local orders at different processes might be inconsistent. Our idea is to combine these local orders using a tree structure among processes, in which the root of a subtree is responsible for combining the local orders of its descendants, or discovering inconsistencies and resolving them by aborting transactions.

A transaction $t$ can be validated if all observers of items in $WS(t)$ have treated $t$, and if the local orders of these observers are consistent up to $t$; i.e., they can be combined into one strict partial order.
The first step of validating a transaction $t$ is to ensure that $t$ is included in the local order of every observer for each item in $WS(t)$. The next step is to merge the local observer orders and check if they are consistent. As explained above, we achieve this by organizing processes in a tree structure, called the validation hierarchy. After a transaction is ordered at the observer level, the proposed ordering is propagated upwards in the hierarchy. Eventually, each transaction is included in a total order at the root of the hierarchy; however, the validation (and commit) of a transaction $t$ may take place before $t$ is included in this total order, as explained below.

Example 2. Consider the validation hierarchy in Fig. 1. Process $p_e$ represents the European headquarters of our travel agent. Processes $p_g$ and $p_f$ are observers for German and French hotel rooms, respectively. Let $t_1$ and $t_2$ be two transactions, reserving one room in Berlin and one room in Paris, respectively, and let $t_3$ reserve a room in both cities. The orderings then develop as follows:

- $p_g$ orders $t_1$ and $t_3$, and all other transactions updating German rooms. The resulting local ordering $\prec_{p_g}$ is then propagated to $p_e$.
- $p_f$ orders $t_2$ and $t_3$, and all other transactions updating French hotel rooms. The resulting local ordering $\prec_{p_f}$ is then propagated to $p_e$.
- Finally, the local ordering $\prec_{p_e}$ combines $\prec_{p_g}$ and $\prec_{p_f}$.

![Fig. 1. Example validation hierarchy](image)

Transactions only accessing German rooms can therefore be validated by $p_g$ alone. A transaction accessing both German and French rooms is validated by $p_e$, which combines the orderings of $p_g$ and $p_f$.

3.3 Validation

We next explain in more detail how a transaction $t$ is validated in FLACS. The validating process $p$ for $t$, called $t$’s validator, is given as follows:

1. For each item $i$ in $WS(t)$, all observers $obs(i)$ of $i$ are contained in the subtree rooted at $p$ in the validation hierarchy.
2. At least one observer of each item in $RS(t)$ is contained in the subtree rooted at $p$ in the validation hierarchy.
3. No descendant of $p$ in the validation hierarchy satisfies properties 1 and 2.

To validate $t$, $t$’s initiator sends a validation request to $t$’s validator $p$ containing $RS(t)$, $WS(t)$, $Wval(t)$ (values written by $t$), and $Rver(t)$ (item versions read by $t$; each version is represented by the id of the updating transaction). Transaction $t$ is ready for validation once this message is received and $t$ is included in $\prec_p$.
is granted commit if and only if, for each member $i$ of $RS(t)$, $Rver(t)$ contains the most recent version of $i$ according to $\prec_p$.

The correctness argument is the following: To perform this test at the validating process $p$ is equivalent to performing it at the root of the validation hierarchy, where the ordering is global. Since all observers for $t$ are contained within the subtree rooted at $p$, $t$’s ordering at $p$ is consistent. Additionally, due to the ordering being propagated upwards in the validation hierarchy, we know that any preceding transaction in conflict with $t$ will be known at $p$ upon $t$’s validation. Therefore, the validation test for $t$ at $p$ is equivalent to testing at the root of the validation hierarchy and FLACS guarantees serializability (and consequently, strong consistency).

If $t$ fails the validation test, a message $abort(t)$ is broadcast. Otherwise, a commit message for $t$ is sent to all processes replicating items updated by $t$. This may include processes that are neither the initiator, observers or part of the validation hierarchy for $t$. Since transactions updating the same items may be validated by different processes, commit messages can arrive out of order. To handle this, we introduce sequence numbers. For an item $i$, the lowest process $p$ where all $q \in obs(i)$ are in the subtree rooted at $p$, is responsible for the sequence number of $i$. Whenever $p$ orders a transaction $t$ updating $i$, the sequence number of $i$ is incremented and propagated upwards in the validation hierarchy together with the proposed ordering for $t$. Consequently, $t$’s validator will have a complete set of sequence numbers for items in $WS(t)$. We denote this set $Wseq(t)$.

Upon receiving a commit message $commit(t, WS(t), Wval(t), Wseq(t))$, each process $p$ replicating items in $WS(t)$ initiates a local transaction containing $t$’s write operations. For each item $i$, the sequence number of the most recent version is stored at $p$. We refer to this value as $curseq(i, p)$. We then apply Thomas’ Write Rule: Let $seq_i(t)$ represent the sequence number of $i$ created by $t$. For a replicated item $i$ at process $p$, we apply $t$’s write operation at $p$ if and only if $curseq(i, p) < seq_i(t)$.

### 3.4 Fault Tolerance and Read-only Transactions

For fault tolerance, our ordering protocol represents the first phase of a two-phase commit. If we assign more than one observer to an item, and then require the validator to synchronize with observers before commit, this item will be accessible as long as a majority of observers are available. In future work, we will combine FLACS with Paxos to provide more sophisticated fault tolerance.

To ensure a consistent read set, a read-only transaction $t_r$ must be executed at, or validated by, a process $p_u$ where, for every item $i$ in $RS(t_r)$, there is at least one observer for $i$ in the subtree rooted by $p_u$. Read-only transactions requiring “fresh” data follow the same validation procedure as update transactions.

### 4 The FLACS Protocol

This section presents the FLACS protocol in more detail. The complete formal, executable Real-Time Maude specification of FLACS is available at http://
In this paper, we describe the protocol using pseudocode as a set of rules. The following message types are involved in completing the execution of a transaction $t$:

- **informObserver**: Sent from $t$’s initiator to $t$’s observers to initiate $t$’s ordering.
- **propagateOrder**: Propagate the order upwards in the validation hierarchy.
- **validateRequest**: Sent from $t$’s initiator to its validator (see Section 3.3).
- **commit**: Sent from $t$’s validator to all processes to signal commit.
- **abort**: Sent from a process which determines that $t$ must abort.

The following variables represent the local state of each process $p$:

- **DATABASE**: A set of records $(i, value, seqnum, update-history, lock-reqs)$ representing $p$’s version of the database, where $value$ is the local value of item $i$; $seqnum$ is the sequence number of the most recent update of $i$; $update-history$ is a list containing the transaction name of previous updaters of $i$; and $lock-reqs$ is a list of requests for either read lock or write lock on $i$.
- **LOCAL-ORDER**: A list of transaction ids representing the local order at $p$.
- **REMOTE-TRANS**: The set of currently executing remote transactions.
- **ORDER-GRAPH**: A graph of transactions awaiting to be ordered at $p$.
- **VALIDATE-REQ**: A list representing received validation requests.

**RULE**: `EXECUTE-TRANS(t)`

*while* $t$ has more operations *do*

*when* lock granted *do* executeOperation($t$, optype)  

*when* aborted by high priority transaction *do* abortTransaction()  

$ops_t = getExecutedOperations(t)$  

$RS(t) = getReadSet(ops_t)$  

$RVer(t) = findReadVersions(ops_t, DATABASE)$  

$WS(t) = getWriteSet(ops_t)$  

$writeobs = getWriteObservers(WS(t))$  

multicast informObserver($t$, $WS(t)$) *from* $p$ to $writeobs$  

unicast validateRequest($t$, $RS(t)$, $WS(t)$, $WVal(t)$, $RVer(t)$) *from* $p$ to $validator$  

*await* commit decision  

*if* commit granted *then* report success to client  

*else* report abort to client  

releaseLocks($t$, DATABASE)

**Fig. 2.** Initial execution at initiator

### 4.1 Initial Execution

The execution of a transaction $t$ at $t$’s initiator is described in Fig. 2. The operations in $t$ are executed sequentially, and we assume local concurrency control using locks.

When all operations of $t$ have been executed, $t$ is submitted for ordering and validation. The list of executed operations is logged, and the read set and write set (including written values) can be retrieved. The initiator determines the observers for $t$ and initiates the ordering protocol by multicasting `informObservers` to those observers. This message contains the write set of $t$ which is used to
acquire locks for the relevant items. Furthermore, the validating process is notified by the message `validateRequest`, which also contains `WVal(t)`, the updated values, and the mapping `RVer(t)`, associating every item `i` in `RS(t)` to the id of the transaction performing the most recent update on `i` prior to `t`’s read. Note that the initiator may be an observer, and often also the validator.

After the ordering and validation messages are sent, the initiator waits for the commit decision, and replies to the client accordingly. Finally, locks are released.

### 4.2 Ordering

Figure 3 describes the rules for ordering transactions. Whenever an observer receives an `informObserver` message for transaction `t` (rule `INIT-ORDER`), it creates a remote subtransaction to apply `t`’s updates (unless this observer is the initiator). Remote subtransactions are write-only, request high priority locks to abort any local transaction (these would eventually fail validation anyway), and await the commit decision of `t` before committing. A node for `t` is also added to the order graph.

The rule `ORDERED` is executed at process `p` when a transaction `t` satisfies the requirements to be ordered at `p`; i.e., all expected order requests have been received and there are no preceding transactions in the local ordering graph. Then, `t` is appended to the local order at `p` and a `propagateOrder`-message is sent to `p`’s parent in the validation hierarchy. Since we use FIFO-unicast, the ordering of `propagateOrder`-messages from process `p_a` to `p_b` reflects the local order at `p_a`.

The rule `RCV-ORDER` is executed whenever a process `p` receives a `propagateOrder` message for `t` from a child `p'` in the validation hierarchy. Unless `t` is already known at `p`, the process `p` first initiates a remote subtransaction to acquire the necessary write locks. In any case, an edge from `t_{prev}` to `t` is added to the local order graph of `p`, where `t_{prev}` is the most recent transaction received from `p'` before `t`. If the ordering becomes inconsistent, there will be a cycle in the local order graph and the transaction is aborted. This rule will be triggered repeatedly for `t` until all expected `propagateOrder` messages have arrived. Eventually, `t` will either be aborted or satisfy the conditions for the rule `ORDERED` at `p`; the proposed ordering is then propagated to `p`’s parent.

### 4.3 Validation

The rule for validation is given in Fig. 4. For each transaction `t`, validation is performed by the receiver of the `validateRequest`. Validation of `t` occurs as soon as `t` has been ordered at `p_v` and `p_v` has received the `validateRequest` message for `t`. The validation test is a standard optimistic validation procedure, using the local update history at `p_v` to verify that for each item `i` read by `t`, `t` saw the most recent version of `i` according to `p_v`’s local order.
5 Performance Evaluation

We have implemented a simulation model using the Real-Time Maude tool, and compared FLACS to a "classical" approach where one master process acts as the central validator. The latter approach was previously shown to outperform protocols that use atomic broadcast in wide-area networks [12]. Since recent research focuses on atomic broadcast-based replica control or weaker consistency models, this comparison is relevant to evaluate the performance of FLACS.

5.1 Experiment Setup

Our experiment setup is an imaginary international travel agent, providing hotel bookions in Paris, New York, London, and Los Angeles. Each city is served by one process, and each process maintains a complete copy of the database. Scenario A is a setting with a master validating all transactions. Scenario B is our FLACS model, where we assign as observer for an item i the process most likely to access i. We assume a validation hierarchy and network setup as shown in Fig. 5. We model a network with stochastic delay with average values
RULE: VALIDATION
when hasLocalOrder(t) and hasReceivedValidationRequest(t) do
  if isValid(RS(t), RVer(t), WS(t)) then broadcast commit(t, WS(t), WVal(t), Wseq(t))
  else broadcast abort(t)

func isValid(RS(t), RVer(t), WS(t))
  foreach i in RS(t) do
    version = getVersion(i, RVer(i))
    if version < getLatestVersion(i, DATABASE) then return false
  return true

Fig. 4. Validation

Fig. 5. Validation hierarchy with observer placement, and average network delay (ms).

chosen according to the geographical distance. We inject transactions with a load generator per process, which generates transaction requests at random times with an adjustable average rate, measured in transactions per second (TPS). All processes have the same average. Once a lock is acquired, we assume a delay of 2 ms per local operation. We do not model protocol overhead since network latency is the dominating factor. In these experiments, no failures are injected.

Each item represents one hotel room at some date. We assume a “hotspot” setting (e.g., a sale period) with only 10 items at each process. We have different transaction types, and each transaction type will access either one room in one city or two rooms (total) in two cities. Each load generator randomly selects a transaction type according to the distribution given in Table 1. The rooms accessed are chosen randomly, and Book London represents a read and consecutive write of one room in London. Correspondingly, Book London+Paris is the read and consecutive write of one room in London and then one in Paris. We performed four experiments, varying the overall target throughput between 20 and 60 transactions per second. We measured the abort rate and transaction latency, i.e., the time between a request is submitted and it is successfully returned.

5.2 Results

The abort rate and average transaction latency for Scenarios A and B are shown in Fig. 6. Decentralized validation allows FLACS to commit a significantly higher

---

4 The delays New York–Paris and New York–London are the same, assuming transatlantic backbone links from each of these cities. The delay between Paris and London reflect that network equipment and local lines increase delivery times.
number of transactions, and the observed transaction latency, affecting both abort rate and user experience, is significantly lower where observers are distributed. This is as expected: In Scenario A, all processes except New York have an average of 84 ms added latency before commit. This increases the delay from an update is initiated until it is applied to other replicas, and consequently, there is a higher probability for transactions elsewhere to read stale data. In Scenario B, the abort rate for transactions accessing items from multiple locations is relatively high. Especially the transaction *Book London+Paris* initiated in Los Angeles suffers, with an abort rate close to 54% at 60 TPS, it should be noted that our experiment is an extreme scenario with only 10 items per city, which greatly increase the chance of conflicts. Figure 7 shows the abort rate per process for both scenarios. In Scenario A, the validation site has significantly lower abort rate than other processes, while in Scenario B, the aborts are more evenly distributed. In FLACS, observer placement and the validation hierarchy are crucial parameters, and in a real system, the observer mapping would benefit from historical data on access patterns, and possibly also semi- or fully-automatized dynamic reconfiguration. The general rule is that observers for items commonly accessed together should be close in the validation hierarchy.

### Table 1. Distribution of transaction types per city.

<table>
<thead>
<tr>
<th></th>
<th>New York</th>
<th>Los Angeles</th>
<th>London</th>
<th>Paris</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book NY</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
<td>80%</td>
</tr>
<tr>
<td>Book Paris</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>Book NY+LA</td>
<td>10%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Book London+Paris</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
</tbody>
</table>

![Abort rate](image1)

**Fig. 6.** Abort rate.

![Commits vs. aborts, per process.](image2)

**Fig. 7.** Commits vs. aborts, per process.
6 Related Work

Most recent proposals for efficient data storage in cloud systems are based on
decentralization with partitioning. In ElasTraS [4], transactions spanning parti-
tions are only allowed as short transactions with predeclared read sets and write
sets. Megastore [6] assumes a relatively fine-grained partitioning of the data set,
and replicates each partition across a subset of servers. Consistency is achieved
by running Paxos within each partition. Spinnaker [5] coordinates updates in
the same way as Megastore, but executes consistent reads directly at the leader
of each partition. Although more fault-tolerant than FLACS, they do not pro-
vide consistency across partitions (Megastore provides two-phase commit, but
without serializability).

In a wide-area setting, approaches based on atomic multicast have been lim-
ited by the message delay required to order every update among all processes.
The protocols in [13, 14] build upon atomic multicast, but they target a wide
area setting through partial replication, requiring message ordering only within
the group of replicas that together manage the read sets and write sets of the
transaction. This differs from FLACS since FLACS allows full replication, but
only requires coordination among a limited set of participants (the observers).

Many real-world systems are deployed on top of non-transactional data stores
such as Amazon’s Dynamo [1] and Cassandra [2]. Both provide eventual con-
sistency by committing updates with synchronization among only a subset of
participating sites, and the new values are then propagated among other replicas
in the background. Updates are versioned using vector clocks, and in the case
of conflicts updates are reconciled by the application. Although efficient, lacking
transactions is a significant disadvantage for systems managing critical data such
as audit records, reservations, or financial data.

Microsoft’s Azure [15] and Google’s Spanner [8] also provide large-scale trans-
actions for cloud applications. Azure is known to give good performance [16]
through a master-slave approach, but publicly available details are scarce. Com-
pared to FLACS the transaction latency of any master-based approach will be
worse for clients far from the master site, since every update transaction needs
at least one message exchange with the master. Spanner provides both high
availability through Paxos, replication across wide area networks, and consis-
tency through Multi Version Concurrency Control (using global timestamps).
But Spanner is hard to deploy; one obstacle for widespread adoption is that to
provide global timestamps, Spanner depends on precisely synchronized clocks
and demands a relatively complex infrastructure involving GPS hardware and
atomic clocks. Our approach with logical ordering through the validation hier-
archy provides a simpler, more generic solution.

7 Conclusion

We have defined a new approach to ensure consistency in cloud-based database
systems. The main features of our approach are a method for incremental order-
ing and a distributed hierarchical validation procedure. Together, these features allow most transactions to be validated near or at the originating site.

We have formalized the entire protocol in Real-Time Maude, and our Real-Time Maude simulations show, as expected, that this approach outperforms a more classical approach where validation takes place at centralized master site.

A number of systems for cloud-based data management use Paxos for high availability. We believe FLACS could be combined with one of these, e.g., Megastore, to provide both high availability and consistency across partitions.

References

Paper 3: Formal Modeling and Analysis of Google’s Megastore in Real-Time Maude
Formal Modeling and Analysis of Google’s Megastore in Real-Time Maude

Jon Grov$^{1,2}$ and Peter Csaba Ölveczky$^{1,3}$

$^1$ University of Oslo
$^2$ Bekk Consulting AS
$^3$ University of Illinois at Urbana-Champaign

Abstract. Cloud systems need to replicate data to ensure scalability and high availability. To enable their use for applications where consistency of the data is important, cloud systems should provide transactions. Megastore, developed and widely applied at Google, is one of very few cloud data stores that provide transactions; i.e., both data replication, fault tolerance, and data consistency. However, the only publicly available description of Megastore is short and informal. To facilitate the widespread study, adoption, and further development of Megastore’s novel approach to transactions on replicated data, a much more detailed and precise description is needed. In this paper, we describe an executable formal model of Megastore in Real-Time Maude that we have developed. Our model is the result of many iterations resulting from correcting design flaws uncovered during Real-Time Maude analysis. We describe our model and explain how it can be simulated for QoS estimation and model checked to verify functional correctness.

1 Introduction

Cloud systems enable customers to deploy applications in a highly scalable and available infrastructure. Key to these features is replication: several copies of customer data in geographically distributed data centers allow cloud services to cope with peaks in system load, as well as with network and site failures.

Many applications require database facilities for storing valuable data. Databases provide transactions: for a given sequence of read and write operations on data items, the user is assured atomicity, which means that either no operation is completed or all operations are completed, and serializability, which means that the execution of concurrent transactions must provide the same result as some sequential execution. Transactions are necessary protection against inconsistency due to interleaved operations on shared data. For example, if two transactions $t_1$ and $t_2$ both read and write bank account $x$ to deposit $\$20$, it is crucial to avoid both the execution $t_1: \text{read}(x) = 10; \ t_2: \text{read}(x) = 10; \ t_1: x := 10 + 20; \ t_2: x := 10 + 20; \ t_1: \text{write}(x, 30); \ t_2: \text{write}(x, 30)$, where $t_1$’s deposit is lost, and the execution $t_1: \text{read}(x) = 10; \ t_1: x := 10 + 20; \ t_1: \text{write}(x, 30); \ t_2: \text{read}(x) = 10; \ t_2: x := 10 + 20; \ t_2: \text{write}(x, 30)$.

$^*$ This work was partially supported by AFOSR Grant FA8750-11-2-0084.
30; \( t_2 : x := 30 + 20; \) 
\( t_2 : \text{write}(x, 50); \) 
\( \text{abort}(t_1), \)

where \( t_2 \) was allowed to read \( t_1 \)’s update which was later aborted.

Some applications, such as newspaper content management and social networks like Facebook, can tolerate lower degrees of consistency. Other applications have strict consistency requirements; notable examples include stock exchange systems, online auctions, banking, and medical systems: it is clear that a lost update due to concurrent transactions could have serious consequences in a system recording the medication of hospital patients.

Transactions are among the most important features of a database management system (DBMS), since a correct implementation of atomicity and serializability impose significant challenges. To quote Michael Stonebraker [20]:

“It is possible to build your own [transaction support] on any of these systems, given enough additional code. However, the task is so difficult, we wouldn’t wish it on our worst enemy. If you need [transaction support], you want to use a DBMS that provides them; it is much easier to deal with this at the DBMS level than at the application level.”

Transaction management in the cloud, with geographical distribution and data replication, involves additional challenges because of:

- **Performance**: Concurrent access to replicas at different locations requires costly network coordination.
- **Availability**: The complexity of coordinating transactions across network sites increases significantly due to possible network and site failures.

Given the difficulties of transaction management on replicated data, we believe that formal methods are crucial to enable the use of cloud-based data stores also for applications where strong data consistency is required. First of all, formal analysis should be used to catch subtle “corner case” errors during design and development of the data store. Second, because of the complexity and criticality of such systems, it is necessary for application providers to be convinced that the cloud system indeed provides transaction support. Formal verification could be a major component in providing such assurance to application providers, just like formal methods can be used in Level A certification of critical avionics systems.

There are currently only a few cloud data stores with transaction support. Microsoft’s SQL Azure [4] uses a master-based approach to coordination, which reduces fault-tolerance and gives worse performance for clients far from the master site. Google’s Spanner [6] demands a complex infrastructure involving GPS hardware and atomic clocks, which reduces its applicability. Google’s Megastore [2] provides replication and transactions through a replicated transaction log. Despite its relatively low performance, Megastore is used by Google for many well-known services such as GMail, Android Market, and Google+ [6], and is offered to customers using Google’s cloud-based application platform AppEngine.

In this paper, we use the rewriting-logic-based Real-Time Maude language and tool [17] to formally model, simulate, and model check Megastore. The design of Megastore is informally described in the paper [2]. However, designing a
Our contributions are:

1. We provide a precise, formal model of Megastore, which includes many details and aspects not even described informally in [2]. Because of the ambiguity and the lack of detail in the informal specification, we had to make a number of assumptions and design choices in our formalization. Our model is the result of several modifications resulting from extensive model checking during this formalization process.

2. We show how Megastore can be model checked and probabilistically simulated using Maude and Real-Time Maude.

3. We provide a general method for analyzing serializability in distributed transactional systems with replicated data.

Our formal model should facilitate further research on the Megastore approach. In particular, we are working on combining Megastore with the FLACS approach [8] to provide serializable transactions also across partitions.

The rest of the paper is organized as follows: Section 2 gives some background on Maude and Real-Time Maude. Section 3 presents an overview of Megastore and its approach to fault-tolerance. Section 4 describes our formal model of Megastore. Section 5 explains how we have formally analyzed our model. Finally, Section 6 discusses related work and gives some concluding remarks.

2 Maude and Real-Time Maude

Real-Time Maude [13] is a language and tool that extends Maude [5] to support the formal specification and analysis of real-time systems. The specification formalism emphasizes ease and generality of specification, and is particularly suitable for modeling distributed real-time systems in an object-oriented style. Real-Time Maude specifications are executable, and the tool provides a variety of formal analysis methods, including simulation, reachability analysis, and LTL and timed CTL model checking.

2.1 Maude

Maude [5] is a rewriting-logic-based formal language and high-performance simulation and model checking tool. A Maude module specifies a rewrite theory [10,3] \((\Sigma, E \cup A, R)\), where:

- \(\Sigma\) is an algebraic signature; that is, a set of declarations of sorts, subsorts, and function symbols.
- \((\Sigma, E \cup A)\) is a membership equational logic theory [11], with \(E\) a set of possibly conditional equations and membership axioms, and \(A\) a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed modulo the axioms \(A\). The theory \((\Sigma, E \cup A)\) specifies the system’s state space as an algebraic data type.
– $R$ is a collection of labeled conditional rewrite rules specifying the system’s local transitions, each of which has the form $[l] : t \rightarrow t'$ if $\bigwedge_{j=1}^{m} \text{cond}_j$, where each $\text{cond}_j$ in the condition is either an equality $u_j = v_j$ (if $u_j$ and $v_j$ have the same normal form) or a rewrite $t_j \rightarrow t'_j$ (if $t_j$ rewrites to $t'_j$ in zero or more rewrite steps), and $l$ is a label. Such a rule specifies a one-step transition from a substitution instance of $t$ to the corresponding substitution instance of $t'$, provided the condition holds. The rules are universally quantified by the variables appearing in the $\Sigma$-terms $t, t', u_j, v_j, t_j, t_j'$, and are applied modulo the equations $E \cup A$.

We briefly summarize the syntax of Maude and refer to [5] for more details. Operators are introduced with the $\text{op}$ keyword: $\text{op } f : s_1 \ldots s_n \rightarrow s$. They can have user-definable syntax, with underbars ‘_’ marking the argument positions. Some operators can have equational attributes, such as $\text{assoc}$, $\text{comm}$, and $\text{id}$, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are used by the Maude engine to match terms modulo the declared axioms. An operator can also be declared to be a constructor ($\text{ctor}$) that defines the carrier of a sort. Equations and rewrite rules are introduced with, respectively, keywords $\text{eq}$, or $\text{ceq}$ for conditional equations, and $\text{rl}$ and $\text{crl}$. The mathematical variables in such statements are declared with the keywords $\text{var}$ and $\text{vars}$, or can be introduced on the fly in a statement without being declared previously, in which case they have the form $\text{var:sort}$. An equation $f(t_1, \ldots, t_n) = t$ with the $\text{otherwise}$ (for “otherwise”) attribute can be applied to a subterm $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.

In object-oriented Maude modules, a class declaration

```plaintext
class C | att_1 : s_1, \ldots , att_n : s_n .
```

declares a class $C$ with attributes $\text{att}_1$ to $\text{att}_n$ of sorts $s_1$ to $s_n$. An object of class $C$ in a given state is represented as a term $< O : C | \text{att}_1 : \text{val}_1, \ldots , \text{att}_n : \text{val}_n >$ of sort $\text{Object}$, where $O$, of sort $\text{Obj}$, is the object’s identifier, and where $\text{val}_1$ to $\text{val}_n$ are the current values of the attributes $\text{att}_1$ to $\text{att}_n$. A message is a term of sort $\text{Msg}$, where the declaration $\text{msg } m : s_1 \ldots s_n \rightarrow \text{Msg}$ defines the syntax of the message ($m$) and the sorts ($s_1 \ldots s_n$) of its parameters.

The state is a term of the sort $\text{Configuration}$ in a concurrent object-oriented system, and has the structure of a multiset made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is multiset rewriting supported directly in Maude. Since a class attribute may have sort $\text{Configuration}$, we can have hierarchical objects which contain a subconfiguration of other (possibly hierarchical) objects and messages.

---

4 An equational condition $u_i = w_i$ can also be a matching equation, written $u_i := w_i$, which instantiates the variables in $u_i$ to the values that make $u_i = w_i$ hold, if any.

5 Operationally, a term is reduced to its $E$-normal form modulo $A$ before any rewrite rule is applied in Real-Time Maude.
The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

\[
\text{rl } [1] : \ m(0,w) \\
\quad < O : C | a1 : x, a2 : O', a3 : z > \\
\quad \rightarrow \\
\quad < O : C | a1 : x + w, a2 : O', a3 : z > \\
\quad m'(O',x) .
\]

defines a parameterized family of transitions (one for each substitution instance) in which a message \( m \), with parameters \( O \) and \( w \), is read and consumed by an object \( O \) of class \( C \), the attribute \( a1 \) of the object \( O \) is changed to \( x + w \), and a new message \( m'(O',x) \) is generated. The message \( m(O,w) \) is removed from the state by the rule, since it does not occur in the right-hand side of the rule. Likewise, the message \( m'(O',x) \) is generated by the rule, since it only occurs in the right-hand side of the rule. By convention, attributes whose values do not change and do not affect the next state of other attributes or messages, such as \( a3 \), need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as \( a2 \), can be omitted from right-hand sides of rules.

A subclass inherits all the attributes and rules of its superclasses.

**Formal Analysis in Maude.** A Maude module is executable under some conditions, such as the equations being confluent and terminating, possibly modulo some structural axioms, and the theory being coherent [5].

Maude’s *rewrite command* simulates one of the many possible system behaviors from the initial state by rewriting the initial state. Maude’s *search command* uses a breadth-first strategy to search for states that are reachable from the initial state, match the search pattern, and satisfy the search condition.

Maude’s *linear temporal logic model checker* analyzes whether each behavior satisfies a temporal logic formula. State propositions, possibly parametrized, are operators of sort \( \text{Prop} \), and their semantics is defined by equations of the form

\[
\text{ceq } \text{statePattern } \models \text{prop } = b \text{ if cond}
\]

for \( b \) a term of sort \( \text{Bool} \), which defines the state proposition \( \text{prop} \) to hold in all states \( t \) such that \( t \models \text{prop} \) evaluates to \text{true}. A temporal logic formula is constructed by state propositions and temporal logic operators such as \text{True}, \text{False}, \sim \text{ (negation)}, \land, \lor, \rightarrow \text{ (implication)}, \square \text{ (“always”)}, \triangledown \text{ (“eventually”)}, \text{U} \text{ (“until”)}, \text{W} \text{ (“weak until”). The command

\[
(\text{red modelCheck}(t, \text{formula}) .)
\]

then checks whether the temporal logic formula \( \text{formula} \) holds in all behaviors starting from the initial state \( t \). Such model checking terminates if the state space reachable from the initial state \( t \) is finite.
2.2 Real-Time Maude

A Real-Time Maude [17] timed module specifies a real-time rewrite theory [16], that is, a rewrite theory \( \mathcal{R} = (\Sigma, E \cup A, R) \), such that:

1. \( (\Sigma, E \cup A) \), contains a specification of a sort \( \text{Time} \) defining the (discrete or dense) time domain.
2. The rules in \( R \) are decomposed into:
   - “ordinary” rewrite rules that model instantaneous change that is assumed to take zero time, and
   - tick (rewrite) rules of the form
     \[
     \text{crl} [l] : \{t\} \Rightarrow \{t'\} \text{ in time } u \text{ if } \text{cond}
     \]
     that model the elapse of time in a system, where \( \{ \_ \} \) is a constructor of a new sort \( \text{GlobalSystem} \) and \( u \) is a term of sort \( \text{Time} \) denoting the duration of the rewrite.

The initial state of a system must be equationally reducible to a term \( \{t_0\} \). The form of the tick rules then ensures uniform time elapse in all parts of a system.

Real-Time Maude extends Maude’s analysis features to the real-time setting. Real-Time Maude’s timed fair rewrite command simulates one behavior of the system up to a certain duration. It is written with syntax

\[
\text{tfrew } t \text{ in time } \leq \text{timeLimit} .)
\]

where \( t \) is the term to be rewritten (“the initial state”), and \( \text{timeLimit} \) is a ground term of sort \( \text{Time} \). Real-Time Maude extends Maude’s search command to search for states that can be reached within a given time interval from the initial state.

Real-Time Maude provides both unbounded and time-bounded LTL model checking. The unbounded model checking command

\[
\text{mc } t \models u \text{ formula .}
\]

checks whether the temporal logic formula \( \text{formula} \) holds in all behaviors starting from the initial state \( t \). When the reachable state space is infinite, time-bounded LTL model checking, in which each behavior starting in \( t \) is only analyzed up to a certain time bound, can be used to ensure termination of the model checking.

3 Overview of Megastore

A data store is a system providing functionality to write and access persistent data. Data stores are used to offload the complexity of data management from individual applications by providing transaction support, access control, and/or fault recovery. A data store often uses replication to ensure high availability in the presence of site and/or network failures: several copies of the same data are stored at different locations.

Megastore [2] is a data store offering very high availability and transaction support. It is deployed within Google’s own cloud infrastructure. In addition
to being widely used internally at Google, Megastore is also used by Google’s customers through the cloud-based application platform AppEngine. Megastore handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of data across many global data centers [2].

Data are replicated among sites (data centers), and Megastore can tolerate failure of up to $n-1$ replicas, with $n$ the total number of replicas. A transaction is a sequence of read and write operations on entities, followed by a commit request. Clients can issue transaction requests from any site replicating the relevant data, and updates are propagated to the other replicas before the transaction commits.

In Megastore, data are stored as entities, each entity being a set of key-value pairs. Entities are organized into entity groups. Transactional serializability is only guaranteed for operations within the same entity group.

Initially, all operations in a transaction are executed locally at the receiving site. When a commit request is issued, a coordination procedure between the sites is used to decide whether or not the transaction is valid and can be committed. If not, usually due to some concurrent update of the same data, the entire transaction is aborted and must be restarted from the beginning.

Megastore uses the Paxos protocol [9] for coordinating updates. This allows most transactions to complete even in the presence of site and/or network failures. Section 3.1 explains the behavior of Megastore in the absence of failures in more detail, and Section 3.2 explains how Megastore deals with faults.

3.1 Megastore without Failures

Any Megastore site $S$ may receive transaction requests for entities replicated at $S$. Entities are versioned, and Megastore provides reads with different levels of consistency. We focus here on current reads, which give the most recent version written. Any transaction updating an entity must perform a current read before performing the update.

In the absence of failures, current read operations are performed locally. Each site has a coordinator, which is always informed whether the local replica is up-to-date. When a current read is issued, it is executed locally if and only if granted by the local coordinator. Otherwise, a majority read is required, as explained in Section 3.2.

During the execution of a transaction, read operations are completed immediately, while write operations are buffered. When receiving the commit request, the site initiates the coordination procedure. Megastore’s approach to combine availability with serializability is to partition data into relatively small units (entity groups), and maintain a separate transaction log for each entity group. This log is replicated, and serializability within the entity group is ensured, since, at any given time, only one transaction is allowed to update the log.

Therefore, when committing a transaction $t$, the site receiving $t$, denoted the originating site of $t$, performs the following steps for each entity group accessed by $t$: (i) prepare a log entry containing $t$’s updates within this entity group, and propose this to the other sites as the next entry in the replicated transaction
log; and (ii) if accepted by a majority of replicas, t’s log entry is added to the replicated log, and its updates are applied at all sites.

3.2 Megastore in the Presence of Failures

Failures may cause some processes to stop responding and/or may block network messages from being delivered. Fault tolerance implies that a transaction execution must be able to proceed even if some replicating sites are unable to participate in the coordination procedure.

Majority Read. If a site has missed previous updates, a current read must synchronize with other sites to retrieve the correct data. Since the coordinator is assumed to always be informed whether the local replica is in sync, it must be reachable even if a site is unable to receive an update. Otherwise the update is blocked until the entire site is confirmed to be down by Megastore’s underlying failure detection mechanism.

If a site is missing updated values for the entity requested by a current read, it performs a majority read for the relevant entity group before proceeding. During the majority read, the local site $s_l$ requests from each other replicating site $s_r$ the most recent log position known to be valid by $s_r$. When $s_l$ has received a reply from a majority of the replicating sites, it initiates a catchup: any log position missing at $s_l$ is requested from some updated site. When the catchup is complete, the local coordinator marks the replica as valid, and the current read operation can proceed.

Update Coordination. Megastore uses Paxos [9] to commit transactions in the presence of failures. Paxos is a generic consensus protocol for distributed systems which consists of the following phases:

1. Agree on a leader. The leader for this Paxos round then proposes a value to the participating processes.
2. Once the proposed value is acknowledged by at least a majority of the processes, the leader informs all participants about the decision.

In the presence of failures, this may be insufficient to reach consensus, in which case a new round is initiated where another process becomes the leader.

In Megastore, the originating site proposes an entry in the replicated log, it initiates a new run of Paxos to ensure consensus. Megastore optimizes Paxos by including in each log entry the Paxos-leader for the next log entry. Phase 1 is therefore replaced by a request from the originating site directly to the leader, before the log entry is multicast to the other replicating sites. In the case of conflict, i.e., if multiple sites request different values for the same log position, Paxos ensures that only one is elected, and the other is aborted.
4 Formalizing Megastore in Real-Time Maude

This section explains how we have formalized Megastore in Real-Time Maude. Our model contains 56 rewrite rules, of which we only present 15 in this paper. The entire executable formal specification is available at http://folk.uio.no/jongr/megastore/maude.html. Section 4.1 lists our system assumptions, Section 4.2 presents our model of Megastore in the absence of failures, and Section 4.3 shows our model in the presence of failures.

4.1 System Assumptions

Based on the description in [2], we make the following system assumptions:

– Megastore is deployed across geographically distant sites connected by a wide-area network. The network delays between two nodes can therefore vary significantly, and we do not assume FIFO delivery between the same pair of nodes.
– A site always knows all the other replicating sites for an entity group.
– Sites can fail and recover spontaneously, and messages can be dropped due to site or network failures.
– Coordinators are supposed to be very stable. Furthermore, Megastore requires that the coordinator of each running site is accessible; otherwise update transactions are blocked until the given replica is confirmed down and can be excluded. We therefore assume that coordinators are always available.
– Small time differences caused by clock skews of the local clocks are ignored.

4.2 The Model without Failure Handling

We model Megastore in an object-oriented way, where the global 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:

```latex
class Site |
  entityGroups : Configuration,
  localTransactions : Configuration,
  coordinator : EntGroupLogPosPairSet .
```

The attribute `entityGroups` contains one `EntityGroup` object for each entity group replicated at the site, and the attribute `localTransactions` contains one `Transaction` object for each active transaction originating at the site. The attribute `coordinator` denotes the local coordinator state for each entity group, and is a `;`-separated set of pairs `eglp(eg,lp)`, denoting that the entity group `eg` is up-to-date at log position `lp`, and pairs `invalidCstate(eg,lp)`, denoting that the local replica of `eg` may be missing some log entries at or before `lp`. 
**Entity Groups.** Each local entity group copy is modeled as an object instance of the following class:

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

The attribute `entitiesState` describes the available versions of each entity in the entity group. Each such record is a term `entity(eg, i) |-> (lpos(pj), vj) :: ... :: (lpos(pk), vk),` where `entity(eg, i)` denotes the ith entity of the entity group `eg`, and `(lpos(pj), vj)` is an entity version containing the value `vj`, created at log position `pj`.

The attribute `transactionLog` denotes the local copy of the replicated transaction log which is the core of Megastore’s replication protocol. Each log entry belongs to a given log position. 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`.

The attribute `replicas` denotes the set of sites replicating this entity group. The attribute `proposals` denotes the local state in ongoing Paxos processes involving this entity group. It contains two types of values: `proposal(s, t, lp, pm)`, which represents a request from site `s` to become the leader for log position `lp` on behalf of transaction `t`, and `accepted(s, le, pm)`, which states that this site has accepted Paxos proposal number `pm` containing the log entry `le` from site `s`.

Megastore executes write operations in two steps: (i) write to the log, which occurs immediately when the chosen log entry is committed; and (ii) updating the actual data in the `entityState`. The attribute `pendingWrites` maintains a list of write operations waiting to be applied to the `entityState`.

**Transactions.** A transaction request is a `::`-separated list of current read operations `cr(e)` and write operations `w(e, v)`. Transactions being executed are modeled as object instances of the following class:

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

The attribute `operations` initially contains the transaction request. During execution, operations are removed from this list. For a read operation the resulting entity is stored in the attribute `reads`. The attribute `writes` is used to buffer write operations. `status` denotes the overall transaction status: `idle`, `executing(lp, t)` (the transaction is executing at log position `lp` and will continue executing for time `t`), and `in-paxos`, which is used during the commit process. The attributes `readState` and `paxosState` store transient data for each entity group accessed by the transaction execution.
Modeling Communication. We assume that the communication delay is nondeterministic. The set of possible delays depends on the sender and receiver, and is given by \( \text{possibleMsgDelays}(s_1, s_2) \) as a ';' separated set of time values:

\[
\text{sort TimeSet.} \quad \text{subsort Time < TimeSet.} \\
\text{op emptyTimeSet: -> TimeSet.} \\
\text{op _;_: TimeSet TimeSet -> TimeSet [ctor assoc comm id: emptyTimeSet].} \\
\text{op possibleMsgDelays: SiteId SiteId -> TimeSet [comm].}
\]

A "ripe" message has the form

\[
\text{msg } mc \text{ from } sender \text{ to } receiver
\]

where \( mc \) is the message content. A message in transit that will be delivered after \( t \) time units is modeled by a term \( \text{dly}(\text{msg } mc \text{ from } sender \text{ to } receiver, t) \):

\[
\text{sort DlyMsg.} \\
\text{subsort Msg < DlyMsg < NEConfiguration.} \\
\text{op dly: Msg Time -> DlyMsg [ctor right id: 0].} \\
\text{msg msg_from_to_: MsgContent Oid Oid -> Msg.}
\]

Nondeterministically selecting any possible delay from \( \text{possibleMsgDelays}(s_1, s_2) \) can be done using a matching equation in the condition of the rewrite rule. A rule creating a single message with nondeterministic delay should have the form\(^6\)

\[
\text{var T : Time.} \quad \text{var TS : TimeSet.} \\
\text{crl [sendMsgAnd...]} : \\
\quad < \text{SID : Site | ...} > \ldots \\
\quad = \ldots \\
\quad < \text{SID : Site | ...} > \ldots \\
\quad \text{dly}(\text{msg } mc \text{ from } \text{SID to } \text{SID'}, T) \quad \text{if ... \& T ; TS := possibleMsgDelays(SID,SID').}
\]

A site must often multicast a message to all other sites replicating an entity group. The delay of each single message must of course be selected nondeterministically. A naïve solution to model such multicast by generating the corresponding single messages in any order would be prohibitively expensive from a model checking perspective: if there are \( n \) recipients, there would be \( n! \) different orders in which these messages could be created. We can therefore use a "partial order reduction" technique, in which the messages are sent in a certain order. In particular, the replicas attribute of an EntityGroup object contains sets of tuples \( \text{egrs(SID,N)} \), where the second component is unique in the group. We can therefore order the set of recipients, and generate the single messages in this order, reducing the number of possible orders of sending the messages from \( n! \) to 1. The following rewrite rule is used to "dissolve" a "multicast message"\(^7\)

\[
\text{multiCast } mc \text{ from } \text{SID to } \text{EGRS}
\]

\(^6\) We do not show most variable declarations, but follow the Maude convention that variables are written with capital letters.
into single messages with nondeterministically selected delays:

\[
\text{op } \text{multiCast}_\text{from}_\text{to} : \text{MsgContent} \text{ Oid} \text{ EntityGroupReplicaSet} \rightarrow \text{Configuration} \ [\text{ctor}] . \\
\text{eq } \text{multiCast} \ \text{MC} \text{ from } \text{SID} \text{ to } \text{noEGR} = \text{none} .
\]

\[
\text{crl} \ [\text{multiCastToUnicast}] : \\
\text{multiCast} \ \text{MC} \text{ from } \text{SID} \text{ to } (\text{egrs(SID', N)} ; \text{EGRS}) \\
\rightarrow \\
\text{dly}(\text{msg} \ \text{MC} \text{ from } \text{SID} \text{ to } \text{SID}', \text{T}) \\
(\text{multiCast} \ \text{MC} \text{ from } \text{SID} \text{ to } \text{EGRS}) \\
\text{if } N = \text{smallest}(\text{egrs(SID', N)} ; \text{EGRS}) \\
\text{\\} \ \text{T} ; \text{TS} := \text{possibleMsgDelays(SID, SID')} .
\]

Therefore, to multicast a message with message content \( mc \) to all other sites replicating the entity group \( EG \), a rule of the following form \( could \) be used:

\[
\text{rl} \ [\text{multicastReplicatingSites}] \\
< \text{SID} : \text{Site} | \text{entityGroups} : < \text{EG} : \text{EntityGroup} | \text{replicas} : \text{EGRS}, \ldots > \ldots \\
\rightarrow \\
< \text{SID} : \text{Site} | \ldots > \ldots \\
(\text{multiCast} \ \text{mc} \text{ from } \text{SID} \text{ to } \text{EGRS}) .
\]

However, this would still involve \( n + 1 \) rewrite steps needed to get to a state where all the single messages have been generated, unnecessarily increasing the state space explored during model checking. By using rewrite conditions, we can replace the above rewrite rule with a rule:

\[
\text{var } \text{SINGLE-MSGS} : \text{NConfiguration} .
\]

\[
\text{crl} \ [\text{multicastReplicatingSitesEfficient}] \\
< \text{SID} : \text{Site} | \text{entityGroups} : < \text{EG} : \text{EntityGroup} | \text{replicas} : \text{EGRS}, \ldots > \ldots \\
\rightarrow \\
< \text{SID} : \text{Site} | \ldots > \ldots \\
\text{SINGLE-MSGS} \\
\text{if } (\text{multiCast} \ \text{mc} \text{ from } \text{SID} \text{ to } \text{EGRS}) \Rightarrow \text{SINGLE-MSGS} .
\]

where \( \text{SINGLE-MSGS} \) is a variable of some sort containing sets of delayed messages, but no occurrences of the \text{multiCast} operator. In this rewrite rule, all the single messages are created in \( one \) rewrite step, drastically reducing the reachable state space. (The local “partial order reduction” is still important, since it significantly reduces the number of behaviors explored by Maude during the evaluation of the rewrite condition; however, it does not reduce the reachable state space.)

**Dynamic Behavior.** The dynamic behavior of Megastore \( without \) fault tolerance features is modeled by 16 rewrite rules, 7 of which are given below. A transaction request with operations \( ol \) and name \( t \) is sent to a site \( s \) by a message \text{newTrans}(s,t,ol). When a site gets such a transaction request, the site adds a corresponding transaction object to its \text{localTransactions}.

\[
\text{rl} \ [\text{newTrans}] : \\
\text{newTrans}(\text{SID}, \text{TID}, \text{QL}) \\
< \text{SID} : \text{Site} | \text{localTransactions} : \text{LOCALTRANS} >
\]
If the next operation in an idle transaction $TID$ is a current read (cr) of an entity $entity(EG,N)$ in entity group $EG$, the transaction goes to the local state executing($LP, \text{readDelay}$), where $LP$ is the local coordinator's current log position for $EG$, and $\text{readDelay}$ is the time it takes to perform a read operation:

```plaintext
\text{crl [startCurrentLocalRead]} :
< SID : Site | coordinator : (eglp(EG, LP) ; CES),
entityGroups : EGROUPS
< EG : EntityGroup | pendingWrites : emptyPWList >
localTransactions : LOCALTRANS
< TID : Transaction | operations : cr(entity(EG,N)) :: OL,
status : idle > >
```

If not $\text{containsUpdate(entity(EG,N), OL)}$ and $\text{inConflictWithRunning(EG, LOCALTRANS)}$.

To avoid locals conflicts, a site only allows one active update transaction for each entity group. The condition of the rewrite rule blocks the read request if the transaction $TID$ contains an update operation on $entity(EG,N)$ until there are no other active conflicting transactions.

When the executing timer expires (i.e., becomes zero), the read operation completes and adds the version read at the given log position to reads. The transaction status is then set to idle, allowing execution to proceed:

```plaintext
\text{rl [endCurrentLocalRead]} :
< SID : Site |
entityGroups : EGROUPS
< EG : EntityGroup | entitiesState : (entity(EG,N) |-> EVERSIONS) ; BSTATE >,
localTransactions : LOCALTRANS
< TID : Transaction | operations : cr(entity(EG,N)) :: OL, readState : RSTATE,
status : executing(LP, 0), reads : READS > >
```

A write operation is moved to the buffer writes, and will be executed once the transaction is committed:

```plaintext
\text{rl [bufferWriteOperation]} :
< SID : Site | localTransactions : LOCALTRANS
< TID : Transaction | operations : w(EID, VAL) :: OL, writes : WRITEOPS,
```

...
When all operations in the `operations` list are completed (reads) or buffered (writes), the transaction is ready to commit. All buffered updates are merged into a candidate log entry. If the transaction updates entities from several entity groups, one log entry is created for each group.

For each such entity group, the first step is to send the candidate log entry to the leader for the _next_ log position, which was selected during the previous coordination round. The rule for initiating Paxos is modeled as follows:

crl [initiateCommit] :
< SID : Site | entityGroups : EGROUPS,
localTransactions : LOCALTRANS
< TID : Transaction | operations : emptyOpList,
writes : WRITEOPS, status : idle
readState : RSTATE, paxosState : PSTATE > >
=>
< SID : Site |
localTransactions : LOCALTRANS
< TID : Transaction | paxosState : NEW-PAXOS-STATE,
status : in-paxos > >

ACC-LEADER-REQ-MGS
if EIDSET := getEntityGroupIds(WRITEOPS) /
NEW-PAXOS-STATE := initiatePaxosState(EIDSET, TID, WRITEOPS,
SID, RSTATE, EGROUPS)
/\ (createAcceptLeaderMessages(SID, NEW-PAXOS-STATE)) => ACC-LEADER-REQ-MGS .

getchEntityGroupIds(WRITEOPS) contains entity groups accessed by operations in WRITEOPS, and NEW-PAXOS-STATE contains one record for each entity group. These records contain the log position that TID requests to update and the candidate log entry le. The operator `createAcceptLeaderMessages` generates an `acceptLeaderReq` message to the leader of each entity group containing the transaction id TID and candidate log entry le.

The execution then proceeds as follows for each entity group:

1. When the leader $s_l$ receives an `acceptLeaderReq` message from the originating site $s_o$ for the transaction TID, the leader site inspects the proposals set for the given entity group, to check whether it has previously accepted some value for this log position and entity group. If so, there is a conflict, and $s_l$ signals this with a message to the originating site of TID, which aborts the transaction. Otherwise, $s_l$ sends an `acceptLeaderRsp` message to $s_o$.
2. When it receives an `acceptLeaderRsp` message, the originating site proceeds by multicasting the log entry to the other replicating sites. Each recipient of this message must verify that it has not already granted an accept for this log position. If so, the recipient replies with an accept message to the originating site. We show this rule below.
3. After receiving an `acceptAllRsp` message from all replicating sites, the originating site confirms the commit by multicasting an `applyReq` message. When receiving this message, a recipient appends the proposed log entry to the `transactionLog` of the entity group, and the update operations are added to the `pendingWrites` list. With this, the transaction is committed.

The following rule shows the rule from step 2 where a replicating site receives an `acceptAllReq` message. The site verifies that it has not already granted an accept for this log position (since messages could be delayed for a long time, it checks both the transaction log and received proposals). If there are no such conflicts, the site responds with an accept message, and stores its accept in `proposals` for this entity group. The record `(TID', LP SID OL)` represents the candidate log entry, containing the transaction identifier `TID'`, the log position `LP`, the proposed leader site `SID`, and the list of update operations `OL`.

\[
\text{crl} \ [\text{rcvAcceptAllReq}] : \\
\text{msg} \ \text{acceptAllReq}(TID, \ EG, (TID', LP, SID, OL), \ PROPNUM) \ \text{from} \ SENDER \ \text{to} \ THIS) \\
< \ THIS : \ Site | \\
\text{entityGroups} : \ \text{EGROUPS} \\
< \ EG : \ EntityGroup | \ \text{proposals} : \ \text{PROPSET}, \ \text{transactionLog} : \ \text{LEL} > > \\
=> \\
< \ THIS : \ Site | \\
\text{entityGroups} : \ \text{EGROUPS} \\
< \ EG : \ EntityGroup | \\
\text{proposals} : \ \text{accepted}(\ SENDER, (TID', LP, SID, OL), \ PROPNUM) ; \\
\ \text{removeProposal}(LP, \ \text{PROPSET}) > > \\
\text{dly}(\ \text{acceptAllRsp}(TID, \ EG, \ LP, \ PROPNUM) \ \text{from} \ THIS \ \text{to} \ SENDER), \ T) \\
\text{if not} \ \text{(containsLPos}(LP, \ \text{LEL}) \ \text{or hasAcceptedForPosition}(LP, \ \text{PROPSET}) \\
\text{\}/} \ T ; \ TS := \ \text{possibleMessageDelay}(THIS, \ SENDER) .
\]

**Modeling Time and Time Elapse.** We follow the guidelines in [17] for modeling time in object-oriented specifications. Since an action can only be triggered by the arrival of a message, the expiration of a timer, or by another event, we use the following tick rule to advance time until the next event will take place:

\[
\text{crl} \ [\text{tick}] : \ \{\text{SYSTEM}\} \Rightarrow \ \{\delta \text{te}(\text{SYSTEM}, \ \text{mte}(\text{SYSTEM})) \\\n\text{if \ mte(\text{SYSTEM})} > 0 \ \land \ \text{mte(\text{SYSTEM})} \neq \text{INF} .
\]

The function \(\text{mte}\) denotes the minimum time that can elapse until the next event will take place, and \(\delta \text{te}\) defines the effect of time elapse on the state. For example, \(\text{mte}(\text{dly}(M,T) \ \text{REST}) = \min(T, \ \text{mte(REST)})\), which means that \(\text{mte}(m)\) is zero for a ripe message \(m\) (since \(m\) is identical to \(\text{dly}(m,0)\)). Therefore, time cannot advance when there are ripe messages in the configuration.

We import the built-in module \text{NAT-TIME-DOMAIN-WITH-INF}, which defines the time domain \text{Time} to be the natural numbers, with an additional constant \text{INF} (for \(\infty\)) of a supersort \text{TimeInf}. 
4.3 Modeling Megastore’s Fault Tolerance Mechanisms

Megastore is supposed to tolerate: (i) site failures (except for the coordinators); (ii) message loss; and (iii) arbitrarily long message delays. We have formalized these fault tolerance features using 37 rewrites rules, out of which we show only 1 rule in this paper. Our model provides fault tolerance and consistency through the following mechanisms:

- A Paxos-based commit protocol to ensure that even in the presence of multiple failure and recovery events, all available replicas agree on the value for the next log position. If the originating site $s_o$, after sending an `acceptLeaderReq` message for log position $lp$, does not receive a response from the leader of $lp$ within a certain amount of time, it attempts to become the leader itself by sending a `prepareAllReq` message to all replicating sites. When receiving a positive response from a majority of sites, $s_o$ proceeds with the accept phase by multicasting an `acceptAllReq` message to all replicating sites. If at this point $s_o$ fails to receive an `acceptAllRsp` message from a majority of sites, it re-initiates the prepare step after a nondeterministic backoff.

- If a replicating site $s_r$ is unable to apply an update, the coordinator at $s_r$ must ensure that the site avoids serving invalid data. After obtaining a `acceptAllRsp` message from a majority of the replicating sites, the originating site sends an `invalidateCoordinator` message to each site which did not respond in time to the `acceptAllReq` message.

- A majority read and catchup procedure is used to bring a replica up-to-date in case of failures. When executing a current read operation requesting an entity from an invalid entity group $eg$ (according to the coordinator), the originating site $s_o$ broadcasts a `majorityRead` request to all sites replicating $eg$. Each available recipient responds with the highest log position seen so far. When a majority of replicating sites have responded, $s_o$ sends a `catchupRequest` containing the highest received log position to one of the responding sites. If this site does not have a complete log, $s_o$ sends several catchup requests. Once $s_o$’s log is complete, the entity group is marked as valid in the coordinator.

The following rule belongs to the first mechanism above, and shows how we meet a requirement of Paxos: after a site has accepted a log entry, it can never accept another log entry for this log position. Therefore, if a replicating site receives a `prepareAllReq` message for a log position where it has already accepted a log entry, the entry is sent to the originating site in a `prepareAllRsp` message. At the originating site, the log entry for the highest proposal number seen so far is stored within the `prepare` record of `paxosState`. If the originating site has received `prepareAllRsp` from a majority of the participating sites (hasQuorum(size(SIS ; SENDER), REPLICAS)), it initiates the `acceptAll` step by multicasting an `acceptAllReq` to all sites replicating the entity group $EG$:

\[
\text{crl [recvPrepareAllRspWithValue] :} \\
\text{(msg prepareAllRsp(TID, EG, (TID2 LP MSID1 OL1), PROPNUM, PN) from SENDER to THIS)} \\
\text{< THIS : Site !}
\]
Site Failures. All processing is blocked and incoming messages are dropped when a site has failed. The exception is that the (co-located) coordinator of the site is supposed to be available, and be able to receive and respond to invalidateCoordinator messages even when the site is otherwise failed.

We model site failures in a modular way by enclosing the failed site object by a “wrapper”: a failed site is modeled as a term failed(< s : Site | ... >). This wrapper is declared to be a frozen operator (see [5])

op failed : Object -> Object [ctor frozen (1)] .

which ensures that no activity takes place inside the failed object.

A message arriving at a failed site is dropped, unless it is a message to the coordinator:

crl [msgWhenSiteFailure] :
  (msg MC from SENDER to SID) failed(< SID : Site | >)
=>
  failed(< SID : Site | >)
if not isInvalidateCoordinator(MC) .

crl [invalidateCoordinator] :
  (msg invalidateCoordinator(EG, LP) from SENDER to THIS)
  failed(< THIS : Site | coordinator : CES >)
=>
  failed(< THIS : Site | coordinator : applyInvalidate(EG, LP, CES) >)
(dly invalidateConfirmed(EG, LP) from THIS to SENDER, T)
if T ; TS := possibleMsgDelays(THIS,SENDER) .

In our analysis, we use “messages” siteFailure(s) and siteRepair(s) to inject failures and repairs as follows:
msgs siteFailure siteRepair : SiteId -> Msg .

crl [siteDown] :
    siteFailure(SID) < SID : Site | > => failed(< SID | >) dly(siteRepair(SID), T)
if T ; TS := possibleSiteRepairTimes .

dl [siteUp] :
    siteRepair(SID) failed(< SID : Site | >) => < SID : Site | > .

5 Formally Analyzing our Model of Megastore

We used both simulation and temporal logic model checking throughout the development of our formal model from the overview description in [2]. Simulation provided quick feedback; allowed us to analyze large systems with many sites, transactions, and failures; and “probabilistic” simulation was used for quality of service (QoS) estimation of the model. Model checking, which explores all possible system behaviors, turned out to be very useful to find a number of subtle design flaws that were not uncovered during extensive simulations.

This section shows how our model of Megastore can be formally analyzed in (Maude and) Real-Time Maude. In particular, Section 5.1 lists some parameters of our model, Section 5.2 shows how we can simulate our model for QoS estimation; Section 5.3 explains our model checking of the model without fault-tolerance features, and Section 5.4 describes the model checking of the entire model. Finally, Section 5.5 presents a general technique for formally analyzing the serializability property of transactional systems: each execution is equivalent to one in which all operations of a transaction are completed before the next transaction begins.

5.1 System Parameters

There are a number of system parameters in our model, including:

- the number of sites;
- the set of possible message delays between each pair of sites;
- the number of transactions and their arrival times;
- the set of operations in each transaction;
- the number of entities and their organization into entity groups;
- the degree of replication of the different entity groups;
- the number and time distribution of site failures, and the set of possible durations of a site failure;
- the amount of message losses; and
- the duration of the timeouts before initiating fault handling procedures.

Changing these parameters allows us to analyze the model under different scenarios. For example, to define the set of possible message delays, we need to define the function possibleMsgDelays. In some of the model checking commands, we use three sites and the following message delays:
eq possibleMsgDelays(PARIS, LONDON) = (10 ; 30 ; 80) .
eq possibleMsgDelays(PARIS, NEW-YORK) = (30 ; 60 ; 120) .
eq possibleMsgDelays(LONDON, NEW-YORK) = (30 ; 60 ; 120) .

Transactions and failures are injected into the system by (delayed) messages $\text{dly(newTrans(s,t,ol),startTime)}$ and $\text{dly(siteFailure(s),failureTime)}$. For example, some of our analyses use $\text{initTransactions}$ and $\text{initFailures}$:

crl [delayTransactions] :
  initTransactions
  => dly(newTrans(PARIS, T-K, cr(entity(EG1,0)) :: w(entity(EG1,0),value(2))), T1)
  dly(newTrans(LONDON, T-L, cr(entity(EG1,0)) :: w(entity(EG1,0),value(5))), T2)
  dly(newTrans(NEW-YORK, T-M, cr(entity(EG2,0)) :: w(entity(EG2,0),value(4))), T3)
if T1 ; TS1 := transStartTime \& T2 ; TS2 := transStartTime
\& T3 ; TS3 := transStartTime .
eq transStartTime = (10 ; 50 ; 200) .

crl [delayFailures] :
  initFailures => dly(siteFailure(LONDON), T1) dly(siteFailure(NEW-YORK), T2)
if T1 ; TS1 := ttf \& (T2 ; TS2) := ttf .
eq ttf = (40 ; 100) .

The initial state $\text{initMegastore}$ can then be defined as follows:

op initMegastore : -> GlobalSystem .
eq initMegastore = {initSites initTransactions initFailures} .

eq initSites =
  < PARIS : Site | coordinator : (eglp(EG1, lpos(0)) ; eglp(EG2, lpos(0))),
    entityGroups : entityGroupsParis, localTransactions : none >
  < LONDON : Site | coordinator : (eglp(EG1, lpos(0)) ; eglp(EG2, lpos(0))),
    entityGroups : entityGroupsLondon, localTransactions : none >
  < NEW-YORK : Site | coordinator : (eglp(EG1, lpos(0)) ; eglp(EG2, lpos(0))),
    entityGroups : entityGroupsNY, localTransactions : none > .

5.2 Simulation

We can use Real-Time Maude’s timed rewrite command to simulate the system for a certain duration:

Maude> (tfrew initMegastore in time <= 850 .)

\(< LONDON : Site | coordinator : (eglp(EG1,lpos(0)); eglp(EG2,lpos(1))),
  entityGroups : ( < EG1 : EntityGroup |
    entitiesState : entity(EG1,0) |-> lpos(0)value(0) ; entity(EG1,1) |-> lpos(0)value(0),
    pendingWrites : emptyPWLList, proposals : accepted(LONDON,T-L lpos(1) LONDON w(entity(EG1,0),value(5)),2),
    replicas : egr(LONDON,0,lpos(0)) ; egr(NEW-YORK,2,lpos(0)) ; egr(PARIS,1,lpos(0)),
    transactionsLog : initTrans lpos(0) PARIS emptyOpList >
  < EG2 : EntityGroup | ... >),
  localTransactions :
Although this gives very quick and useful feedback, each application of a rule which selects a value nondeterministically will select the same value. To simulate more random behaviors, and to obtain more realistic QoS estimates, we have also defined a “probabilistic” version of our model where the different delays are given by discrete probability distributions. We then add an object containing the seed to Maude’s built-in random function to the configuration, and use this random value to sample a message delay from the probability distribution. Our probability distribution for the network delays is as follows:

<table>
<thead>
<tr>
<th></th>
<th>30%</th>
<th>30%</th>
<th>30%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>London ↔ Paris</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>London ↔ New York</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>100</td>
</tr>
<tr>
<td>Paris ↔ New York</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>100</td>
</tr>
</tbody>
</table>

We generate transactions with a transaction generator for each site, which generates transaction requests at random times, with an adjustable average rate measured in transactions per second (TPS). We simulated two fully replicated entity groups. We assume a delay of 10 ms for a local read operation in accordance with the real-world measurements reported in [2].

**Simulation without Fault Injection.** With an average of 2.5 TPS and no failures, we observe the following results in a run of 200 seconds:

<table>
<thead>
<tr>
<th></th>
<th>Avg. latency (ms)</th>
<th>Commits</th>
<th>Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>122</td>
<td>149</td>
<td>15</td>
</tr>
<tr>
<td>New York</td>
<td>155</td>
<td>132</td>
<td>33</td>
</tr>
<tr>
<td>Paris</td>
<td>119</td>
<td>148</td>
<td>18</td>
</tr>
</tbody>
</table>

The relatively high abort rate is expected, since we have only two entity groups. While our calibration data are estimates based on a typical setup for this type of cloud service combined with information given in [2], our measured latency appears fairly consistent with Megastore itself [2]: “Most users see average write latencies of 100–400 milliseconds, depending on the distance between datacenters, the size of the data being written, and the number of full replicas.”

**Simulation with Fault Injection.** We have modified the above experiment by adding a fault injector that randomly injects short outages in the sites. The delays New York–Paris and New York–London are the same, assuming transatlantic backbone links from each of these cities. The delay between Paris and London reflect that network equipment and local lines increase delivery times.
mean time to failure and the mean time to repair for each site was set to 10 and 2 seconds, respectively. This is a challenging scenario where a large fraction of the transactions will experience failure on one or multiple sites. The results from our simulation are given in the following table.

<table>
<thead>
<tr>
<th></th>
<th>Avg. latency (ms)</th>
<th>Commits</th>
<th>Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>218</td>
<td>109</td>
<td>38</td>
</tr>
<tr>
<td>New York</td>
<td>336</td>
<td>129</td>
<td>16</td>
</tr>
<tr>
<td>Paris</td>
<td>331</td>
<td>116</td>
<td>21</td>
</tr>
</tbody>
</table>

Although both the average latency and the abort rate increase significantly, these results indicate that Megastore is able to maintain an acceptable quality of service under this challenging failure scenario.

5.3 Model Checking the Model without Fault Tolerance

If there are a finite number of transactions to be executed, then the main properties that the system should satisfy are:

1. All transactions will eventually finish their execution.
2. All replicas of an entity must eventually have the same value.
3. All logs for the same entity group must eventually contain the same entries.
4. The execution is serializable; i.e., it gives the same result as some execution where the transactions are executed one after the other.
5. Furthermore, from some point on, the properties 1-4 above must hold for all future states.

We use linear temporal logic model checking to verify that all possible executions from a given initial state satisfy these properties (the serializability analysis is explained in Section 5.5).

The state proposition `allTransFinished` is true in all states where all transactions have finished executing. That is, there are no `Transaction` objects remaining in a site’s `localTransactions` and there are no messages in the system:

```
vars SYSTEM REST LOCALTRANS EGS1 EGS2 : Configuration .
var M : Msg . vars ES1 ES2 : EntitySet . vars TL1 TL2 : LogEntryList .

op allTransFinished : -> Prop [ctor] .
eq {initTransactions REST} |= allTransFinished = false .
eq {< S1 : Site | localTransactions : < TID : Transaction | > LOCALTRANS > REST}
    |= allTransFinished = false .
eq {M REST} |= allTransFinished = false .
eq {SYSTEM} |= allTransFinished = true [owise] .
```

(This definition first characterizes the states where `allTransFinished` does not hold, and then the last equation, with the `owise` attribute, defines `allTransFinished` to be true for all other states.)

The following proposition `entityGroupsEqual` is true for all states where all replicas of each entity have the same value:
In the same way, we can define when all transitions logs for each entity group are equal:

\[
\text{op} \text{transLogsEqual} : \rightarrow \text{Prop} \ \text{[ctor]}.
\]

\[
\text{ceq} \left\{ \begin{array}{l}
\langle S_1 : \text{Site} \mid \text{entityGroups} : < \text{EG}_1 : \text{EntityGroup} \mid \text{transactionLog} : \text{T}_1 \rangle \ EGS_1 > \\
\langle S_2 : \text{Site} \mid \text{entityGroups} : < \text{EG}_1 : \text{EntityGroup} \mid \text{transactionLog} : \text{T}_2 \rangle \ EGS_2 > \\
\text{REST} \end{array} \right\} \models \text{transLogsEqual} = \text{false} \text{ if } \text{T}_1 \neq \text{T}_2.
\]

\[
\text{eq} \left\{ \text{SYSTEM} \right\} \models \text{transLogsEqual} = \text{true} \ \text{[owise]}.
\]

The temporal logic formula

\[
\langle [] (\text{allTransFinished} \land \text{entityGroupsEqual}) \land \neg \text{transLogsEqual} \rangle
\]

says that in \textit{all possible executions} from the initial state, \text{a state satisfying Properties 1–3 and where all subsequent states also satisfy those properties, will eventually be reached.}

In the absence of the sophisticated failure handling, this formula should hold for all possible message delays and transaction (start and execution) times. We have therefore abstracted from the real-time features of our model, such as message delays, execution times, and timers, and have transformed our model into an \textit{untimed model} that will exhibit \textit{all possible} behaviors of the system. Model checking this property for the initial state $init\text{Megastore}$ (without delays) with the three sites and three transactions can be done in Maude as follows:

\[
\text{Maude> (red modelCheck(initMegastore,} \\langle [] (\text{allTransFinished} \land \text{entityGroupsEqual}) \land \neg \text{transLogsEqual} \rangle) .)
\]

result Bool : true

That is, the desired property holds. The model checking took 950 seconds on an Intel Xeon 1.87Ghz CPU with 128 GB RAM. A simple reachability analysis showed that this untimed model has 992,992 states reachable from $init\text{Megastore}$. Both model checking and reachability analysis from $init\text{Megastore}$ extended with one transaction were aborted due to lack of memory after 11 hours.

### 5.4 Model Checking the Model with Failure Handling

The analysis in Section 5.3 shows that model checking the \textit{untimed} model is unfeasible for four transactions even \textit{without} the large fault-tolerance part. Furthermore, the fault-tolerance features of Megastore require an extensive use of timers. Therefore, we model check only the real-time version described in Section 4 when including the fault-tolerance part.

Since we consider a finite number of transactions, the desired property must now also take into account the following possibility: if a failure causes one or
more of the sites to miss the last update, leaving its coordinator invalidated, then no further transactions will arrive to initiate a majority read. Therefore, we use modified versions of the propositions in Section 5.3, that make sure that we only require equal entitiesState and transactionLog among sites where the coordinator indicates that the given entity group is up-to-date:

\[
\text{op entityGroupsEqualOrInvalid : } \to \text{ Prop} \text{ [ctor]} .
\]

\[
\text{ceq } \{< S1 : \text{Site} | \text{coordinator : eglp(EG1, LP)} ; \text{EGLP}, \text{entityGroups : } < \text{EG1 : EntityGroup} | \text{entitiesState : ES1} > \text{EGS1} >
\]

\[
< S2 : \text{Site} | \text{coordinator : eglp(EG1, LP)} ; \text{EGLP}, \text{entityGroups : } < \text{EG1 : EntityGroup} | \text{entitiesState : ES2} > \text{EGS2} >
\]

\[
\text{REST} \} \models \text{entityGroupsEqual} = \text{false} \text{ if } \text{ES1} =\neq \text{ES2} .
\]

\[
\text{eq } \{\text{SYSTEM} \} \models \text{entityGroupsEqualOrInvalid} = \text{true} \text{ [owise]} .
\]

We have model checked a number of scenarios, all with three sites, two entity groups, three transactions (each accessing one item in each entity group). The parameters we modify are: the number of possible message delays, the possible start times of a transaction, and the number of failures and their start times. In the case with possible message delays \(\{20, 100\}\), possible transaction start times \(\{10, 50, 200\}\), and one failure at time 60, the following (unbounded) Real-Time Maude model checking command verifies the desired property in 1164 seconds:

\[
\text{Maude: } (mc \text{ initMegastore } /u \leftrightarrow \[] \text{ (allTrans Finished } /\text{ entityGroupsEqualOrInvalid} \text{ }
\]

\[
\text{ / transLogsEqualOrInvalid } ) .\}
\]

\[
\text{result } \text{Bool} : \text{true}
\]

We summarize the execution time of the above model checking command for different system parameters, where \(\{n_1, \ldots, n_k\}\) means that the corresponding value is selected nondeterministically from the set. All the model checking commands that finished executing returned true. DNF means that the execution was aborted after more than 4 hours.

<table>
<thead>
<tr>
<th>Msg. delay</th>
<th>#Trans</th>
<th>Trans. start time</th>
<th>#Fail.</th>
<th>Fail. time</th>
<th>Run (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>({20, 100})</td>
<td>4</td>
<td>(19, 80) and (50, 200)</td>
<td>0</td>
<td>-</td>
<td>1367</td>
</tr>
<tr>
<td>({20, 100})</td>
<td>3</td>
<td>(10, 50, 200)</td>
<td>1</td>
<td>60</td>
<td>1164</td>
</tr>
<tr>
<td>({20, 40})</td>
<td>3</td>
<td>20, 30, and (10, 50)</td>
<td>2</td>
<td>{40, 80}</td>
<td>872</td>
</tr>
<tr>
<td>({20, 40})</td>
<td>4</td>
<td>20, 20, 60, and 110</td>
<td>2</td>
<td>{70 and {10, 130}}</td>
<td>241</td>
</tr>
<tr>
<td>({20, 40})</td>
<td>4</td>
<td>20, 20, 60, and 110</td>
<td>2</td>
<td>{30, 80}</td>
<td>DNF</td>
</tr>
<tr>
<td>({10, 30, 80}) and ({30, 60, 120})</td>
<td>3</td>
<td>20, 30, 40</td>
<td>1</td>
<td>{30, 80}</td>
<td>DNF</td>
</tr>
<tr>
<td>({10, 30, 80}) and ({30, 60, 120})</td>
<td>3</td>
<td>20, 30, 40</td>
<td>1</td>
<td>60</td>
<td>DNF</td>
</tr>
</tbody>
</table>

5.5 Model Checking Serializability

The serialization graph for a given 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 $e$ before transaction $t_2$ executed an operation on the same entity, and at least one of the operations was a write operation. It is well known that an execution of multiple transactions is serializable if and only if its serialization graph is acyclic [21].

If there is only one version of each entity, and every update therefore over-writes the previous version, the before relation follows real time. In a multi-versioned replicated data store like Megastore, we require a defined 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 we require serializability within each entity group only, and 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 therefore add to the state an object of class TransactionHistory containing the current serialization graph. Each time a transaction commits, this object reads the above message and updates its serialization graph.

The sort SerGraph defines a set of edges:

```maude
var E : Edge .
sort SerGraph . sort Edge . subsort Edge < SerGraph .
op _<_ : TransId TransId -> Edge [ctor] .
op emptyGraph : -> SerGraph [ctor] .
op _;_ : SerGraph SerGraph -> SerGraph [ctor assoc comm id: emptyGraph] .
eq E ; E = E .

class TransactionHistory | graph : SerGraph .
```

The proposition isSerializable can then be defined as expected:

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

We can therefore verify that for each state, the execution up to the current state is serializable:

```maude
Maude: (mc initMegastore /=u [] isSerializable .)
result Bool : true
```
6 Related Work and Concluding Remarks

Despite the importance of transactional data stores, we are not aware of any work on formalizing and verifying such systems. We are also not aware of any detailed description of Megastore itself beyond [2].

The paper [18] addresses the need for formal analysis of replication and concurrency control in transactional cloud data stores. Using Megastore as a motivating example, the authors propose a generic framework for concurrency control based on Paxos, and include a pseudo-code description of Paxos and a proof of how it can be used to ensure serializability. In contrast, we provide a much more detailed and formal model not only of Paxos, but of Megastore itself.

The value of Maude for formally analyzing other cloud mechanisms is demonstrated in [19], where the authors point out possible bottlenecks in a naïve implementation of ZooKeeper for key distribution, and in [7], where the authors analyze denial-of-service prevention mechanisms using Maude and PVeStA.

Real-Time Maude has been used to model and analyze a wide range of advanced state-of-the-art systems, including multicast protocols [14], wireless sensor network algorithms [15], and scheduling protocols [12]. In all these applications, Real-Time Maude analysis uncovered significant design errors that could be traced back to flaws in the original system. The work presented in this paper differs fundamentally from those applications of Real-Time Maude: in this case, our starting point was a fairly brief and informal overview paper on Megastore – in addition to a number of papers describing the underlying Paxos protocol. We therefore had to “fill in” a lot of details, in essence developing and formalizing our own version of the Megastore approach. The available source on Megastore was not detailed enough to allow us to map flaws found during Real-Time Maude model checking to flaws in the original description of Megastore. Instead, we used Real-Time Maude simulation and model checking extensively throughout our development of this very complex system to improve our model to the point where we cannot find any flaws during our model checking analyses.

Our main contribution is therefore this fairly detailed executable formal model of (our version of) Megastore. Minor contributions include general techniques for: (i) efficiently modeling multicast with nondeterministic message delays in Real-Time Maude; and (ii) model checking the serializability property of distributed transactions on replication data in (Real-Time) Maude.

We hope that our formalization contributes to further research on the Megastore approach to transactional data stores. In particular, we are planning on combining Megastore with the FLACS approach [8] to provide serializability also for transactions accessing multiple entity groups. Other future work includes defining a probabilistic version of our model in a probabilistic extension of Maude, and use the PVeStA tool [1] for statistical model checking and more advanced QoS estimation.
References

Paper 4: Increasing Consistency in Multi-Site Data Stores: Megastore-CGC and its Formal Analysis
Increasing Consistency in Multi-Site Data Stores: Megastore-CGC and its Formal Analysis

Jon Grov
University of Oslo

Peter Csaba Ölveczky
University of Oslo
University of Illinois at Urbana-Champaign

ABSTRACT
The few distributed data stores that provide support for transactions only offer fairly limited consistency guarantees. Google’s Megastore partitions the set of data into entity groups and provides serializability only for transactions accessing single entity groups. In this paper, we propose an extension to Megastore, called Megastore-CGC, that also allow us to ensure consistency for transactions accessing data from multiple entity groups. One important advantage of our approach is that most of the additional information that must be maintained by Megastore-CGC to enable cross-group validation can be added to the messages being exchanged by Megastore. This allows Megastore-CGC to offer its additional functionality while maintaining the performance and fault tolerance of Megastore. A fault-tolerant coordination protocol, such as Megastore-CGC, is a very complex artifact whose correctness is virtually impossible to prove “by hand.” We therefore formalize Megastore-CGC in Real-Time Maude, use Real-Time Maude simulations to estimate the performance of Megastore-CGC, and use Real-Time Maude model checking to automatically verify that Megastore-CGC satisfies some crucial system properties, at least for the initial system configurations considered.

1. INTRODUCTION
Multi-site data stores replicate user data in geographically distributed data centers. This improves scalability since requests can be distributed, and availability, since data are accessible even if a data center fails. However, ensuring transactional consistency in this setting is hard for multiple reasons, in particular since controlling concurrent access requires costly message coordination, and because combining fault tolerance and consistency is a highly complex and error-prone task.

Google’s Megastore [1] is one of few multi-site data stores offering transaction support. It is used both for Google’s own services such as GMail, and by Google’s clients through the Platform-as-a-Service offering Google AppEngine. Although Megastore provides strong fault tolerance and scalability, its consistency guarantees have some limitations.

Megastore’s Approach to Consistency. Megastore’s approach to combine consistency, fault tolerance, and scalability is to partition the data items (called entities) into a number of entity groups, and to maintain a replicated transaction log for each entity group. Updates to one such log are distributed among replicating sites using a variant of the Paxos [11] consensus protocol. This approach provides consistency within each entity group, since only one running update is allowed on each entity group, and fault tolerance, since update transactions are allowed to commit as long as a majority of replicas are available.

How the data are partitioned into entity groups depends on application access patterns and requirements for consistency. For an application accessing two entities \( A \) and \( B \), consistent access is only guaranteed if \( A \) and \( B \) belong to the same entity group. Large entity groups are therefore desired to ensure consistency for many different transactions types. However, this is unacceptable for scalability purposes. Since only one concurrent update is allowed per entity group, the system’s ability to serve multiple users would be severely restricted. As the following example illustrates, it can sometimes be hard (or even impossible) to find a partitioning of the entities that achieves the required levels of consistency and concurrency.

Example. Consider a hospital with thousands of employees. To enable efficient and safe allocation of personnel to tasks (both planned and emergencies), the hospital is implementing a shared scheduling system to assign each employee a status throughout the day. The scheduling system maintains a set of entities of the form \( (\text{employee}, \text{time slot}, \text{status}) \), where each employee has a set of capabilities (heart surgery, anesthesia, etc), and where status is either booked, available, or off-duty.

The scheduling system must satisfy the following critical constraints:

1. An employee can be booked for at most 12 hours during a 24-hour period.

2. Emergency preparedness requires the hospital to have a certain number of employees with a given capability and status available for every time slot. There should,
for example, always be an available heart surgeon to deal with emergencies.

To satisfy these constraints, transactions 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 to ensure that the number of available employees does not fall below the required emergency capacity.

Megastore is an attractive option for the scheduling system because of its fault tolerance features. The question is how to group the records into entity groups such that the above constraints are preserved.

One option is to group all entities into the same entity group, which allows us to check various constraints. However, simultaneous assignments (by different operators) would be impossible in this solution. Since changes in scheduling occur constantly, this is unacceptable.

The other option is to instead group all entities belonging to one employee into the same entity group. This allows us to enforce Constraint 1: If two operators simultaneously try to book the same employee such that the two updates combined would violate the 12-of-24-hours constraint, the violation will be detected by Megastore, and one of the updates will be aborted. However, with this option, we are unable to ensure Constraint 2: Let $H_1$ and $H_2$ represent two heart surgeons with status available at time slot $\tau$. At least one of them must always be available. Furthermore, let two concurrent transactions $Book-H_1$ and $Book-H_2$ attempt to book $H_1$ and $H_2$, respectively, at time $\tau$. If $H_1$ and $H_2$ belong to different entity groups, Megastore cannot ensure consistency across $H_1$ and $H_2$. It is therefore possible that both $Book-H_1$ and $Book-H_2$ see that the other heart surgeon is available and therefore, both bookings are allowed, leading to the violation of Constraint 2.

Similar scenarios occur in other applications; e.g., in systems used for resource management and planning, such as for flight reservations, where we want to allow multiple seat reservations at one flight while preserving constraints such as maximum number of infants. Likewise, in e-commerce applications, there could be a limited number of items available for a given price.

This motivates our work to extend Megastore to support consistent transactions across multiple entity groups, while maintaining Megastore’s scalability and reliability features.

**Megastore-CGC.** In this paper, we propose an extension of Megastore, called Megastore-CGC (“Megastore with Cross-Group Consistency”), which also provides validation across entity groups.

Megastore-CGC is based on the key observation that, in Megastore, a site replicating a set of entity groups participates in all updates on these entity groups. Therefore, this site implicitly has an ordering on these updates. Making this ordering explicit makes it possible to validate the transactions to ensure that only transactions with a consistent view across multiple entity groups are allowed to commit.

A significant advantage of making such an implicit ordering explicit is that validation information can be piggybacked onto Megastore’s coordination messages, which implies that:

- Megastore’s performance and scalability are preserved, since no additional messages or waiting is introduced.
- Transactions not requiring validation across entity groups have the same fault tolerance as in Megastore.

**Specification and Validation Methodology.** Fault-tolerant data management protocols for replicated data stores are highly complex artifacts, yet it is crucial to establish their performance and correctness.

Complex protocols are typically described using a combination of English prose and pseudo-code. For prototyping and performance estimation purposes, a prototype of the protocol is usually implemented in a programming language like Java, or using some simulation tool. Finally, the correctness of the protocol is “proved by hand.” This methodology has a number of disadvantages, including:

1. The prose + pseudo-code description is ambiguous and imprecise, and does not make explicit critical assumptions (or lack thereof).
2. A Java prototype is problematic for two reasons: (1) it is an additional artifact beyond the specification of the protocol; and (2) we must somehow ensure that this Java prototype is consistent with the specification.
3. Proofs “by hand” tend to be error-prone. Indeed, absent a precise and unambiguous definition of the protocol, there is in principle nothing that can be formally proved. Furthermore, the size and complexity of the systems that are the topic of this paper make the possibility of a serious “hand proof” quite futile anyways.

We therefore advocate a formal-methods-based approach to the specification, performance estimation, and correctness analysis of replica management protocols. One obvious advantage is that a formal specification defines a mathematical model of the protocol that is precise, unambiguous, and where implicit assumptions are made explicit. Furthermore, a mathematical model of the system enables rigorous mathematical analysis of its correctness. However, the specification and validation of a system of the size and complexity that we develop in this paper presents a number of challenges for formal methods, including:

1. The formal language should be easy and intuitive to allow protocol developers—who may have limited (or no) background in formal methods—to define their protocols using the formal modeling language.
2. The need to analyze both performance and correctness.
3. Typically, more restricted formal modeling languages ensure decidability of important properties, and hence support fully automatic formal analyses, whereas more expressive languages (often based on higher-order logics) support interactive theorem proving, which may require highly nontrivial user interaction during the verification process.

In this paper, we advocate using the rewriting-logic-based Real-Time Maude formal specification language and analysis tool [13] for the modeling and analysis of replica management protocols for the following reasons:

1. The Real-Time Maude specification language is object-oriented and fairly easy and intuitive, making the barrier to use the language acceptable for researchers with
limited/no formal methods background. Furthermore, previous experience shows that network engineers can easily understand Real-Time Maude specifications without formal methods background [14].

2. Real-Time Maude specifications are executable, and therefore, provides for free a prototype that can be executed for immediate feedback.

3. Real-Time Maude specifications can be subjected to Monte Carlo simulations for performance estimation.

4. Real-Time Maude is fairly expressive, allowing us to specify Megastore-CGC at an appropriate level of abstraction with reasonable effort.

5. Real-Time Maude provides a range of fully automated formal analyses methods. For example, temporal logic model checking investigates whether all possible system behaviors (from a given system configuration) satisfy desired properties. Such model checking can be regarded both as automatic verification, and as a method to find “corner case” bugs that are easily missed even during very extensive testing of the system.¹

In this paper, we formalize Megastore-CGC in Real-Time Maude. We then use Real-Time Maude simulations to compare the performance with our model of Megastore. Finally, we use automated model checking to verify a number of key properties of our protocol. One main benefit of Real-Time Maude is that it enables a “test-driven” development cycle where new ideas and features can be tested in a large number of scenarios in very short time using both simulation and model checking. We have also experienced that, in the presence of failures, anticipating all possible behaviors is impossible. This experience is shared by Google’s Megastore team which have implemented a pseudo-random test framework for this purpose, and report that “Through running thousands of simulated hours of operation each night, the tests have found many surprising problems” [1]. Compared to such a framework, Real-Time Maude model checking analyzes not only a set of pseudo-random behaviors, but covers all possible behaviors. Given that Google’s test framework tests the real implementation, it could not be replaced by model checking a Real-Time Maude model, but we believe that, especially in the early stages of protocol development, using Real-Time Maude provides an effective way to quickly and easily analyze the consequences of different design choices.

2. PRELIMINARIES

2.1 Megastore

Megastore [1] is a data store system with transaction support developed and run by Google. Megastore provides very high availability through replication across data centers, and is used both within Google’s own cloud infrastructure and by Google’s customers through the application platform AppEngine. Megastore handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of data across many global data centers [1].

In Megastore, data are stored as entities, each entity being a set of key-value pairs. A transaction is a sequence of read and write operations on entities, followed by a commit request. Megastore’s approach to transactional consistency is to partition data into fairly small units, denoted entity groups, and maintain a replicated transaction log for each entity group. Serializability within each entity group is ensured, since, at any given time, only one transaction is allowed to update the log.

Clients can issue transaction requests to any site replicating the relevant data, and updates are propagated to the other replicas before the transaction commits.

Megastore works as follows. Initially, all operations in a transaction are executed locally at the receiving site. When a commit request is issued, a coordination procedure among the sites is used to decide whether or not the transaction is valid and can be committed. If not, usually due to some concurrent update of the same data, the transaction is aborted.

Entities are versioned, and Megastore provides reads with different levels of consistency. We focus on current reads, which give the most recent version. A transaction updating an entity must perform a current read before writing.

Each site has a coordinator, which is always informed about whether the local replica is up-to-date. A current read is executed locally if and only if granted by the local coordinator. Otherwise, a majority read is required.

A transaction t accessing an entity group eg reads entities in eg at a given log position lp. t’s updates are buffered during transaction execution. When all operations of t are completed, the originating site of t prepares a log entry for eg containing t’s updates and runs Paxos [11] to request that this log entry becomes entry lp + 1 in the replicated log.

Paxos is a generic consensus protocol for distributed systems which consists of the following phases:

1. Agree on a leader.

2. The leader proposes a value to the other processes.

3. When the proposed value is acknowledged by a majority of the processes, the leader informs all participants about the decision.

In the presence of failures, this may be insufficient to reach consensus, in which case a new round is initiated where another process becomes the leader.

Megastore diverges from Paxos by (i) waiting for accept (or invalidation of the coordinator) from all replicating sites before allowing a transaction to commit, and (ii) including in each log entry the Paxos leader for the next log entry. Phase 1 is therefore replaced by a request from the originating site directly to the leader. If multiple sites request different log entries for the same log position, Paxos ensures that only one is elected, and the others are aborted.

¹Since most properties of Real-Time Maude specification in general are undecidable, there is no guarantee that the automatic analysis will terminate.
After a successful Paxos round, 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.

### 2.2 Real-Time Maude

Real-Time Maude [13] is an object-oriented formal modeling language and high-performance simulation and model checking tool for distributed real-time systems. The modeling formalism is expressive yet simple and intuitive, which together should allow developers with limited formal methods experience to model very complex real-time systems.

In short, an algebraic equational specification defines the data types and the necessary functions in a “functional programming style,” and rewrite rules \( t \rightarrow t' \) if \( \text{cond} \) define local transitions from state \( t \) to state \( t' \). A specification may exhibit nondeterministic behaviors, since (i) a rewrite rule might be applied to different parts of the state, and/or (ii) different rewrite rules might be applied to the same state.

Real-Time Maude specifications are executable, and the tool provides a variety of formal analysis methods. Simulation, that simulates one possible behavior of the system, is useful for quick prototyping and performance estimation. Reachability analysis and temporal logic model checking automatically check whether all possible system behaviors from an initial state satisfy a desired requirement.

**Specification.** A Real-Time Maude module specifies a real-time rewrite theory [13] \((\Sigma, E \cup A, IR, TR)\), where:

- \( \Sigma \) is an algebraic signature; that is, a set of declarations of sorts, subsorts, and function symbols.

- \((\Sigma, E \cup A)\) is a membership equational logic theory [12], with \( E \) a set of possibly conditional equations, and \( A \) a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed modulo the axioms \( A \). \((\Sigma, E \cup A)\) specifies the system’s state space as an algebraic data type.

- \( IR \) is a collection of labeled conditional rewrite rules specifying the system’s local transitions, each of which has the form \( \exists l : t \rightarrow t' \) if \( \text{cond} \), where \( l \) is a label. Such a rule specifies an instantaneous transition from an instance of \( t \) to the corresponding instance of \( t' \), provided the condition holds.

- \( TR \) is a set of tick rules \( l : \{t\} \rightarrow \{t'\} \) if \( \text{cond} \) that advance time in the entire state \( t \) by \( \text{time units} \).

Equations and rewrite rules are introduced with, respectively, keywords \texttt{eq}, or \texttt{ceq} for conditional equations, and \texttt{rl} and \texttt{crl}. The mathematical variables in such statements are declared with the keywords \texttt{var} and \texttt{vars}.

A declaration \texttt{class C | att1 : s1, ..., attn : sn} declares a class \( C \) with attributes \( att1 \) to \( attn \) of sorts \( s1 \) to \( sn \). An object of class \( C \) in a given state is represented as a term \( o : C | att1 : val1, ..., attn : valn \) > of sort \texttt{Object}, where \( o \), of sort \texttt{Obj}, is the object’s identifier, and where \( val1 \) to \( valn \) are the current values of the attributes \( att1 \) to \( attn \). A message is a term of sort \texttt{Msg}.

---

2 An equational condition \( u_i = v_i \) can also be a matching equation, written \( u_i : = v_i \), which instantiates the variables in \( u_i \) to the values that make \( u_i = v_i \) hold, if any.

---

The state is a term of sort \texttt{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. For example, the rewrite rule

\[
\text{rl} [1] : \quad m(0, w) \\
\quad \rightarrow \quad < 0 : C \mid a1 : x, a2 : O', a3 : z > \\
\quad \rightarrow \quad < 0 : C \mid a1 : x + w, a2 : O', a3 : z > \\
\quad \rightarrow \quad \text{dly}(m'(0', x), z).
\]

defines a family of transitions in which a message \( m \), with parameters \( 0 \) and \( w \), is read and consumed by an object \( 0 \) of class \( C \), the attribute \( a1 \) of object \( 0 \) is changed to \( x + w \), and a new message \( \text{dly}(m'(0', x), z) \) is generated; this message will become the “ripe” message \( m'(0', x) \) after \( z \) time units. Attributes whose values do not change and do not affect the next state of other attributes or messages, such as \( a3 \), need not be mentioned in a rule. Attributes that are unchanged, such as \( a2 \), can be omitted from right-hand sides of rules.

**Formal Analysis.** Real-Time Maude’s \texttt{timed fair rewrite} command simulates one of the many possible system behaviors from the initial state by rewriting the initial state up to a certain duration. It is written with syntax

\[
\text{tfrew} \ t \ \text{in time} \ \text{<} \ \text{timeLimit} \ \text{.}
\]

where \( t \) is the term to be rewritten (“the initial state”).

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

\[
\texttt{ceq} \ \text{statePattern} \ \text{=} \ \text{prop} = \ a \ \text{if cond}
\]

for \( a \) a term of sort \texttt{Bool}, which defines \texttt{prop} to hold in all states \( t \) where \( t \ \text{=} \ \text{prop} \) evaluates to \texttt{true}. A temporal logic \texttt{formula} is constructed by state propositions and temporal logic operators such as \texttt{True}, \texttt{False}, “)” (negation), \( \&\), \( \lor\), \( \ightarrow\) (implication), \( \square \) (“always”), and \( \texttt{Until} \) (“eventually”), and \( \texttt{U} \) (“until”). The unbounded model checking command

\[
\text{mc} \ \ t \ \text{=} \ u \ \text{formula} \ .
\]

checks whether the temporal logic formula \texttt{formula} holds in all behaviors starting from the initial state \( t \). If the reachable state space is infinite, \texttt{time-bounded LTL} model checking, in which each behavior is only analyzed up to a given time bound, can be used to ensure termination of the analysis.

### 3. MEGASTORE-CGC: CONSISTENCY ACROSS ENTITY GROUPS

This section describes Megastore-CGC, an extension to Megastore which provides consistency also for transactions accessing multiple entity groups.

In Megastore, the data is a set \( E \) of entities replicated across a set \( S \) of sites. \( E \) is partitioned into a set \( \text{EG} = \{eq_1, ..., eq_n\} \) of non-empty entity groups. A function \( R : S \rightarrow \mathcal{P}(\text{EG}) \) assigns to a site the entity groups it replicates.

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

**Example 1.** In the example in Section 1, constraint (2) requires that there is always a given number of available employees with a certain expertise. As explained in Section 1, different employees should belong to different entity groups. Since Megastore only provides consistency within single entity groups, using Megastore may lead to violation of constraint (2). In Megastore-CGC, grouping the entity groups for all employees with a given expertise into the same ordering class allows us to ensure also constraint (2).

In Megastore, a transaction \( t \) first executes a sequence of read and write operations at one site, \( t's \) originating site. Then, before commit, all sites replicating entities updated by \( t \) must agree to accept \( t's \) updates. Conflicts between concurrent updates in the same entity group are detected and resolved by aborting all but one conflicting transaction.

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

**Example 2.** Let the status of heart surgeon \( h \) at time slot \( \tau \) be represented by the entity \( e_h \), which is part of the entity group \( e_h \), representing all time slots of \( h \) at the given day.

Let \( t \) be a transaction, initiated at site \( s_t \), that wants to book \( h \). Since there must always be at least one heart surgeon available, \( t \) also reads the status of the other heart surgeons at time \( \tau \). These entities belong to different entity groups which, together with \( e_h \), all belong to the ordering class \( HS \). Finally, \( t \) completes by changing the availability status of \( h \) to booked, if possible.

There is a risk of inconsistency if some concurrent transaction \( t' \), executing at some other site \( s_t' \), attempts to book another heart surgeon \( h' \) at time slot \( \tau \). Let \( h' \) be the only other available heart surgeon at \( \tau \), and let \( t' \) issue a booking for entity \( h'_o \) at the same time \( t \) is executing. \( h'_o \) belongs to entity group \( e_{h'_o} \), with \( e_h \neq e_{h'_o} \). In Megastore, constraint (2) could be violated in the following scenario:

1. \( t \) reads the value of \( h \) and \( h'_o \) at \( s_t \).
2. \( t' \) reads the value of \( h \) and \( h'_o \) at \( s_t' \).
3. \( t \) books \( h \). This update is distributed by \( s_t \) and applied at all sites replicating \( h \), including \( s_t' \).
4. \( t' \) books \( h'_o \). This update is distributed by \( s_t' \) and applied at all replicating sites, including \( s_t \).

This execution is clearly not serializable, and both heart surgeons are booked. This scenario is possible in Megastore, which enforces consistency within one entity group.

This execution will not occur in Megastore-CGC, whose commit protocol includes ordering and validation of concurrent updates to entity groups in the ordering class \( HS \). The ordering site of \( HS \) ensures that \( t \) and \( t' \) are ordered and then validated 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, leaving one heart surgeon available.

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 – as explained in detail in Sections 3.1 and 3.2 – 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.

### 3.1 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 uses a replicated transaction log for each entity group, and updates these logs using Paxos. Megastore-CGC maintains the following additional information:

- A mapping \( os : OC \rightarrow S \), which assigns to each ordering class \( oc \) its ordering site \( s \) satisfying \( oc \subseteq R(s) \).
- A function \( ol : OC \rightarrow Orderlist \), 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.

As mentioned above, any Megastore site replicating all entity groups in an ordering class \( oc \) implicitly orders all transactions updating these entity groups. We can select any of these sites as the ordering site for \( oc \). The ordering list \( ol(oc) \) makes this implicit order explicit at the ordering site of \( oc \). The list is replicated at all sites, with each site maintaining a projection of \( ol(oc) \) containing updates to locally replicated entity groups only.

The mapping \( os \) is stored as a special entity group \( e_{os} \) replicated at all sites. This ensures a consistent view among all participating sites.

The entity groups accessed by transaction \( t \) determine the ordering class of \( t \). When a transaction \( t \) with ordering class \( oc \) commits, an entry for \( t \) is appended to the list \( ol(oc) \) by the ordering site of \( os(oc) \). This represents the ordering of \( t \) in \( oc \). When \( t \) is ordered within \( oc \), \( t \) can be validated: its execution is valid if and only if all its read operations have seen the most recent update according to \( ol(oc) \).

The challenge is to provide ordering and validation without reducing performance, scalability, or fault tolerance. Our approach is to avoid introducing new messages before commit, but instead piggyback the necessary information onto the coordination messages in Megastore’s commit protocol.

Recall from Section 2.1 that the Megastore commit protocol can be summarized by the following steps:

1. Request accept from the leader for this log position.
2. The leader checks whether there are conflicting updates for this log position. If no such conflicts exist, the request is accepted. Otherwise, the leader signals a conflict and the transaction will be aborted.
3. After leader-accept, request all participating processes to accept the transaction.

4. Upon receiving the accept request, each participant acknowledges the transaction.

5. When the transaction has been acknowledged by all processes, the originating site requests all participants to apply the update.

Let \( t \) be a transaction with ordering class \( oc \). Megastore-CGC then extends Megastore’s commit protocol as follows:

- \( t \) is ordered once the ordering site \( os(oc) \) accepts \( t \). If \( os(oc) \) is the leader for this log position, this occurs at Step 2. Otherwise, it occurs at Step 4.

- After ordering, \( os(oc) \) validates \( t \), using the read set of \( t \) as input. The read set is included in the accept-request from \( t \)’s initiator, and contains the id of all entities read by \( t \), together with the version (represented by the log position) read by \( t \). The validation procedure works as follows: \( t \) is allowed to commit if and only if, for each element of the read set, \( t \) has seen the most recent version according to \( ol(oc) \). This provides consistency as follows: For any pair of transactions in a read-write conflict (i.e., where one is reading and the other is writing the same entity), this ensures that unless the conflicting operations occur in transaction order, i.e., according to \( ol(oc) \), one of the transactions is aborted. This is sufficient to verify that the serialization graph [20] for any Megastore-CGC schedule is acyclic, as long as all transactions access entity groups within one ordering class only.\(^3\)

- If validation at \( os(oc) \) is successful, the originating site 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 a rollback-step, requesting all participating sites to abort \( t \).

The steps for committing a transaction \( t \), which reads a set of entity groups \( EG \) and updates an entity group \( eg \in EG \), are summarized in Table 1. All entity groups in \( EG \) belong to an ordering class \( oc \). \( R_{eg} \) denotes all sites replicating \( eg \).

### 3.2 Failure Handling in Megastore-CGC

This section explains how Megastore-CGC deals with site failures and/or message losses. The goal is to complete as many transactions as possible while ensuring consistent execution even in the presence of failures. An additional fault-tolerance challenge in Megastore-CGC is that the ordering of transactions must be consistent even if the ordering site fails and/or messages containing ordering information are lost. The key ideas in our approach to fault tolerance 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.

- We choose a new ordering site if the current ordering site is suspected to be unavailable.

Megastore-CGC extends the failure handling features of Megastore, including electing a new leader if the leader has failed, and ensuring commit of a transaction as long as a majority of sites have acknowledged the update.

Some characteristics of Megastore-CGC are:

- Transactions accessing only one entity group have the same fault tolerance and performance as in Megastore. This means that even during failures, transactions are completed with the same number of messages as in Megastore. If ordering fails, we provide features to order such transactions later, and piggyback this ordering onto the next successful transaction. Therefore, reinstating a complete transaction order is provided with no additional messages being transmitted.

- Transactions accessing multiple entity groups are either successfully ordered or aborted. This ensures cross-group-consistency in case of failure.

- For an ordering class, ordering will fail if the ordering site becomes unavailable. To minimize the time to recovery, an ordering site failover procedure is provided.

A new ordering site is elected using a special purpose update transaction, thus ensuring fault tolerance and consistency through Paxos.

### Ordering Failure

We first discuss how Megastore-CGC deals with situations where a transaction \( t \) is acknowledged by a majority of sites, without being ordered (and validated) by the ordering site. This can happen for several reasons:

1. The ordering site is down (or recovering from failure, and hence unable to safely order new transactions).

2. The accept request from the initiator to the ordering site was lost.

3. The acknowledgment from the ordering site to the initiator was lost.

4. The initiator site crashed after sending the accept request, and this transaction was completed by some other site (this is a feature provided by Paxos).

In this scenario, the apply message for \( t \) is sent without the ordering information for \( t \). Since this implies that \( t \) has not been validated, 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.

### Ordering Site Failover

In case of failure of the ordering site, 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, \( t \)’s initiator site \( s_i \) initiates an ordering site failover for order class \( oc \).\(^3\)
### 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/mgc/](http://folk.uio.no/jongr/mgc/).

**Classes.** 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:

```plaintext
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 \((t, e, l p)\) for each ordering class \(oc\), representing \(ol(oc)\), where \(l p\) denotes the log position of \(t\)'s update in the transaction log for entity group \(eg\); and `awaitingOrder` is a set of entries on the form \((oc, t, e, l p)\), used during failures to register transactions requiring ordering later.

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

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

The attribute `entitiesState` describes the available versions of each entity in the entity group. `transactionLog` denotes the local copy of the replicated transaction log. A log entry \((t, l p, s, ol)\) contains the identity \(t\) of the originating transaction, the log position \(l p\), the leader site \(s\) for the next log entry, and the list \(ol\) of write operations executed by \(t\). The attribute `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 object instances of the class:

```plaintext
class Transaction | operations : OperationList,
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Site(s)</th>
<th>Megastore</th>
<th>CGC extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(s_t)</td>
<td>(t)'s operations are executed at site (s_t).</td>
<td>If (s_t = os(oc)), include (t)'s read set and request ordering and validation from (s_t).</td>
</tr>
<tr>
<td>1</td>
<td>(s_t)</td>
<td>Send an <code>acceptLeader</code> request to the leader (s_t) for the current log position.</td>
<td>If (s_t = 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).</td>
</tr>
<tr>
<td>2</td>
<td>(s_t)</td>
<td>Receive <code>acceptLeader</code> request. If there are no conflicting updates within (eg), send accept to (s_t). Otherwise, request (s_t) to abort (t).</td>
<td>If (s_t = os(oc)) and (s_t = os(oc)), order and validate (t).</td>
</tr>
<tr>
<td>3</td>
<td>(s_t)</td>
<td>Receive response from (s_t). If (s_t) requests abort, (t) is aborted. Otherwise, multicast an <code>accept</code> request for (t) to all sites replicating entity group (eg), excluding (s_t) and (s_t).</td>
<td>If (s_t = os(oc)) and (s_t = os(oc)), order and validate (t).</td>
</tr>
<tr>
<td>4a</td>
<td>(R_{eg} \setminus {os(oc), s_t, s_t})</td>
<td>Receive and store the <code>accept</code> request, send acknowledgment to (s_t).</td>
<td>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.</td>
</tr>
<tr>
<td>4b</td>
<td>(os(oc), s_t)</td>
<td>if (os(oc) \neq s_t \land os(oc) \neq s_t)</td>
<td>Receive and store the <code>accept</code> request, send acknowledgment to (s_t).</td>
</tr>
<tr>
<td>5</td>
<td>(s_t)</td>
<td>Multicast <code>apply</code> message containing (t)'s updates.</td>
<td>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)).</td>
</tr>
<tr>
<td>6</td>
<td>(R_{eg})</td>
<td>Apply (t)'s updates to local transaction log and replicated entity store.</td>
<td>If the <code>apply</code> message contains (ol(oc)), update the local copy of (ol(oc)).</td>
</tr>
</tbody>
</table>

**Table 1: Transaction execution in Megastore-CGC without failure handling.**

- \(s_t\) 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 \(eg\), which for each ordering class \(oc\) contains the current ordering site.
- 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`. Their ordering is also included in response to the next `accept` message received for an update \(t'\) within \(oc\), and propagated to the other sites as part of the the `apply` message of \(t'\).

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 \((t, e, l p)\) for each ordering class \(oc\), representing \(ol(oc)\), where \(l p\) denotes the log position of \(t\)'s update in the transaction log for entity group \(eg\); and `awaitingOrder` is a set of entries on the form \((oc, t, e, l p)\), used during failures to register transactions requiring ordering later.

Each site's copy of an entity group is modeled as an object instance of the following class:
We assume that the sites are connected FIFO delivery between the same pair of nodes. Nodes can therefore vary significantly, and we do not assume by a wide-area network. The network delays between two nodes can be significant, and we do not assume.

We assume that the sites are connected FIFO delivery between the same pair of nodes. Nodes can therefore vary significantly, and we do not assume by a wide-area network. The network delays between two nodes can be significant, and we do not assume.

The attributes of entity groups contains the remaining operations in the transaction; reads stores the value 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.

Communication. We assume that the sites are connected by a wide-area network. The network delays between two nodes can therefore vary significantly, and we do not assume FIFO delivery between the same pair of nodes.

The set of possible delays between $s_1$ and $s_2$ is given by possibleMsgDelays($s_1$, $s_2$) as a "ripened" set of time values. A "ripe" message has the form

```
msg mc from sender to receiver
```

where mc is the message content. We can nondeterministically select any delay from the set possibleMsgDelays($s_1$, $s_2$) by sending messages using rewrite rules of the form

```
var T : Time . var TS : TimeSet.
crl [sendMsgAnd...] :
    < SID : Site | ... > ...
=>
    < SID : Site | ... > ...
dly(msg mc from SID to SID', T)
if ... /\ T ; TS := possibleMsgDelays(SID,SID')
```

This is a valuable feature: it allows us to define an initial state (with a set of transactions and a list of possible message delays), and then through model checking inspect all possible executions to check a large number of different message orderings and protocol states for consistency.

Dynamic Behavior. The dynamic behavior of Megastore-CGC is defined by 72 rewrite rules; we present one of them in Figure 1. This rule formalizes Step 4b in Table 1, with additional failure handling, when a site SID receives an accept request for transaction TID from TID's initiator SENDER.

In this rule, SID receives a message acceptAllReq(TID, EG, (TID LP SENDER OL), READS, PNUM) from the site SENDER. TID is the transaction identifier and EG is the entity group updated by TID. The tuple (TID LP SENDER OL) is a candidate log entry for log position LP in the replicated log for EG, with OL the list of updates executed by TID. READS is the read set of TID, i.e., all entities and entity versions read by TID. PNUM is the proposal number, used by Paxos to distinguish competing requests for the same log position.

Since the message contains the read set READS of TID, we know from Step 3 in Table 1 that SID is supposed to be the ordering site for TID, and this request asks SID to order and validate TID. However, since messages can be delayed, SID must verify that it is still the ordering site before distributing TID’s order. Therefore, the rule first extracts the ordering class OCID for EG; it then extracts the current ordering site ORDERSITE for OCID from the special entity group egos, which is replicated as an entity group in SID.

Next, the rule checks whether SID is can order TID. It verifies that SID is still the ordering site (SID == ORDERSITE) and that no updates are missing due to failures (isUpToDate(...)). If both conditions hold, SID can order and validate TID.

If there are updates awaiting ordering (due to previous failures), these are first applied to the local order list with outdated log OP. The resulting list is assigned to the variable OLIST. Then, TID is ordered and the resulting order list is stored in NEW-OLIST.

The transaction is VALID if either TID has only read entities from one entity group in (IN-SINGLE-EG is true), or if SID can order TID and its read set is consistent (i.e., it has seen the most recent version of each entity according to $ocid$).

The attributes of SID are updated as follows:

- If TID is ordered and validated, the egOrderings attribute, denoting SID's local order lists, is updated to the new value NEW-OLIST.
- The proposals attribute of entity group EG, which contains SID's current registry of Paxos-interactions for...
is updated if \( \text{TID} \) is valid: all obsolete entries are removed, and an accepted-entry for \( \text{TID} \) is added.

- If \( \text{SID} \) was able to order \( \text{TID} \), the awaitingOrder set is reset (since any updates previously awaiting ordering will now be included in new-OList).

Finally, the response to \( \text{sender} \) depends on the validation outcome. If \( \text{TID} \) is both validated and ordered, an acceptAll-\( \text{reply} \) message is sent, containing the updated order. Otherwise, the same message is sent without ordering information.

5. PERFORMANCE ESTIMATION

This section shows how randomization of Real-Time Maude simulations can estimate the performance of Megastore-CGC by rewriting a given initial system configuration. We measure the following performance parameters:

- 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 model of) Megastore-CGC with that of (our Real-Time Maude model of) Megastore, both when transactions read multiple entity groups (requiring cross-group validation), and when they only read one entity group. Given the right system parameters, Real-Time Maude simulations should provide realistic performance estimates. For example, it is shown in [15] that Real-Time Maude simulations of wireless sensor networks give as good performance estimates as dedicated simulation tools.

The main parameters of our simulations 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 very easily change these parameters by modifying the initial state shown in Figure 2. We use an example scenario with three sites, four entities, two entity groups, and a set of different 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 for reads.

For Megastore-CGC, we use one ordering class (containing both entity groups). Our scenario is a "hot spot" setting where the chance of conflicting transactions is high. We assume two sites, Site 1 and Site 2, located in the same area, and a third site (RSite) at a more remote location. The probability distribution for the network delays is as follows:

<table>
<thead>
<tr>
<th>Site 1 ↔ Site 2</th>
<th>30%</th>
<th>30%</th>
<th>30%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1 ↔ RSite</td>
<td>30%</td>
<td>35%</td>
<td>40%</td>
<td>100%</td>
</tr>
<tr>
<td>Site 2 ↔ RSite</td>
<td>30%</td>
<td>35%</td>
<td>40%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Transaction requests are generated at each site by a transaction generator that creates transactions of different types randomly, according to the following frequency distribution:

Since rewriting only returns the final state, we add "record" objects that record events during the simulation, using techniques in [15]. The initial state \( \text{initState} \), shown in Figure 2, is then a multiset containing: one Site object for each site; one NetworkDelays object containing the (possibly dynamically changing) network delay distributions; one Random object containing the seed used to randomly select a network delay when a message is sent; one SiteStatistics object for each site to record relevant statistics during simulation; and a PoissonTransGen object for each site, which generates transactions randomly according to the given distribution.

We can then simulate the system up to 1,000,000 ms by giving the Real-Time Maude command:

\[
\text{tfrew \ initState(10) in time <= 1000000 .}
\]

which returns the term^4

\[
\{ < \text{state}(\text{Site}) : \text{SiteStatistics} | \begin{align*}
\text{avgLatency} & : 94579/631, \text{commitCount} : 631, \\
\text{conflictAborts} & : 171, \text{validationAborts} : 10, \ldots > \ldots \}
\end{align*}
\]

in 145,957 ms cpu time on a Pentium Intel Core i7 2.6 GHz.

We have also run these experiments on our model of Megastore. We have run experiments with different system parameters, 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).

<table>
<thead>
<tr>
<th>Site</th>
<th>Comm.</th>
<th>Abs.</th>
<th>Avg.lat</th>
<th>Val.abs</th>
<th>Avg.lat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>632</td>
<td>152</td>
<td>126</td>
<td>660</td>
<td>144</td>
</tr>
<tr>
<td>Site 2</td>
<td>704</td>
<td>100</td>
<td>118</td>
<td>674</td>
<td>115</td>
</tr>
<tr>
<td>RSite</td>
<td>640</td>
<td>172</td>
<td>151</td>
<td>631</td>
<td>171</td>
</tr>
</tbody>
</table>

Since Megastore-CGC provides validation without additional messages, the average latency is virtually the same as in Megastore. Some transactions accessing multiple entity groups (book-H1-A and book-H2-A) could see an inconsistent read set. In Megastore-CGC, this shows up as validation aborts, whereas they are committed by Megastore.

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:

<table>
<thead>
<tr>
<th>Site</th>
<th>Comm.</th>
<th>Abs.</th>
<th>Avg.lat</th>
<th>Val.abs</th>
<th>Avg.lat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>684</td>
<td>120</td>
<td>122</td>
<td>679</td>
<td>125</td>
</tr>
<tr>
<td>RSite</td>
<td>674</td>
<td>138</td>
<td>132</td>
<td>677</td>
<td>135</td>
</tr>
<tr>
<td>Site 2</td>
<td>693</td>
<td>111</td>
<td>110</td>
<td>691</td>
<td>113</td>
</tr>
</tbody>
</table>

The small differences can be explained by nondeterminism; e.g., the set of rewrite rules is different in the two models.

^4 Parts of the term are replaced by ‘...’
The following table shows the result of simulating Megastore-CGC in a challenging scenario, where 2% of the messages are lost, and where one site (Site 2) repeatedly fails for half a second, with a mean time between failures of 200 seconds:

<table>
<thead>
<tr>
<th>Megastore-CGC</th>
<th>Comm.</th>
<th>Abs.</th>
<th>Val.abs</th>
<th>Avg.lat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>658</td>
<td>154</td>
<td>9</td>
<td>185</td>
</tr>
<tr>
<td>RSite</td>
<td>497</td>
<td>162</td>
<td>163</td>
<td>209</td>
</tr>
<tr>
<td>Site 2</td>
<td>469</td>
<td>160</td>
<td>189</td>
<td>169</td>
</tr>
</tbody>
</table>

If the ordering site cannot verify that it has the most recent state (e.g., due to lost messages), transactions requiring cross-entity group validation are preventively aborted, explaining the number of validation aborts in this setting.

6. MODEL CHECKING VERIFICATION

It must be validated that Megastore-CGC is working in the desired way before deploying it in a real cloud system. However, Megastore-CGC is very complex, and any “hand proof” of its correctness would be too superficial and error-prone to instill much confidence in its correctness. Computer-aided verification could prove correctness of Megastore-CGC, but would require significant non-trivial user interaction.

We instead use model checking [3] to analyze our model. Model checking automatically explores all possible behaviors that can happen nondeterministically from a given initial system configuration; such analysis therefore does not verify that the system is correct for all possible initial configurations. In addition to verifying desired properties, model checking is invaluable during the design process, and helped us discover many subtle “corner case” bugs in (earlier versions of) Megastore-CGC that were not uncovered during extensive simulation. (For example, what happens if a site goes down, is invalidated, misses some updates, and then comes back and becomes the ordering site again before another site has been selected as the ordering site?)

We analyze the original nondeterministic model (not the randomized one used for performance estimation). For the model checking analysis to terminate, we only analyze scenarios with a finite number of transactions. Since the reachable state space quickly becomes very large due to the large number of possible concurrent executions, we have also restricted the different message delays, transaction start times, site and communication failures, etc.

With a finite number of messages, 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, again unless a coordinator is invalidated.
4. The execution was serializable; i.e., it gives the same result as some execution where the transactions are executed one after the other.

This correctness property can be formalized as the temporal logic formula (which we denote $\Phi$ below)

\[
<> [] (\text{allFinished} \land \text{entityGroupsEqualOrInvalid} \land \text{transLogsEqualOrInvalid} \land \text{isSerializable})
\]

where: $\text{allFinished}$ is a state proposition that is true in a state if all transactions have finished; $\text{entityGroupsEqualOrInvalid}$ 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 $\text{transLogsEqualOrInvalid}$ 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:

\[
\text{op transLogsEqualOrInvalid} := \text{Prop} \text{[ctor]}.\text{eq} \text{[REST]} \\
\text{eq (REST)} \quad \text{eq (SYSTEM)} := \text{transLogsEqualOrInvalid} \land \text{isSerializable}
\]

This definition first characterizes the states where $\text{transLogsEqualOrInvalid}$ does not hold, namely, the states where there are two sites with valid coordinators and with some entity group $E_G$ with different values. The last equation, with the
**Table 2: Example: Model checking setup**

<table>
<thead>
<tr>
<th>Site</th>
<th>Transaction</th>
<th>Operations</th>
<th>Start time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site 1</td>
<td>update-H1-A</td>
<td>read H1-A; write(H1-A, Avail1)</td>
<td>150</td>
</tr>
<tr>
<td>RSite</td>
<td>update-H2-A</td>
<td>read H2-A; write(H2-A, Avail1)</td>
<td>150</td>
</tr>
<tr>
<td>Site 2</td>
<td>update-H2-A</td>
<td>read H2-A; write(H2-A, Avail1)</td>
<td>150</td>
</tr>
<tr>
<td>RSite</td>
<td>book-H2-A</td>
<td>read H1-A; read H2-A; write(H2-A, Booked)</td>
<td>(180, 210)</td>
</tr>
<tr>
<td>Site 2</td>
<td>book-H1-A</td>
<td>read H2-A; read H1-A; write(H1-A, Booked)</td>
<td>(180, 210)</td>
</tr>
</tbody>
</table>

ouise ("otherwise") attribute, defines `transLogEqualOrInvalid` to be `true` for all other states. The other state propositions can be defined similarly, as explained in [9].

To analyze the serializability property, we use the technique in [9] and add an "observer" object to the state that stores the serialization graph resulting from the execution. The state proposition `isSerializable` is then `true` in a state if the serialization graph in the state does not contain cycles.

**Model Checking.** We have model checked the above temporal logic formula $\Phi$ 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. Three of the transactions have fixed start times at 150, while the two remaining transactions may nondeterministically start at either time 180 or time 210, i.e., we inject two conflicting transactions in the middle of an execution with three single-entity-group transactions. This setup is shown in 2.

As in Section 5, we use three sites and one ordering class containing both entity group H1 and entity group H2.

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

```
(mc init1 |=u $\Phi$ .)
```

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 the entire behavior invalidating $\Phi$ when the model checking fails; this was allowed us to easily identify the (often subtle) issues causing the problem.

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], Cassandra [10] and Google’s BigTable [2] 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, ElaTras [6], Spinnaker [17], and Calvin [19] 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, publicly available method for transactional consistency across partitions besides Megastore-CGC. Google’s Spanner [5] provides both high availability, scalability, and transactional consistency across partitions, but this approach is less generic since it demands a complex infrastructure involving GPS hardware and atomic clocks.

Despite the importance of transactional data stores, we have not seen any other work on formalizing and verifying such systems using formal verification tools. In [16] the authors assert the need for formal analysis of replication and concurrency control in transactional cloud data stores, and they present and analyze a prose-and-pseudo-code description of a concurrency control protocol based on Paxos. 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 in itself a challenging task, since Megastore is an internal system at Google that is publicly described only in a quite informal way in the paper [1]. In [9] we therefore develop and model check 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 [18], where the authors point out possible bottlenecks in a naive implementation of ZooKeeper for key distribution, and in [8], where the authors analyze denial-of-service prevention mechanisms.

**8. CONCLUDING REMARKS**

We have proposed Megastore-CGC as an extension of Megastore to also provide consistency for transactions that access multiple entity groups. The main observation behind our approach is that, in Megastore, sites replicating multiple partitions (entity groups) implicitly observe an ordering of updates across this set of partitions. We make this ordering explicit by defining ordering classes. An ordering class is a set of entity groups, with at least one site replicating all the entity groups in the set. One such site, the ordering site, maintains an ordering on all updates in the ordering class, and uses this ordering to validate transactions. 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 in-
dicate that the performance of Megastore-CGC is virtually the same as that of Megastore.

We believe that the Megastore-CGC approach could be applicable to other Paxos-based transactional data stores such as Spinnaker [17] and Calvin [19]. However, one distinguishing feature of Megastore is the quite strong assumption 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.

A main question when applying Megastore-CGC is how to partition entity groups into ordering classes. On the one hand, ordering classes should be as large as possible to support transactions reading many different entity groups. On the other hand, replicating a large number of entity groups at the same site(s) may be impractical, and reduces fault tolerance since failover of the ordering site is provided only if there are other available sites replicating all entity groups in the ordering class.

Designing and validating a sophisticated, fault tolerant protocol such as Megastore-CGC is very challenging. We use Real-Time Maude to give a precise, formal specification of Megastore-CGC. Real-Time Maude specifications are executable, which allows us both to simulate the system for quick prototyping and performance estimation, as well as to use model checking to automatically explore all possible behaviors from a given system configuration, both to verify properties, but also to find subtle “corner case” design errors. Model checking was very helpful during the design process, since it uncovered many subtle errors that were not found during extensive simulations.

9. REFERENCES

Part III
Appendices
Appendix A
Real-Time Maude Example

The following listing contains:

- The modules defined in the examples of Section 3.
- A sample initial state, defined as the equation `initState`.
- A sample model-checking setup for `initState`.
- A sample simulation setup for `initState`.

Assuming a working Maude environment, and Real-Time Maude installed in the directory `RTM32`, the model can be executed as is.

Listing A.1  `rtm_example.rtmaude`

```
load RTM32/real-time-maude
(tmod TIME-DOMAIN is pr NAT-TIME-DOMAIN-DEFAULT-INF .
endtm)

(tomod TIMED-BEHAVIOR is
  pr TIME-DOMAIN .
  var C: Configuration .
  vars NEC NEC': NECConfiguration .
  var T: Time .
  crl [tick]:
    {C} => {delta(C, mte(C))} in time mte(C) if mte(C) > 0 \ mte(C) /= INF .
  op delta: Configuration Time -> Configuration [format (r! o) frozen (1)] .
  eq delta(none, T) = none .
  eq delta(NEC, T) = delta(none, T) .
  op mte: Configuration -> TimeInf [format (r! o) frozen (1)] .
  eq mte(none) = INF .
  eq mte(NEC, T) = min(mte(none), mte(NEC')) .
endtom)

(mod SETUP is
  inc TIMED-BEHAVIOR .
  sorts Item ItemVal .
  sorts OpList Op Read Write .
  sort NoOp .
```

127
subsorts Read Write < Op < NoOp < OpList .

\begin{verbatim}
op read : Item -> Read [ctor] .
op write : Item ItemVal -> Write [ctor] .

vars PRED SUCC : OpList .

op isReadOnly? : OpList -> Bool .
eq isReadOnly?(PRED :: write(I:Item, IV:ItemVal) :: SUCC) = false .
eq isReadOnly?(OPLIST:OpList) = true [wise] .
\end{verbatim}

endm)

(tomod TRANSACTION is
  inc SETUP .

  var OPLIST : OpList .
  var T1 T2 : Time .
  var TID : Oid .

  class Trans | ops : OpList, nextop : Time .
  class UpdateTrans | bufferedWrites : OpList .
  subclass UpdateTrans < Trans .

  eq delta(< TID : Trans | ops : OPLIST, nextop : T1 , T2 >=
              < TID : Trans | ops : OPLIST, nextop : T1 minus T2 >).

endtm)

(tomod SITE is
  inc TRANSACTION .
  inc NAT .

  var OPLIST : OpList .
  var SID TID : Oid .
  var TRANS : Configuration .
  var T : Time .

  class Site |
    transactions : Configuration, available : Bool, numCompl : Nat .

  eq mte(< SID : Site | transactions : TRANS >) = mte(TRANS) .
  eq delta(< SID : Site | transactions : TRANS >, T) =
             < SID : Site | transactions : delta(TRANS, T) > .


  crl [receiveNewUpdateTrans]:
    newTrans(TID, OPLIST)
    | ( < SID : Site | transactions : TRANS, available : true >
    => [ < SID : Site | transactions : TRANS < TID : Trans | ops : OPLIST, nextop : 0 > >
    if not isReadOnly?(OPLIST) .

  rl [nextOperation]:
    < SID : Site |
      transactions : < TID : Trans | ops : OP :: OPLIST, nextop : 0 > TRANS
\end{verbatim}
A Real-Time Maude Example

\[
\begin{array}{l}
\text{A Real-Time Maude Example} \\
\begin{array}{l}
\text{> =)} \text{< SID : Site | transactions : < TID : Trans | ops : OPLIST, nextop : 10 > TRANS} \\
\end{array} \\
\text{endtom)} \\
\text{(tomod INIT is}} \\
\text{inc SITE.} \\
\text{ops t s : } \rightarrow \text{Oid. ops x y : } \rightarrow \text{Item. op v : } \rightarrow \text{ItemVal.} \\
\text{op initState : } \rightarrow \text{GlobalSystem.} \\
\text{eq initState = \{} \\
\text{< s : Site | transactions : none, available : true, numCompl : 0 >} \\
\text{newTrans(t, read(x) :: read(y) :: write(x, v))} \\
\text{\}}. \\
\text{endtom)} \\
\text{(tomod PROPS is}} \\
\text{inc INIT.} \\
\text{inc TIMED-MODEL-CHECKER.} \\
\text{vars SID TID : Oid.} \\
\text{var OPLIST : OpList.} \\
\text{var SYSTEM : Configuration.} \\
\text{op isComplete : } \rightarrow \text{Prop [ctor].} \\
\text{eq \{} \text{< SID : Site | transactions : < TID:Oid : Trans | ops : noOp > > SYSTEM} \\
\text{\} = isComplete = true.} \\
\text{endtom)} \\
\text{(mc initState \=} u \text{<> isComplete \})} \\
\text{(set tick def 1.)} \\
\text{(tfrew initState in time } \leq 30 \text{.)}
\end{array}
\]
Appendix B
Real-Time Maude Model of Megastore

Listing B.1 time_behavior.rtmaude

(tomod TIMED-BEHAVIOR is
  pr TIME-DOMAIN .
  var C : Configuration .
  vars NEC NEC' : NEConfiguration .
  var T : Time .
  crl [tick] : {C} => {delta(C, mte(C))} in time mte(C) if mte(C) > 0 /\ mte(C) /= INF .
  op delta : Configuration Time --> Configuration [format (r! o) frozen (1)] .
  eq delta(none, T) = none .
  eq delta(NEC NEC', T) = delta(NEC, T) delta(NEC', T) .
  op mte : Configuration --> TimeInf [format (r! o) frozen (1)] .
  eq mte(none) = INF .
  eq mte(NEC NEC') = min(mte(NEC), mte(NEC')) .
endtom)

Listing B.2 megastore_setup.rtmaude

(tomod MEGASTORE-SETUP is
  inc TIMED-BEHAVIOR .
  inc RANDOM .
  var SID : SiteId .
  var EGRS : EntityGroupReplicaSet .
  var LP : LogPosition .
  var TID : TransId .
  vars N N' : Nat .
  var SIL : SiteIdList .
  vars OID 0 0' : Oid .
  vars OS GIS OIS S1 OS2 : OidSet .
  var CES : EntGroupLogPosPairSet .
  var LP : LogPosition .
  var EG : EntityGroupId .
  var OL : OperationList .
  var LOCALTRANS : Configuration .

  *** (Mapping from entity group to log position ***)
  sorts EntGroupLogPosPair EntGroupLogPosPairSet DefEntGroupLogPosPair .
  subsort EntGroupLogPosPair < EntGroupLogPosPairSet .
op noEntGroupLogPosPair : \rightarrow \text{EntGroupLogPosPairSet} \langle \text{ctor} \rangle .

op eglp : \text{EntityGroupId} \text{LogPosition} \rightarrow \text{EntGroupLogPosPair} \langle \text{ctor} \rangle .

op _\_ : \text{EntGroupLogPosPairSet} \text{EntGroupLogPosPairSet} \rightarrow \text{EntGroupLogPosPairSet} \langle \text{ctor assoc id: noEntGroupLogPosPair} \rangle .

--- Used for coordinator
op invalidCstate : \text{EntityGroupId} \text{LogPosition} \rightarrow \text{EntGroupLogPosPair} \langle \text{ctor} \rangle .

op containsEntityGroupId : \text{EntityGroupId} \text{EntGroupLogPosPairSet} \rightarrow \text{Bool} .

eq containsEntityGroupId(EG, eglp(EG, LP) : CES) = true .

\text{eq containsEntityGroupId}(EG, CES) = \text{false [owise]} .

\text{*** (Sites) ***}

\text{sort SiteId} .

\text{subsort SiteId} < \text{Oid} .

\text{class Site} |
\text{coordinator : EntGroupLogPosPairSet} ,
\text{entityGroups : Configuration} ,
\text{localTransactions : Configuration} .

\text{*** (Transaction Log) ***}

\text{sorts LogPosition LogPositionList DefLogPosition} .

\text{subsort LogPosition} < \text{LogPositionList} .

\text{subsort LogPosition} < \text{DefLogPosition} .

op noLogPosition : \rightarrow \text{DefLogPosition} \langle \text{ctor} \rangle .

op emptyLPList : \rightarrow \text{LogPositionList} \langle \text{ctor} \rangle .

op lpos : \text{Nat} \rightarrow \text{LogPosition} \langle \text{ctor} \rangle .

op _\_ : \text{LogPositionList LogPositionList} \rightarrow \text{LogPositionList} \langle \text{ctor assoc id: emptyLPList} \rangle .

op min : \text{LogPosition DefLogPosition} \rightarrow \text{LogPosition} .

\text{eq min}(\text{lpos}(N), \text{lpos}(N')) = \text{if} (N <= N') \text{then} \text{lpos}(N) \text{else} \text{lpos}(N') \text{fi} .

\text{eq min}(\text{lpos}(N), \text{noLogPosition}) = \text{lpos}(N) .

\text{sorts LogEntry LogEntryList} .

\text{subsort LogEntry} < \text{LogEntryList} .

op noEntries : \rightarrow \text{LogEntryList} \langle \text{ctor} \rangle .

--- Usage: Transaction LogPosition SiteId Leader- replica Updates

op _\_ : \text{TransId LogPosition SiteId OperationList} \rightarrow \text{LogEntry} \langle \text{ctor} \rangle .

op _\_ : \text{LogEntryList LogEntryList} \rightarrow \text{LogEntryList} \langle \text{ctor assoc id: noEntries} \rangle .

\text{*** (Entities) ***}

\text{sorts EntityId EntityIdSet} .

\text{subsort EntityId} < \text{EntityIdSet} .

op emptyEntityIdSet : \rightarrow \text{EntityIdSet} \langle \text{ctor} \rangle .

op entity : \text{EntityGroupId} \text{Nat} \rightarrow \text{EntityId} \langle \text{ctor} \rangle .

op _\_ : \text{EntityIdSet EntityIdSet} \rightarrow \text{EntityIdSet} \langle \text{ctor assoc id: emptyEntityIdSet} \rangle .

\text{sorts Entity EntitySet} .

\text{subsort Entity} < \text{EntitySet} .

op _\_ : \text{EntitySet EntitySet} \rightarrow \text{EntitySet} \langle \text{ctor assoc id: emptyEntitySet} \rangle .

\text{sort EntityValue} .

op v : \text{Nat} \rightarrow \text{EntityValue} \langle \text{ctor} \rangle .

\text{sorts EntityVersion EntityVersionList} .

\text{subsort EntityVersion} < \text{EntityVersionList} .

op noEntityVersions : \rightarrow \text{EntityVersionList} \langle \text{ctor} \rangle .

op _\_ : \text{LogPosition EntityValue} \rightarrow \text{EntityVersion} \langle \text{ctor} \rangle .

*** (Transactions) ***
sorts TransId.
subsort TransId < Oid.
op initTrans : -> TransId [ctor].

sorts Operation OperationList.
subsort Operation < OperationList.
op cr : EntityId -> Operation [ctor].
op w : EntityId EntityValue -> Operation [ctor].
op emptyOpList : -> OperationList [ctor].

sort TransStatus.
ops idle in-paxos : -> TransStatus [ctor].
op executing : LogPosition Time -> TransStatus [ctor].
op transTimer : Time -> TransStatus [ctor].
op defTimeout : -> Time.

***
Coordination state represents a mapping to allow a running transaction to keep metadata per replica, eg. while conducting a current read
***
sorts ReadState ReadStateSet.
subsort ReadState < ReadStateSet.
op emptyReadState : -> ReadStateSet [ctor].
op _:: : ReadStateSet ReadStateSet -> ReadStateSet [ctor assoc comm id: emptyReadState].

sorts PaxosState PaxosStateSet.
subsort PaxosState < PaxosStateSet.
op emptyPaxosState : -> PaxosStateSet [ctor].
op _:: : PaxosStateSet PaxosStateSet -> PaxosStateSet [ctor assoc comm id: emptyPaxosState].

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

op createNewTrans : TransId OperationList -> Object.
eq createNewTrans(TID, OL) =
< TID : Transaction | operations : OL, status : idle,
readState : emptyReadState,
paxosState : emptyPaxosState,

*** (Applying updates) ***
sort PendingWriteState.
ops idle : -> PendingWriteState [ctor].
op updating : Time -> PendingWriteState [ctor].

sorts PendingWrite PendingWriteList.
subsort PendingWrite < PendingWriteList.
op pw : LogPosition PendingWriteState OperationList -> PendingWrite [ctor].
op emptyPWL : -> PendingWriteList [ctor].
op _:: : PendingWriteList PendingWriteList -> PendingWriteList [ctor assoc id: emptyPWL].
*** (Update coordination ***)

**sorts**

- Propnum, DefPropnum
- Nat < Propnum
- Propnum < DefPropnum

**ops**

- noPropnum : > DefPropnum

**sorts**

- PaxosProposal, PaxosProposalSet
- PaxosProposal < PaxosProposalSet

**ops**

- emptyProposalSet : > PaxosProposalSet
- proposal : SiteId, TransId, LogPosition, OperationList, Propnum > PaxosProposal [ctor]
- accepted : SiteId, LogEntry, Propnum > PaxosProposal [ctor]

**ops**

- _ ; _ : PaxosProposalSet, PaxosProposalSet > PaxosProposalSet [ctor assoc comm id: emptyProposalSet]

*** (Replicas with metadata ***)

**sorts**

- EntityGroupReplica, EntityGroupReplicaSet
- EntityGroupReplica < EntityGroupReplicaSet

**ops**

- egr : SiteId, Nat, LogPosition > EntityGroupReplica [ctor]
- noEGR : > EntityGroupReplicaSet [ctor]
- _ ; _ : EntityGroupReplicaSet, EntityGroupReplicaSet > EntityGroupReplicaSet [ctor assoc comm id: noEGR]

**ops**

- getSites : EntityGroupReplicaSet > SiteIdSet
- eq getSites(egr(SID, N, LP)) = SID ; getSites(EGRS).
- eq getSites(noEGR) = emptyOidSet.

*** (EntityGroups ***)

**sort**

- EntityGroupId

**subsorts**

- EntityGroupId < Oid

**class** EntityGroup

- entitiesState : EntitySet
- replicas : EntityGroupReplicaSet
- proposals : PaxosProposalSet
- pendingWrites : PendingWriteList
- transactionLog : LogEntryList

*** (Site id−lists ***)

**sort**

- SiteIdList

**subsorts**

- SiteId < SiteIdList

**ops**

- emptySiteIdList : > SiteIdList [ctor]
- _ ; _ : SiteIdList, SiteIdList > SiteIdList [ctor assoc id: emptySiteIdList]

**ops**

- length : SiteIdList > Nat
- eq length(SID :: SIL) = 1 + length(SIL)
- eq length(emptySiteIdList) = 0

*** ( Necessary set constructs ***)

**sort**

- NatSet

**subsorts**

- Nat < NatSet

**ops**

- emptyNatSet : > NatSet [ctor]
- _ ; _ : NatSet, NatSet > NatSet [ctor assoc comm id: emptyNatSet]

**sorts**

- EmptyOidSet, SiteIdSet, TransIdSet, EntityGroupIdSet, OidSet

**subsorts**

- EmptyOidSet < TransIdSet
- EntityGroupIdSet < SiteIdSet < OidSet
- TransIdSet < TransIdSet
- EntityGroupId < EntityGroupIdSet
- SiteId < SiteIdSet
- Oid < OidSet

**eqs**

- emptySiteIdList
- length(SID :: SIL)
- length(emptySiteIdList)
- emptyNatSet
- _ ; _ : SiteIdList, SiteIdList

---

### B Real-Time Maude Model of Megastore
\begin{verbatim}
op emptyOidSet : -> EmptyOidSet [ctor] .
op .- : EmptyOidSet EmptyOidSet -> EmptyOidSet [ctor assoc comm id: emptyOidSet] .
op .- : TransIdSet TransIdSet -> TransIdSet [ctor ditto] .
op .- : EntityGroupIdSet EntityGroupIdSet -> EntityGroupIdSet [ctor ditto] .
op .- : SiteIdSet SiteIdSet -> SiteIdSet [ctor ditto] .
op .- : OidSet OidSet -> OidSet [ctor ditto] .

op _setminus_ : OidSet OidSet -> OidSet [assoc] .

op _in_ : Oid OidSet -> Bool .

op size : OidSet -> Nat .

endom)

endom)
\end{verbatim}
Listing B.3  client_unc.rtmaude

(omod CLIENT−FOR−MODEL−CHECKING is
  inc CLIENT−INTERFACE .
  inc TRANSACTION−HISTORY .

  var TID : TransId .
  vars OL : OperationList .
  vars SID SID' : SiteId .
  var LOCALTRANS : Configuration .
  var READS : EntitySet .
  var WRITES : OperationList .
  var EG : EntityGroupId .
  var EVAL : EntityValue .
  vars N N' : Nat .
  vars LP LP' : LogPosition .
  var EID : EntityId .
  var ES : EntitySet .
  var TH : Oid .
  var EGLP : EntGroupLogPosPairSet .

  --- Receive transaction request
  rl [newTrans] :
    (newTrans(SID, TID, OL))
  < SID : Site | localTransactions : LOCALTRANS >
  =>
  < SID : Site | localTransactions : createNewTrans(TID, OL) LOCALTRANS > .

  rl [notifyCommit] :
    (notifyCommit(SID, TID, EGLP, READS, WRITES))
  < TH : TransactionHistory | >
  =>
    updateHistoryReads(TID, READS,
    updateHistoryWrites(TID, EGLP, WRITES, < TH : TransactionHistory | >)) .

  rl [notifyReadOnlyCommit] :
    (notifyReadOnlyCommit(SID, TID, READS))
  < TH : TransactionHistory | >
  =>
    updateHistoryReads(TID, READS, < TH : TransactionHistory | >) .

  op updateHistoryReads : TransId EntitySet Configuration -> Configuration .
  eq updateHistoryReads(TID, (EID |-> (LP' EVAL)) ; ES, THIST) =
    updateHistoryReads(TID, ES, addRead(TID, EID, LP', THIST)) .
  eq updateHistoryReads(TID, emptyEntitySet, THIST) = THIST .

  op updateHistoryWrites : TransId EntGroupLogPosPairSet OperationList Configuration ->
    Configuration .
  eq updateHistoryWrites(TID, eglp(EG, LP) : EGLP, w(entity(EG,N), EVAL) ; OL, THIST) =
    updateHistoryWrites(TID, eglp(EG, LP) ; EGLP, OL, addWrite(TID, entity(EG,N), LP, THIST)) .
  eq updateHistoryWrites(TID, EGLP, emptyOpList, THIST) = THIST .

  rl [notifyConflictAbort] :
    (notifyConflictAbort(SID, TID, READS, WRITES))
  =>
    none .
endom)

Listing B.4  current_read.rtmaude
var TID TID' : TransId.
var SID SID' : SiteId.
var TS : TransStatus.
var SIS SIS' : SiteIdSet.
var EGRS : EntityGroupReplicaSet.
var EID : EntityId.
var EGID : EntityGroupId.
var CNT N N1 N2 : Nat.
var EGIS SIS' : SiteIdSet.
var LOCALTRANS : Configuration.
var OL OL' : OperationList.
var ESTATE READS : EntitySet.
var EV : EntityVersion.
var DLP : DefLogPosition.
var LP LP' : LogPosition.
var VAL1 VAL2 : EntityValue.
var EG : EntityGroupId.
var EGROUPS : Configuration.
var EVERSIONS EVERSIONS' : EntityVersionList.
var RSTATE : ReadStateSet.
var LEL : LogEntryList.
var T : Time.
var SIL : SiteIdList.
var OL1 OL2 : OperationList.

op readpos : EntityGroupId LogPosition -> ReadState [ctor].

ops readDelay : -> Time.
------ Proceed transaction locally

***

--- Current read:
--- * If local coordinator is up-to-date (e.g. an
---   entry for the given entityid exists in the coordinator state):
---   Read locally.
--- * If local coordinator is not up-to-date, perform
---   a majority read to find the maximum logposition. Once a given
---   logposition has been received from a majority of sites, the
---   most responsive replica is elected for a "catchup". See
---   MAJORITY-READ for details. In addition to the modelled delay
---   for local access (representing the actual bigtable-lookup), we
---   require the pending write queue to be empty. We store the most
---   recent log entry upon start of the read – this LP is maintained
---   throughout the transaction. Any conflict with concurrent
---   updates will then be detected upon commit.

*** A: Non-faulty scenario: Perform a local read

crl [CRA1-startCurrentLocalRead] :< SID : Site |
    coordinator : (eglp(EG, LP) ; CES),
    entityGroups : < EG : EntityGroup |
    pendingWrites : emptyPWList > EGROUPS,
    localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, status : idle >
LOCALTRANS >
< SID : Site |
localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, status :
executing(LP, readDelay) > LOCALTRANS >
if not (containsUpdate(entity(EG,N), OL) and inConflictWithRunning(EG, LOCALTRANS)) .

op inConflictWithRunning : EntityGroupId Configuration --> Bool .
ceq inConflictWithRunning(EG, < TID : Transaction | status : TS, reads : READS, operations : GL1 : w(entity(EG,N), VAL1) :: OL2 > LOCALTRANS) = true
if not (TS == idle and filterReads(EG, READS) == emptyEntitySet) .

eq inConflictWithRunning(EG, < TID : Transaction | writes : GL1 : w(entity(EG,N), VAL1) :: OL2 > LOCALTRANS) = true .
eq inConflictWithRunning(EG, LOCALTRANS) = false [wise] .

op filterReads : EntityGroupId EntitySet --> EntitySet .
eq filterReads(EG, {entity(EG,N) |--> EV} : ES) = {entity(EG,N) |--> EV} : filterReads(EG, ES) .
eq filterReads(EG, ES) = emptyEntitySet [wise] .

op containsUpdate : EntityId OperationList --> Bool .
eq containsUpdate(EID, OL1 : w(EID, VAL1) :: OL2) = true .
eq containsUpdate(EID, OL) = false [wise] .

rl [CRA2-"endCurrentLocalRead" : < SID : Site |
entityGroups :
< EG : EntityGroup | entitiesState : {entity(EG,N) |--> EVERSIONS} : BSTATE > EGROUPS,
localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, status :
executing(LP, 0), readState : RSTATE, reads : READS > LOCALTRANS >
=>
< SID : Site |

ceq getVersion(lpos(N), EVERSION : (lpos(N1) VAL1) :: (lpos(N2) VAL2) :: EVERSIONS') = (lpos(N1) VAL1)
if (N1 < N \ N < N2) .
ceq getVersion(lpos(N), EVERSION : (lpos(N1) VAL1)) = (lpos(N1) VAL1) if (N1 <= N) .

op hasQuorum : Nat SiteIdSet --> Bool .
eq hasQuorum(N, SIS) = (N >= (size(SIS) quo 2 + 1)) .

e ndtom)

Listing B.5 updates.rtmaude

{(mod UPDATES is
  inc CLIENT-INTERFACE .
  inc CURRENT-READ .

var EID : EntityId .
var EIDSET : EntityGroupIdSet .
vars N N1 N2 : Nat .
var NS : NatSet .
vars T EXP : Time .
vars PN PN' PROPNUM : Propnum .
var PXSID : Nat .
vars DPN SEEN-PROPNUM : DefPropnum .
vars EG EG' : EntityGroupId .
var EGROUPS : Configuration .

endmod)
vars TID TID' TID1 TID2 TID3 : TransId.
vars SID SID' MSID1 MSID2 SENDER THIS : SiteId.
vars SIS SIS' FAILED REPLICAS : SiteIdSet.
var EGRS : EntityGroupReplicaSet.
var RSTATE : ReadStateSet.
vars PSTATE NEW PAIXOS STATE : PaxosStateSet.
vars LOCALTRANS LTRANS1 LTRANS2 : Configuration.
vars WRITEOPS OL' OL1 OL2 : OperationList.
var OP : Operation.
var PROPSET : PaxosProposalSet.
vars VAL VAL' : EntityValue.
vars LE LEL' : LogEntryList.
var LE LE' NEW NEXT VALUE : LogEntry.
var LP LP' : LogPosition.
var ES : EntitySet.
vars EVERSIONS EVERSIONS' : EntityVersionList.
var PWL : PendingWriteList.
var CA : Bool.
var RND : Oid.
var CE : EntGroupLogPosPair.
var CES : EntGroupLogPosPairSet.
var READS : EntitySet.
var WRITES : OperationList.

op defPropExp : -- > Time.
op updateDelay : -- > Time.

*** (Messages ***)
op acceptLeaderReq : TransId EntityGroupId LogEntry -- > MsgContent.
op acceptLeaderRsp : TransId EntityGroupId LogPosition -- > MsgContent.
op signalConflict : TransId EntityGroupId LogPosition -- > MsgContent.
op acceptAllReq : TransId EntityGroupId LogEntry Propnum -- > MsgContent.
op acceptAllRsp : TransId EntityGroupId LogPosition Propnum -- > MsgContent.
op applyReq : TransId EntityGroupId LogPosition Propnum -- > MsgContent.

op acceptLeader : EntityGroupId LogEntry SiteId Time -- > PaxosState [ctor].
  --- Propnum == proposal number, SiteIdSet1 == sites responded yes
op acceptAll : EntityGroupId LogEntry Propnum SiteIdSet Time -- > PaxosState [ctor].
  --- SiteIdSet == sites who did not accept
op acceptedPS : EntityGroupId LogEntry Propnum -- > PaxosState [ctor].

*** ( Paxos states involved in presence of errors, see UPDATE -- FAULT -- HANDLERS ***)
op prepare : EntityGroupId LogEntry Propnum DefPropnum SiteIdSet Time -- > PaxosState [ctor].
op restartPrepare : EntityGroupId LogEntry Time -- > PaxosState [ctor].
  --- SiteIdSet == sites who did not accept
op invalidating : EntityGroupId LogEntry Propnum SiteIdSet Time -- > PaxosState [ctor].

rl [bufferWriteOperation] :
  < SID : Site |
  localTransactions : < TID : Transaction | operations : w(EID, VAL) :: OL, writes : WRITEOPS, 
  status : idle > LOCALTRANS 
>
=>
  < SID : Site |
  localTransactions : < TID : Transaction | operations : OL, writes : WRITEOPS :: w(EID, VAL) > LOCALTRANS 
>.
(Section 4.6.3 of Baker et.al Accept Leader. No conflicting proposal ***)

--- Note: In the "Fast write"-scenario, we do not run the explicit prepare step. But it appears
--- correct to regard the present proposal as proposal number 0 and store this at the leader
--- (if the original proposer then fails, there is a chance its value will "survive" due to
--- this)

crl [l2s-successfulLeaderAccept]:
(msg acceptLeaderReq(TID, EG, (TID LP SID OL)) from SENDER to THIS)
< THIS : Site |
  entityGroups : < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL > EGROUPS
>
=>
< THIS : Site |
  entityGroups : |
    < EG : EntityGroup |
      proposals : accepted(SENDER, (TID LP SID OL), 0) ; PROPSET |
    >
  EGROUPS
>
(uniCast acceptLeaderRsp(TID, EG, LP) from THIS to SENDER)
if not (containsLPos(LP, LEL) or conflictingProposal(TID, LP, 0, PROPSET)) .


ceq conflictingProposal(TID, LP, PROPNUM, proposal(SID, TID', LP, OL, PN) ; PROPSET) =
  true if (PN >= PROPNUM) .

cq conflictingProposal(TID, LP, PROPNUM, accepted(SID, LE, PN) ; PROPSET) =
  true if (PN >= PROPNUM) .

eq conflictingProposal(TID, LP, PROPNUM, PROPSET) = false [owise] .


eq containsLPos(LP, LEL :: (TID LP SID OL)) :: LEL') = true .

eq containsLPos(LP, LEL) = false [owise] .

*** (Section 4.6.3 Accept Leader/Invalidate. Paxos with conflicting proposals ***)

crl [L1rcvAcceptLeaderReq] :
  (msg acceptLeaderReq(TID, EG, (TID LP MSID1 OL1)) from SENDER to THIS)
< THIS : Site |
  entityGroups : < EG : EntityGroup |
    transactionLog : LEL,
    proposals : PROPSET > EGROUPS
>
=>
< THIS : Site |
  entityGroups : < EG : EntityGroup | > EGROUPS
>
(uniCast signalConflict(TID, EG, LP) from THIS to SENDER)
if (containsLPos(LP, LEL) or conflictingProposal(TID, LP, 0, PROPSET)) .

--- If we receive a conflict signal, we abort the transaction.

crl [L2rcvDenyLeaderRsp] :
  (msg signalConflict(TID, EG, LP) from SENDER to THIS)
< THIS : Site |
  localTransactions : < TID : Transaction | reads : READS, writes : WRITES > LOCALTRANS
>
=>
< THIS : Site | localTransactions : LOCALTRANS >
(notifyConflictAbort(THIS, TID, READS, WRITES)) .

--- We ignore a conflict signal for an already missing transaction.

crl [L2rcvDenyLeaderRsp] :
  (msg signalConflict(TID, EG, LP) from SENDER to THIS)
< THIS : Site | localTransactions : LOCALTRANS >
=>
< THIS : Site |>
if not containsTrans(TID, LOCALTRANS) .

op containsTrans : TransId Configuration --> Bool .

eq containsTrans(TID, < TID : Transaction | > LOCALTRANS) = true .
eq containsTrans(TID, LOCALTRANS) = false [owise] .
--- If accept—leader message arrives delayed, ignore the message

\[
\text{crl} \left\langle \text{LFrecvAcceptLeaderRspDelayed} : (\text{msg acceptLeaderRsp(TID, EG, LP) from SENDER to THIS}) \right. \\
< \text{THIS : Site} | \\
\text{localTransactions : LOCALTRANS} > \\
\left. \right. = > \\
< \text{THIS : Site} | \\
\text{localTransactions : LOCALTRANS} > \\
\text{if (not acceptingLeader(TID, EG, LP, LOCALTRANS))} .
\]

\[
\text{op acceptingLeader : TransId EntityGroupId LogPosition Configuration } \rightarrow \text{Bool} .
\]

\[
\text{eq acceptingLeader(TID, EG, LP,} \\
< \text{TID : Transaction | paxosState : acceptLeader(EG, (TID LP SID OL), MSID1, EXP) : PSTATE > } \\
\text{LOCALTRANS} = \text{true} \\
\text{eq acceptingLeader(TID, EG, LP, LOCALTRANS) } = \text{false} \text{ [wise]} .
\]

***( Section 4.6.3 – Accept—step ***)

*** Common case: Received accept from leader, proceed with requesting accept from all

\[
\text{rl \langle LFrecvAcceptLeaderRsp} : (\text{msg acceptLeaderRsp(TID, EG, LP) from SENDER to THIS}) \right. \\
< \text{THIS : Site} | \\
\text{localTransactions : } < \text{TID : Transaction | paxosState : acceptLeader(EG, (TID LP SID OL),} \\
\text{SENDER, EXP) : PSTATE,} \\
\text{status : in—paxos, reads : READS, writes : WRITES > } \\
\text{LOCALTRANS,} \\
\text{entityGroups : } < \text{EG : EntityGroup | proposals : PROPSET, replicas : EGRS > EGROUPS >} \\
\left. \right. = > \\
< \text{THIS : Site} | \\
\text{entityGroups : EGROUPS} \\
< \text{EG : EntityGroup} | \\
\text{proposals : if (SENDER } =/= \text{ THIS) then} \\
\text{accepted(THIS, (TID LP SID OL), 0) else emptyProposalSet fi} \text{ : PROPSET >,} \\
\text{localTransactions : LOCALTRANS} \\
\text{if } (((\text{getSites(EGRS) setminus (SENDER ; THIS)}) =/= emptyOidSet) \text{ then} \\
< \text{TID : Transaction |} \\
\text{paxosState : acceptAll(EG,(TID LP SID OL), 0, (THIS ; SENDER), defTimeout) : PSTATE,} \\
\text{status : in—paxos >} \\
\text{else} \\
\text{none} \\
\text{--- With two replicas and the other being master, or only one and ourself being master, we’re done now} \\
\text{if allEGSAccepted(PSTATE) then} \text{ then none} \\
\text{else (} \\
< \text{TID : Transaction | paxosState : acceptedPS(EG, (TID LP SID OL), 0) : PSTATE > } \\
\text{fi) fi) fi) fi} \\
\text{fi} \\
\text{if } (((\text{getSites(EGRS) setminus (SENDER ; THIS)}) =/= emptyOidSet) \text{ then} \\
(\text{multiCast acceptAllReq(TID, EG, (TID LP SID OL), 0) from THIS to getSites(EGRS) setminus (SENDER ; THIS)}) \\
\text{else (if allEGSAccepted(PSTATE) then (} \\
\text{createApplyMessages(THIS, < EG : EntityGroup | > EGROUPS, acceptedPS(EG, (TID LP SID OL), 0) ;} \\
PSTATE)
### B Real-Time Maude Model of Megastore

notifyCommit(TID, eglp(EG, LP); getEntGroupLogPosPair(PSTATE), READS, WRITES))
else none
fi)
fi).

--- Common case: This is the first time we receive an accept for this log position

\[ A_{\text{recvAcceptAllReq}} : (msg \text{ acceptAllReq}(TID, EG, (TID' LP SID OL), PROPNUM) \text{ from SENDER to THIS}) \]
\[ < \text{THIS} : \text{Site} | \] entityGroups : < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL > EGROUPS
\[ > ==< \text{THIS} : \text{Site} | \] entityGroups : < EG : EntityGroup | proposals : accepted(SENDER, (TID LP SID OL'), PN); PROPSET >
removeProposal(LP, PROPSET) > EGROUPS >
(unicast acceptAllReq(TID, EG, LP, PROPNUM) \text{ from THIS to SENDER})
if not (containsPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET)) .

\[ \text{op hasAcceptedForPosition} : \text{LogPosition} \times \text{PaxosProposalSet} \rightarrow \text{Bool} . \]
\[ \text{eq hasAcceptedForPosition}(LP, accepted(SID, (TID LP SID OL'), PN); PROPSET) = \text{true} . \]
\[ \text{eq hasAcceptedForPosition}(LP, PROPSET) = \text{false} \] (owise).

\[ \text{op removeProposal} : \text{LogPosition} \times \text{PaxosProposalSet} \rightarrow \text{PaxosProposalSet} . \]
\[ \text{eq removeProposal}(LP, proposal(SID, TID, LP, OL, PN); PROPSET) = \text{removeProposal}(LP, PROPSET) . \]
\[ \text{eq removeProposal}(LP, PROPSET) = \text{PROPSET} \] (owise).

--- Log the accept—vote. If this was the last, proceed the transaction

\[ A_{\text{recvAcceptAllRsp}} : (msg \text{ acceptAllRsp}(TID, EG, LP, PROPNUM) \text{ from SENDER to THIS}) \]
\[ < \text{THIS} : \text{Site} | \] entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
localTransactions : < TID : Transaction | paxosState : acceptAll(EG, (TID' LP SID OL), PROPNUM,
SIS, EXP); PSTATE, reads : READS, writes : WRITES > LOCALTRANS
\[ > ==< \text{THIS} : \text{Site} | \text{localTransactions : }
\] LOCALTRANS
(if ((SENDER : SIS) != getSites(EGRS)) then
\[ < \text{TID} : \text{Transaction} | \] paxosState : acceptAll(EG, (TID' LP SID OL), PROPNUM, SIS ; SENDER, EXP); PSTATE >
else (if allEGSAccepted(PSTATE) then none
else (if allEGSAccepted(PSTATE) then none
else (if allEGSAccepted(PSTATE) then none
else (if allEGSAccepted(PSTATE) then none
else
\[ < \text{TID} : \text{Transaction} | \text{paxosState : acceptedPS}(EG, (TID' LP SID OL), PROPNUM); PSTATE > ) \]
fi)
fi)
fi)
fi).

\[ \text{if ((SIS ; SENDER) \text{ getSites}(EGRS) and allEGSAccepted(PSTATE)) then (} \]
createApplyMessages(TTHIS, EG : EntityGroup | > EGROUPS, acceptedPS(EG, (TID' LP SID OL), PROPNUM)
PSTATE) \]
\[ \text{if (TID == TID')} then ( \]
notifyCommit(TTHIS, TID, eglp(EG, LP); getEntGroupLogPosPair(PSTATE), READS, WRITES))
else (notifyConflictAbort(TTHIS, TID, READS, WRITES))
fi))
else none
fi).

\[ \text{op createApplyMessages} : \text{SiteId} \times \text{Configuration} \rightarrow \text{PaxosState} \rightarrow \text{Configuration} . \]
eq createApplyMessages(SID, < EG : EntityGroup | replicas : EGRS > EGROUPS, acceptedPS(EG, (TID LP SID OL), PROPNUM) : PSTATE) =
   (multiCast applyReq(TID, EG, LP, PROPNUM) from SID to getSites(EGRS)) createApplyMessages(SID, EGROUPS, PSTATE).

eq createApplyMessages(SID, EGROUPS, emptyPaxosState) = none.

op getEntGroupLogPosPair : PaxosState -> EntGroupLogPosPairSet.

eq getEntGroupLogPosPair(acceptedPS(EG, (TID LP SID OL), PROPNUM) ; PSTATE) = eglp(EG, LP) ; getEntGroupLogPosPair(PSTATE).

eq getEntGroupLogPosPair(emptyPaxosState) = noEntGroupLogPosPair.

### (Section 4.6.3 - Apply step ***)

op allEGSAccepted : PaxosStateSet -> Bool.

eq allEGSAccepted(acceptedPS(EG, LE, N) ; PSTATE) = allEGSAccepted(PSTATE).

eq allEGSAccepted(emptyPaxosState) = true.

eq allEGSAccepted(PSTATE) = false [owise].

--- Apply at site which previously accepted a proposal

crl [APP3initUpdates]:
   (msg applyReq(TID2, EG, lpos(N2), PN) from SENDER to THIS)
< THIS: Site |
   coordinator : eglp(EG, lpos(N1)) ; CES,
   entityGroups : < EG : EntityGroup | transactionLog : LEL (TID1 lpos(N1) MSID1 OL1),
   pendingWrites : PWL,
   proposals : acceptedSID, (TID3 lpos(N2) MSID2 OL2), PN' ;
   PROPS > EGROUPS,
   localTransactions : LOCALTRANS
>
=>
< THIS: Site |
   coordinator : eglp(EG, lpos(N2)) ; CES,
   entityGroups : < EG : EntityGroup | transactionLog : LEL (TID1 lpos(N1) MSID1 OL1) (TID3 lpos(N2) MSID2 OL2),
   pendingWrites : pw(lpos(N2)), idle, OL2 : PWL,
   proposals : removeProposals(lpos(N2), PN'), PROPS > EGROUPS,
   localTransactions : removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)
>
   (sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)))
if N2 == s N1 \ (PN == PN') or (TID2 == TID3).

op removeOthersForLogPosition : EntityGroupId LogPosition Configuration -> Configuration.

eq removeOthersForLogPosition(EG, LP, < TID2 : Transaction | paxosState : prepare(EG, (TID3 LP MSID1 OL1), PN, PN', SIS, EXP) ; PSTATE > LOCALTRANS) =
   removeOthersForLogPosition(EG, LP, LOCALTRANS).

eq removeOthersForLogPosition(EG, LP, < TID2 : Transaction | paxosState : restartPrepare(EG, (TID3 LP MSID1 OL), EXP) ; PSTATE > LOCALTRANS) =
   removeOthersForLogPosition(EG, LP, LOCALTRANS).

eq removeOthersForLogPosition(EG, LP, < TID2 : Transaction | paxosState : acceptAll(EG, (TID3 LP MSID1 OL), PN, SIS, EXP) ; PSTATE > LOCALTRANS) =
   removeOthersForLogPosition(EG, LP, LOCALTRANS).

eq removeOthersForLogPosition(EG, LP, < TID2 : Transaction | paxosState : acceptLeader(EG, (TID3 LP MSID1 OL), MSID2, EXP) ; PSTATE > LOCALTRANS) =
   removeOthersForLogPosition(EG, LP, LOCALTRANS).

eq removeOthersForLogPosition(EG, LP, LOCALTRANS) = LOCALTRANS [owise].

op removeProposals : LogPosition PaxosProposalSet -> PaxosProposalSet.

eq removeProposals(LP, acceptedSID, (TID LP MSID1 OL1), PN) ; PROPS = removeProposals(LP, PROPS).

eq removeProposals(LP, proposal(SID, TID, LP, OL, PN) ; PROPS) = removeProposals(LP, PROPS).
eq removeProposals(LP, PROPSET) = PROPSET [owise].

op sendNotifyAbort : SiteId Configuration Configuration -> Configuration.
eq sendNotifyAbort(SID, < TID : Transaction | > LTRANS1, < TID : Transaction | > LTRANS2) = 
    sendNotifyAbort(SID, LTRANS1, LTRANS2).
eq sendNotifyAbort(SID, none, LTRANS1) = none.
eq sendNotifyAbort(SID, < TID : Transaction | reads : READS, writes : WRITES > LTRANS1, LTRANS2) = 
    (notifyConflictAbort(SID, TID, READS, WRITES)) sendNotifyAbort(SID, LTRANS1, LTRANS2) [owise].

rl [APP4beginPendingWrite]:
< THIS : Site |
    entityGroups : < EG : EntityGroup | pendingWrites : PWL :: pw(LP, idle, OL) > EGROUPS
> 
=>
< THIS : Site |
    entityGroups : < EG : EntityGroup | pendingWrites : PWL :: pw(LP, updating(updateDelay), OL) > EGROUPS
> .

rl [APPSendPendingWrite]:
< THIS : Site | entityGroups : < EG : EntityGroup | entitiesState : ES, pendingWrites : PWL :: pw(LP, updating(0), OL :: OP) > EGROUPS > 
=>
< THIS : Site | entityGroups : 
    < EG : EntityGroup | entitiesState : applyUpdates(OP, LP, ES), pendingWrites : updatePWListUponComplete(LP, OL, PWL) > EGROUPS > .

op updatePWListUponComplete : LogPosition OperationList PendingWriteList -> PendingWriteList .
eq updatePWListUponComplete(LP, emptyOpList, PWL) = PWL .
eq updatePWListUponComplete(LP, OL, PWL) = PWL :: pw(LP, idle, OL) [owise] .

op applyUpdates : OperationList LogPosition EntitySet -> EntitySet .
eq applyUpdates(w(EID, VAL) :: OL, LP, (EID | -> EVERSIONS) :: ES) = (EID | -> insertEntityValSorted((LP VAL), EVERSIONS)) :: ES .
eq applyUpdates(emptyOpList, LP, ES) = ES .

eq insertEntityValSorted((lpos(N) VAL), (lpos'(N') VAL') :: EVERSIONS) = 
    if (N' < N) then 
        ((lpos'(N') VAL') :: insertEntityValSorted((lpos(N) VAL), EVERSIONS)) 
    else 
        ((lpos(N) VAL) :: (lpos'(N') VAL') :: EVERSIONS) 
    fi .
eq insertEntityValSorted((lpos(N) VAL), noEntityVersions) = (lpos(N) VAL) .
endtm)

Listing B.6 majority_read.rtaude

(* This module implements "catchup", see step 3 and 4 of section 4.6.2 in Baker et al. *)
(tomod MAJORITY--READ is
    inc UPDATES .
    var EG : EntityGroupId .
    var EGROUPS : Configuration .
)
**Majority read**

Due to some previous fault, the local coordinator is not up-to-date. Perform a majority read

**r1 [CRB1-initMajorityRead]**:

< SID : Site |
  coordinator : CES,
  entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
  localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, readState : RSTATE,
  status : idle > LOCALTRANS >

=>< SID : Site |
  localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL,
  readState : majorityRead(entity(EG,N), noLogPosition, emptySiteIdList) ; RSTATE,
  status : transTimer(defTimeout) > LOCALTRANS > (multiCast majorityRead(EG, TID) from SID to getSites(EGRS) setminus SID)
if not (inConflictWithRunning(EG, LOCALTRANS) or containsEntityGroupId(EG, CES)) .

**r1 [CRB2-rcvMajorityReadRequest]**:

{msg majorityRead(EG,TID) from SENDER to THIS}
< THIS : Site |
  entityGroups : < EG : EntityGroup | transactionLog : LEL :: (TID' lpos(N) SID OL) > EGROUPS >

=>< THIS : Site |
  (uniCast majorityReadResponse(EG, TID, lpos(N)) from THIS to SENDER) .

**r1 [CRB3-rcvMajorityReadResponse]**:

{msg majorityReadResponse(EG, TID, LP) from SENDER to THIS}
\[
\begin{align*}
&\text{readState : majorityRead(entity(EG,N), LATEST, SIL') :: SID) ; RSTATE > LOCALTRANS >}
\end{align*}
\]
> 
< THIS : Site | entityGroups : < EG : EntityGroup | > EGROUPS,
localTransactions : < TID : Transaction | 
op getMostRecentLPos : getMostRecentLPos | 
operations : cr(entity(EG, N)) :: OL, status : transTimer(defTimeout),
readState : majorityRead(entity(EG, N), noLogPosition, emptySiteIdList) ;
RSTATE > LOCALTRANS
> 
(multiCast majorityRead(EG, TID) from THIS to getSites(EGRS) setminus THIS).

*** Perform catchup ***
--- Determine missing entries in a given log
op getLogHoles : LogEntryList -- LogPositionList .
ceq getLogHoles((TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2 OL2) :: LEL) =
getLogHoles((TID2 lpos(N2) MSID2 OL2) :: LEL) if N2 == s N1 .
ceq getLogHoles((TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2 OL2) :: LEL) =
addLogPositionsBetween(lpos(s N1), lpos(sd(N2,1))) :: getLogHoles((TID2 lpos(N2) MSID2 OL2) :: LEL )
if N2 == s N1 .
eq getLogHoles(LE) = emptyLPlist .

--- Utility function: Return a log--position list between the two positions N1 and N2 (including
them both)
ceq addLogPositionsBetween(lpos(N1), lpos(N2)) =
lpos(N1) :: addLogPositionsBetween(lpos(s N1), lpos(N2)) if N1 <= N2 .
eq addLogPositionsBetween(lpos(N1), lpos(N2)) = emptyLPlist [wise] .

--- Utility function: Get the most recent log entry.
--- Invariant: The log entry list always has one element
op getMostRecentLPos : LogEntryList -- LogPosition .
eq getMostRecentLPos(LEL :: (TID LP SID OL)) = LP .

--- Upon receiving a catchup request, traverse the log position list representing
--- missing entries, and respond with all entries present at this site
rl [rcvCatchupRequest] :
  (msg catchupRequest(EG, TID, LPL) from SENDER to THIS)
< THIS : Site |
< entityGroups : < EG : EntityGroup | transactionLog : LEL > EGROUPS
> 
< THIS : Site |
< uniCast catchupResponse(EG, TID, getPresentEntries(LPL, LEL)) from THIS to SENDER) .

op getPresentEntries : LogPositionList LogEntryList -- LogEntryList .
eq getPresentEntries(lpos(N) :: LPL, LEL :: (TID lpos(N) SID OL) :: LEL') =
(TID lpos(N) SID OL) :: getPresentEntries(LPL, LEL :: (TID lpos(N) SID OL) :: LEL') .
eq getPresentEntries(emptyLPlist, LEL) = noEntries .
eq getPresentEntries(lpos(N) :: LPL, LEL) =
getPresentEntries(LPL, LEL) [wise] .

crl [rcvCatchupResponse] :
  (msg catchupResponse(EG, TID, LEL--RECEIVED) from SENDER to THIS)
< THIS : Site | localTransactions : LOCALTRANS >
=>
< THIS : Site |
if (not catchingUp(TID, LOCALTRANS)) .

op catchingUp : TransId Configuration -- Bool .
eq catchingUp(TID, < TID : Transaction | readState : catchingUp(EID, SIL, LPL) ; RSTATE > LOCALTRANS) = true .
eq catchingUp(TID, LOCALTRANS) = false [wise] .

crl [rcvCatchupResponse]:
  (msg catchupResponse(EG, TID, LEL−RECEIVED) from SENDER to THIS)
  < THIS : Site | coordinator : CES,
    entityGroups : < EG : EntityGroup | replicas : EGRS, proposals : PROPSET,
    transactionLog : LEL, pendingWrites : PWL >
    EGROUPS,
    localTransactions : < TID : Transaction |
      status : TSTATUS,
      readState : catchingUp(entity(EG, N), SIL, LPL) ; RSTATE > LOCALTRANS
  >
  =>
  < THIS : Site | coordinator : (if CATCHUP−COMPLETE
    then setValidated(EG, getMostRecentLPpos(NEW−TRANS−LOG), CES)
    else CES fi),
    entityGroups : < EG : EntityGroup | proposals : removeObsoleteProposals(LEL− RECEIVED, PROPSET),
    transactionLog : NEW−TRANS−LOG,
    pendingWrites : addPendingWrites(LEL−RECEIVED, PWL) > EGROUPS,
    localTransactions : removeObsoleteTrans(TID, EG, LEL−RECEIVED, LOCALTRANS)
  )
  if CATCHUP−COMPLETE then
    < TID : Transaction | status : idle, readState : RSTATE >
  else
    if (NEXT−SITE /= emptySiteIdList) then
      (< TID : Transaction | status : transTimer(defTimeout),
        readState : catchingUp(entity(EG, N), removeIfPresent(NEXT−SITE, SIL), LPL −MISSING) ; RSTATE >)
    else
      (< TID : Transaction | status : transTimer(defTimeout),
        readState : majorityRead(entity(EG, N), noLogPosition, emptySiteIdList);
        RSTATE >)
      fi
    fi
  fi
( sendNotifyAbort(TTHIS, LOCALTRANS, removeObsoleteTrans(TID, EG, LEL−RECEIVED, LOCALTRANS)))
if (not CATCHUP−COMPLETE) then
  if (NEXT−SITE /= emptySiteIdList) then
    (uniCast catchupRequest(EG, TID, LPL−MISSING) from THIS to NEXT−SITE)
  else
    (multiCast majorityRead(EG, TID) from THIS to getSites(EGRS) setminus THIS)
  fi
else
  none fi
if LPL−MISSING := getLogHoles(applyRemoteLogEntries(LEL−RECEIVED, LEL)) \ /
  CATCHUP−COMPLETE := (LPL−MISSING == emptyLPlist) \ /
  NEW−TRANS−LOG := applyRemoteLogEntries(LEL−RECEIVED, LEL) \ /
  NEXT−SITE := getNextSite(SIL) .

op getNextSite : SiteIdList → SiteId .
eq getNextSite(SIL :: SID) = SID .
eq getNextSite(emptySiteIdList) = emptySiteIdList .

op removeIfPresent : SiteId SiteIdList → SiteIdList .
eq removeIfPresent(SID, SIL :: SID) = SIL :: SIL' .
eq removeIfPresent(SID, SIL) = SIL [wise] .

op removeObsoleteProposals : LogEntryList PaxosProposalSet → PaxosProposalSet .
eq removeObsoleteProposals((TID1 LP MSID1 OL1) :: LEL, proposal(SID, TID2, LP, OL, PW) ; PROPSET) =
  removeObsoleteProposals(LEL, PROPSET) .
eq removeObsoleteProposals((TID1 LP MSID1 OL1) :: LEL, accepted(SID, (TID2 LP MSID2 OL2), PN) :: PROPSET) = removeObsoleteProposals(LEL, PROPSET).

eq removeObsoleteProposals(noEntries, PROPSET) = PROPSET.

eq removeObsoleteProposals((TID1 LP MSID1 OL1) :: LEL, PROPSET) = removeObsoleteProposals(LEL, PROPSET) [ovise].

op removeObsoleteTrans : TransId EntityGroupId LogEntryList Configuration --> Configuration.

eq removeObsoleteTrans(TID1, EG, (TID2 LP MSID1 OL) :: LEL, LOCALTRANS) = removeObsoleteTrans(TID1, EG, LEL, removeOthersForLogPosition(EG, LP, LOCALTRANS)).

eq removeObsoleteTrans(TID1, EG, noEntries, LOCALTRANS) = LOCALTRANS.

op applyRemoteLogEntries : LogEntryList LogEntryList --> LogEntryList.

--- We might receive multiple catchup–response (due to failures). If we already have the log entry, simply ignore it.

eq applyRemoteLogEntries((TID 1pos(N) SID OL) :: LEL–RECEIVED, LEL :: (TID 1pos(N) SID OL) :: LEL') = applyRemoteLogEntries(LEL–RECEIVED, LEL :: (TID 1pos(N) SID OL) :: LEL').

--- Common case: Find the right hole in the log, and insert missing entry.

eq applyRemoteLogEntries((TID1 1pos(N1) MSID1 OL1) :: LEL–RECEIVED, LEL :: (TID2 1pos(N2) MSID2 OL2) :: (TID3 1pos(N3) MSID3 OL3) :: LEL') = applyRemoteLogEntries(LEL–RECEIVED, LEL :: (TID2 1pos(N2) MSID2 OL2) :: (TID1 1pos(N1) MSID1 OL1) :: LEL') if (N2 < N1 \ N1 < N3).

--- A missing log entry might have arrived while we wait for catchup

eq applyRemoteLogEntries((TID1 1pos(N1) MSID1 OL1) :: LEL–RECEIVED, LEL :: (TID1 1pos(N1) MSID1 OL1) :: LEL) = applyRemoteLogEntries(LEL–RECEIVED, LEL :: (TID1 1pos(N1) MSID1 OL1) :: LEL–RECEIVED).

--- We're at the end of the local log, append all entries

eq applyRemoteLogEntries((TID1 1pos(N1) MSID1 OL1) :: LEL–RECEIVED, LEL :: (TID2 1pos(N2) MSID2 OL2)) = LEL :: (TID2 1pos(N2) MSID2 OL2) :: (TID1 1pos(N1) MSID1 OL1) :: LEL–RECEIVED if N1 > N2.

--- All entries applied, we are done

eq applyRemoteLogEntries(noEntries, LEL) = LEL.

op addPendingWrites : LogEntryList PendingWriteList --> PendingWriteList.

eq addPendingWrites((TID LP SID OL) :: LEL, PWL) = addPendingWrites(LEL, pw(LP, idle, OL) :: PWL).

eq addPendingWrites(noEntries, PWL) = PWL.

--- If catchup–response timed out and we have sites to try, try the next site

rl [restartCatchup]:
< THIS : Site | entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
localTransactions : < TID : Transaction |
operations : cr(entity(EG,N)) :: OL, status : transTimer(0),
readState : catchingUp(entity(EG, N), SIL :: SID, LPL) ; RSTATE > LOCALTRANS

> =>
< THIS : Site | entityGroups : < EG : EntityGroup | > EGROUPS,
localTransactions : < TID : Transaction |
operations : cr(entity(EG,N)) :: OL, status : transTimer(defTimeout),
readState : catchingUp(entity(EG, N), SIL, LPL) ; RSTATE > LOCALTRANS

(uniCast catchupRequest(EG, TID, LPL) from THIS to SID).

--- If no sites are available, restart the majority read

rl [restartCatchup]:
< THIS : Site | entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
localTransactions : < TID : Transaction |
operations : cr(entity(EG,N)) :: OL, status : transTimer(0),
readState : catchingUp(entity(EG, N), emptySiteIdList, LPL) ; RSTATE > LOCALTRANS

> =>
< THIS : Site | entityGroups : < EG : EntityGroup | > EGROUPS,
Listing B.7  megastore_timing.rtmaude

((tomsd MEGASTORE–TIMING is
  inc MEGASTORE–SETUP .
  inc UPDATES .

  var SID : SiteId .
  var TID : TransId .
  var EG : EntityGroupId .
  var SIS : SiteIdSet .
  var EGRS : EntityGroupReplicaSet .

  var T1 T2 T3 EXP : Time .
  var TI1 TI2 : TimeInf .

  var N : Nat .
  var OL : OperationList .

  var EGROUPS LOCALTRANS REMAININGTRANS : Configuration .

  var TS : TransStatus .

  var PN PW' : Propnum .
  var DPN : DefPropnum .
  var PWL : PendingWriteList .
  var LE : LogEntry .
  var EG : EntityGroupId .
  var LP : LogPosition .
  var CES : EntGroupLogPosPairSet .

  var PSTATE : PaxosStateSet .

  ***( Sites ***)
  eq mte(< SID : Site | coordinator : CES, entityGroups : EGROUPS, localTransactions : LOCALTRANS >) =
    min (mte(EGROUPS), mteTrans(EGROUPS, LOCALTRANS, LOCALTRANS)) .
  eq delta(< SID : Site | entityGroups : EGROUPS, localTransactions : LOCALTRANS >, T1) =
    < SID : Site | entityGroups : delta(EGROUPS, T1), localTransactions : delta(LOCALTRANS, T1) > .

  ***( Transactions ***)

  op mteTrans : Configuration Configuration Configuration --> TimeInf .

  (multicast majorityRead(EG, TID) from THIS to getSites(EGRS) setminus THIS).

  op setValidated : EntityGroupId LogPosition EntGroupLogPosPairSet --> EntGroupLogPosPairSet .
  ceq setValidated(EG, lpos(N1), eglp(EG, lpos(N2)) ; CES) = eglp(EG, lpos(N1)) ; CES if (N2 >= N1).
  ceq setValidated(EG, lpos(N1), eglp(EG, lpos(N2)) ; CES) = eglp(EG, lpos(N1)) ; CES if (N2 < N1).
  ceq setValidated(EG, lpos(N1), invalidCstate(EG, lpos(N2)) ; CES) = invalidCstate(EG, lpos(N2)) ; CES if (N1 < N2).
  ceq setValidated(EG, lpos(N1), invalidCstate(EG, lpos(N2)) ; CES) = eglp(EG, lpos(N1)) ; CES if (N2 <= N1).
  eq setValidated(EG, lpos(N1), CES) = eglp(EG, lpos(N1)) ; CES [otherwise].

endtom)
--- Determine mte if TS == idle

cq mteTrans(<EG : EntityGroup | pendingWrites : emptyPWLList > EGROUPS, LOCALTRANS,
  <TID : Transaction | operations : cr(entity(EG,N)) :: OL, status : idle > REMAININGTRANS) = 0 if not
  inConflictWithRunning(EG, removeTid(TID, LOCALTRANS)).

cq mteTrans(EGROUPS, LOCALTRANS,
  <TID : Transaction | operations : w(EID:EntityId, EVAL:EntityValue) :: OL, status : idle >
  REMAININGTRANS) = 0.

cq mteTrans(EGROUPS, LOCALTRANS,
  <TID : Transaction | operations : emptyOpList, status : idle > REMAININGTRANS) = 0.

cq mteTrans(EGROUPS, LOCALTRANS,
  <TID : Transaction | status : executing(LP, T1) > REMAININGTRANS) = min(T1, mteTrans(EGROUPS,
  LOCALTRANS, REMAININGTRANS)).

cq mteTrans(EGROUPS, LOCALTRANS,
  <TID : Transaction | status : transTimer(T1) > REMAININGTRANS) =
  min(T1, mteTrans(EGROUPS, LOCALTRANS, REMAININGTRANS)).

cq mteTrans(EGROUPS, LOCALTRANS,
  <TID : Transaction | paxosState : PSTATE, status : invalidating > REMAININGTRANS) =
  min(mte(PSTATE), mteTrans(EGROUPS, LOCALTRANS, REMAININGTRANS)).

cq mteTrans(EGROUPS, LOCALTRANS, REMAININGTRANS) = INF [wise].

op mte : PaxosStateSet -> Time.

cq mte(acceptLeader(EG, LE, SID, T1) : PSTATE) = min(T1, mte(PSTATE)).

cq mte(acceptAll(EG, LE, PN, SIS, T1) : PSTATE) = min(T1, mte(PSTATE)).

cq mte(prepare(EG, LE, PN, DPN, SIS, T1) : PSTATE) = min(T1, mte(PSTATE)).

cq mte(restartPrepare(EG, LE, T1) : PSTATE) = min(T1, mte(PSTATE)).

cq mte(invalidating(EG, LE, PN, SIS, T1) : PSTATE) = min(T1, mte(PSTATE)).

cq mte(PSTATE) = INF [wise].

op hasPrepareQuorum : Configuration PaxosState -> Bool.

cq hasPrepareQuorum(<EG : EntityGroup | replicas : EGRS > EGROUPS, prepare(EG, LE, PN, PN', SIS, EXP)
  PSTATE) =
  true if REPLICA:SiteIdSet := getSites(EGRS) \ hasQuorum[size(SIS), REPLICA:SiteIdSet].

op hasPrepareQuorum(EGROUPS, PSTATE) = false [wise].

op removeTid : TransId Configuration -> Configuration.

eq removeTid(TID, <TID : Transaction | > REMAININGTRANS) = REMAININGTRANS.

eq removeTid(TID, LOCALTRANS) = LOCALTRANS [wise].

eq delta(<TID : Transaction | status : executing(LP, T2) >, T1) = <TID : Transaction | status :
  executing(LP, T2 minus T1) >.

eq delta(<TID : Transaction | status : transTimer(T2) >, T1) = <TID : Transaction | status :
  transTimer(T2 minus T1) >.

eq delta(<TID : Transaction | paxosState : PSTATE, status : in-paxos >, T1) =
  <TID : Transaction | paxosState : delta(PSTATE, T1) >.

eq delta(<TID : Transaction | status : TS >, T1) = <TID : Transaction | > [wise].

op delta : PaxosStateSet Time -> PaxosStateSet.

eq delta(acceptLeader(EG, LE, SID, T2) : PSTATE, T1) =
  acceptLeader(EG, LE, SID, T2 minus T1) ; delta( PSTATE, T1).

eq delta(acceptAll(EG, LE, PN, SIS, T2) : PSTATE, T1) =
  acceptAll(EG, LE, PN, SIS, T2 minus T1) ; delta( PSTATE, T1).

eq delta(prepare(EG, LE, PN, DPN, SIS, T2) : PSTATE, T1) =
  prepare(EG, LE, PN, DPN, SIS, T2 minus T1) ; delta( PSTATE, T1).

eq delta(restartPrepare(EG, LE, T2) : PSTATE, T1) =
  restartPrepare(EG, LE, T2 minus T1) ; delta(PSTATE, T1).

eq delta(invalidating(EG, LE, PN, SIS, T2) : PSTATE, T1) =
  invalidating(EG, LE, PN, SIS, T2 minus T1) ; delta(PSTATE, T1).

eq delta(emptyPaxosState, T1) = emptyPaxosState.

***( Entity groups ***)
Listing B.8  updates_fault_handling.rtaude

{(tomod UPDATE–FAULT–HANDLERS is
  inc UPDATES .
  var EID : EntityId .
  vars N N' N2 : Nat .
  var NS : NatSet .
  vars T EXP : Time .
  vars PN PN' PROPNUM : Propnum .
  var PXSID : Nat .
  var DPN SEEN–PROPNUM : DefPropnum .
  var EG : EntityGroupId .
  var EGROUPS : Configuration .
  vars TID TID' TID1 TID2 TID3 : TransId .
  var TIS : TransIdSet .
  vars SID SID' MSID1 MSID2 SENDER THIS : SiteId .
  var SIS SIS–FAILED REPLICAS : SiteIdSet .
  var EGRS : EntityGroupReplicaSet .
  var PSTATE : PaxosStateSet .
  var LOCALTRANS : Configuration .
  var OL OL' OL1 OL2 : OperationList .
  var OP : Operation .
  var PROPSET : PaxosProposalSet .
  vars LEL LEL' : LogEntryList .
  var LE LE' NEW–LE : LogEntry .
  var LP LP' : LogPosition .
  var PWL : PendingWriteList .
  var CES : EntGroupLogPosPairSet .
  var READS : EntitySet .
  var WRITES : OperationList .
  var COMPLETE : Bool .

  ***( Messages involved in presence of errors ***)
  op prepareAllReq : TransId EntityGroupId LogPosition OperationList Propnum --> MsgContent .
    −−− Used when acceptor has an existing proposal
  op prepareAllRsp : TransId EntityGroupId LogEntry Propnum Propnum --> MsgContent .
    −−− Used when acceptor does not has an existing proposal
  op invalidateCoordinator : EntityGroupId LogPosition --> MsgContent .
  op invalidateConfirmed : EntityGroupId LogPosition --> MsgContent .

  ***( Paxos phase 1: Leader election ***)
    −−− We did not get any response from the leader. Run phase 1 of Paxos.
  crl [PlacceptLeaderFailureRsp] :
    < THIS : Site |
localTransactions : < TID : Transaction | paxosState : acceptLeader(EG, (TID LP MSID1 OL1), MSID2, 0) ; PSTATE,
status : in–poxa > LOCALTRANS,
entityGroups : < EG : EntityGroup | proposals : PROPSET,
replicas : egr(THIS, PXSID, LP’) ; EGRS > EGROUPS
>
=>
< THIS : Site |
localTransactions : < TID : Transaction |
paxosState : prepare(EG, (TID LP MSID1 OL1), PN, noPropnum, emptyDidSet, defTimeout) ; PSTATE,
status : in–poxa > LOCALTRANS,
entityGroups : < EG : EntityGroup | > EGROUPS

(msg prepareAllReq(TID, EG, LP, OL1, PN) from THIS to REPLICAS)
if REPLICA$ := getSites(egr(THIS, PXSID, LP’) ; EGRS) /
PN := createPropnum(getCurPropnum(LP, PROPSET), size(REPLICAS), PXSID).

op getCurPropnum : LogPosition PaxosProposalSet -> DefPropnum.

eq getCurPropnum(LP, proposal(SID, TID, LP, OL, PN) ; PROPSET) =
maxPN(PN, getCurPropnum(LP, PROPSET)).

eq getCurPropnum(LP, accepted(SID, (TID1 LP MSID1 OL), PN) ; PROPSET) =
maxPN(PN, getCurPropnum(LP, PROPSET)).

eq getCurPropnum(LP, PROPSET) = noPropnum [owise].

--- Use the method described in footnote on page 4 of Chandra 2007 ("Paxos made live")
--- to ensure every proposal has a unique PN

op createPropnum : DefPropnum Nat Nat -> Propnum.

eq createPropnum(PN, N, PXSID) =
if ((PN rem N) >= PXSID) then
(N + (a (PN quo N)) + (PXSID rem N))
else
(PN + sd(PXSID, (PN rem N)).
fi.

eq createPropnum(noPropnum, N, PXSID) = 1 + PXSID.

op maxPN : Propnum DefPropnum -> Propnum.

ceq maxPN(PN, PN’) = PN if (PN >= PN’).

ceq maxPN(PN, PN’) = PN’ if (PN < PN’).

eq maxPN(PN, noPropnum) = PN.

--- Receive a prepare–message with a previous proposal for this log position

crl [P2rcvPrepareAllReq] : (msg prepareAllReq(TID1, EG, LP, OL1, PROPNUM) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | transactionLog : LEL, proposals : accepted(SID, (TID2 LP MSID1 OL2), PN) ; PROPSET > EGROUPS
>
=>
< THIS : Site |
entityGroups : < EG : EntityGroup | proposals : accepted(SID, (TID2 LP MSID1 OL2), PROPNUM) ; PROPSET > EGROUPS
>
(msg prepareAllRsp(TID1, EG, (TID2 LP MSID1 OL2), PROPNUM, PN) from THIS to SENDER)
if PROPNUM > PN.

crl [P2rcvPrepareAllReq] :
(msg prepareAllReq(TID1, EG, LP, OL, PROPNUM) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | transactionLog : LEL,
proposals : proposal(SID, TID2, LP, OL2, PN) ; PROPSET > EGROUPS
>
\textbf{B Real-Time Maude Model of Megastore}

\begin{verbatim}
=>
  < THIS : Site |
  entityGroups : < EG : EntityGroup | proposals : proposal(SID, TID2, LP, OL2, PROPNUM) ; PROPSET >
  EGROUPS
>
  \text{(uniCast prepareAllReq(TID1, EG, (TID2 LP SID OL2), PROPNUM, PN) from \text{THIS} to SENDER)}

\textbf{--- If we receive a proposal with an obsolete number.}
\textbf{--- then we can safely ignore it}

crl \text{[PF2.lrcvPrepareAllReqWithObsoletePropnum]} :
  (msg prepareAllReq(TID, EG, LP, OL, PROPNUM) from SENDER to \text{THIS})
  < THIS : Site |
  entityGroups : < EG : EntityGroup | transactionLog : LEL, proposals : PROPSET >
  EGROUPS
>
=>
  < THIS : Site |
  entityGroups : < EG : EntityGroup | >
  EGROUPS
>
  \text{(uniCast prepareAllRsp(TID1, EG, (TID2 LP SID OL2), PROPNUM, PN) from \text{THIS} to SENDER)}
if \text{PROPNUM} > PN .

crl \text{[PF2.lrcvPrepareAllReqForApplied]} :
  (msg prepareAllReq(TID, EG, LP, OL, PROPNUM) from SENDER to \text{THIS})
  < THIS : Site |
  entityGroups : < EG : EntityGroup | transactionLog : LEL, proposals : PROPSET >
  EGROUPS
>
=>
  < THIS : Site |
  entityGroups : < EG : EntityGroup | >
  EGROUPS
>
  \text{(uniCast signalConflict(TID, EG, LP) from \text{THIS} to SENDER)}
if containsLogPosition(LP, LEL) .

\textbf{op conflictingProposal : EntityGroupId LogPosition Propnum PaxosProposalSet } \rightarrow \text{ Bool} .
\textbf{ceq conflictingProposal(EG, LP, PN, accepted(SID, (TID LP MSID1 OL), PN')) ; PROPSET) =}
  \text{true if PN'} >= PN .
\textbf{ceq conflictingProposal(EG, LP, PN, proposal(SID, TID, LP, OL, PN') ; PROPSET) =}
  \text{true if PN'} >= PN .
\textbf{eq conflictingProposal(EG, LP, PN, PROPSET) = false [owise] .}

\textbf{--- Receive a prepare-message without a previous proposal for this log position}

crl \text{[PF3rvcPrepareAllReq]} :
  (msg prepareAllReq(TID1, EG, LP, OL, PROPNUM) from SENDER to \text{THIS})
  < THIS : Site |
  entityGroups : < EG : EntityGroup | transactionLog : LEL, proposals : PROPSET >
  EGROUPS
>
=>
  < THIS : Site |
  entityGroups : < EG : EntityGroup | proposals : proposal(SENDER, TID1, LP, OL, PROPNUM) ; PROPSET >
  EGROUPS
>
  \text{(uniCast prepareAllRsp(TID1, EG, LP, PROPNUM) from \text{THIS} to SENDER)}
if not (containsProposal(EG, LP, PROPSET) or containsLogPosition(LP, LEL)) .

\textbf{op containsProposal : EntityGroupId LogPosition PaxosProposalSet } \rightarrow \text{ Bool} .
\textbf{eq containsProposal(EG, LP, accepted(SID, (TID LP MSID1 OL), PN')) ; PROPSET) = true} .
\textbf{eq containsProposal(EG, LP, proposal(SID, TID, LP, OL, PN') ; PROPSET) = true} .
\textbf{eq containsProposal(EG, LP, PROPSET) = false [owise] .}

\textbf{op removePreviousProposal : LogPosition Propnum PaxosProposalSet } \rightarrow \text{ PaxosProposalSet} .
\textbf{eq removePreviousProposal(LP, PN, proposal(SID, TID, LP, OL, PN') ; PROPSET) = PROPSET} .
\end{verbatim}
eq removePreviousProposal(LP, PN, PROPSET) = PROPSET \{\text{ovise}\}.

r1 [P4rcvPrepareAllRspWithProposal] :
(msg prepareAllRsp(TID, EG, (TID2 LP MSID2 OL2), PROPNUM, PN) from SENDER to THIS)
< THIS : Site
  entityGroups : < EG : EntityGroup | transactionLog : LEL' ; proposals :
  PROPSET > EGROUPS
  >
=>
< THIS : Site |
  entityGroups : < EG : EntityGroup | > EGROUPS
> (if (TID2 /= TID1) then (unicast signalConflict(TID1, EG, LP) from THIS to SENDER) else none fi).

crl [P4rcvPrepareAllRspWithValue] :
(msg prepareAllRsp(TID, EG, (TID2 LP MSID1 OL1), PROPNUM, PN) from SENDER to THIS)
< THIS : Site |
  entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
  localTransactions : < TID : Transaction | paxosState : prepare(EG, (TID3 LP MSID2 OL2), PROPNUM, SEEN=PROPNUM, SIS, EXP) ; PSTATE, status : in=paxos > LOCALTRANS
  >
=>
< THIS : Site |
  localTransactions : LOCALTRANS
  (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
   < TID : Transaction |
   paxosState : acceptAll(EG, NEW=LE, PROPNUM, THIS, defTimeout) ; PSTATE, status : in=paxos >
   else
   < TID : Transaction |
   paxosState : prepare(EG, NEW=LE, PROPNUM, maxPn(PN, SEEN=PROPNUM), (SIS ; SENDER), EXP) ; PSTATE,
   status : in=paxos >
   fi)
> (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
  multiCast acceptAllRsp(TID, EG, NEW=LE, PROPNUM) from THIS to REPLICAS
  else none fi)
if REPLICAS := getSites(EGRS) \/
  NEW=LE := chooseValue(PN, SEEN=PROPNUM, (TID2 LP MSID1 OL1), (TID3 LP MSID2 OL2)) .

op chooseValue : PropNum DefPropNum LogEntry LogEntry \rightarrow LogEntry .
eq chooseValue(PN, noPropNum, LE, LE') = LE .
eq chooseValue(PN, PN', LE, LE') = if PN ’ PN then LE else LE' fi .

crl [P4rcvPrepareAllRspWithoutValue] :
(msg prepareAllRsp(TID, EG, LP) from SENDER to THIS)
< THIS : Site |
  entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
  localTransactions : < TID : Transaction | paxosState : prepare(EG, (TID2 LP MSID1 OL1), PN, SEEN=PROPNUM, SIS, EXP) ; PSTATE,
  status : in=paxos > LOCALTRANS
  >
=>
< THIS : Site |
  localTransactions : LOCALTRANS
  (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
   < TID : Transaction |
   paxosState : acceptAll(EG, (TID2 LP MSID1 OL1), PN, THIS, defTimeout) ; PSTATE >
   else
   < TID : Transaction |
\[
\text{paxosState : prepare(EG, (TID2 LP MSID1 OL1), PN, SEEN—PROPNUM, (SIS ; SENDER), EXP) ; PSTATE > fi)}
\]

\[
> \quad > (\text{if hasQuorum(size(SIS ; SENDER), REPLICAS) then multiCast acceptAllReq(TID, EG, (TID2 LP MSID1 OL1), PN) from THIS to REPLICAS else none fi})\]
\[
\text{if REPLICAS := getSites(EGRS).}
\]

\[
\text{crl [PF1rcvObsoletePrepareRspWithoutValue] : (msg prepareAllRsp(TID, EG, LP, PN) from SENDER to THIS)}\]
\[
< \text{THIS : Site} | \quad \text{localTransactions : LOCALTRANS}>
\]
\[
\Rightarrow > \quad < \text{THIS : Site} | \text{localTransactions : LOCALTRANS} > \quad \text{if (not inPrepare(TID, EG, LP, PROPNUM, LOCALTRANS)).}
\]

\[
\text{crl [PF2rcvObsoletePrepareRspWithValue] : (msg prepareAllRsp(TID, EG, (TID' LP MSID1 OL1), PROPNUM, PN) from SENDER to THIS)}\]
\[
< \text{THIS : Site} | \quad \text{localTransactions : LOCALTRANS}>
\]
\[
\Rightarrow > \quad < \text{THIS : Site} | \text{localTransactions : LOCALTRANS} > \quad \text{if (not inPrepare(TID, EG, LP, PROPNUM, LOCALTRANS)).}
\]

\[
\text{op inPrepare : TransId EntityGroupId LogPosition PropNum Configuration} —> \text{Bool} .
\]
\[
\text{ceq inPrepare(TID, EG, LP, PN, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).}
\]
\[
\text{eq inPrepare(TID, EG, LP, PN, \text{< TID : Transaction | paxosState : prepare(EG, (TID LP MSID1 OL1), PN, SEEN—PROPNUM, SIS, EXP) ; PSTATE > LOCALTRANS}) = true .}
\]
\[
\text{eq inPrepare(TID, EG, LP, PN, \text{< TID : Transaction | paxosState : PSTATE > LOCALTRANS}) = false [otherwise].}
\]

\[
\text{--- NOTE: This is not included in the Megastore paper, but is our interpretation}
\]

\[
\text{crl [PF3failedPrepareAllReq] :}
\]
\[
< \text{THIS : Site} | \quad \text{entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,}
\quad \text{localTransactions : < TID : Transaction |}
\quad \text{paxosState : prepare(EG, (TID2 LP MSID1 OL1), PROPNUM, SEEN—PROPNUM, SIS, 0) ; PSTATE, status : in—paxos > LOCALTRANS}>
\]
\[
\Rightarrow > \quad < \text{THIS : Site} | \quad \text{localTransactions : < TID : Transaction | paxosState : restartPrepare(EG, (TID2 LP MSID1 OL1), N) ; PSTATE, status : in—paxos > LOCALTRANS,}
\quad \text{entityGroups : < EG : EntityGroup | > EGROUPS}>
\]
\[
\text{if (not hasQuorum(size(SIS), getSites(EGRS))) \text{ /\ N ; NS := possibleBackoffs .}
\]
\[
\text{op possibleBackoffs : —> Nat } .
\]

\[
***(\text{Paxos phase 2: Accept **})
\]

\[
\text{--- If we receive another accept request for this log position, accept it if and only if it is the same (re—sent) proposal, or}
\]
\[
\text{the new proposal number is higher than the previous}
\]

\[
\text{crl [A3rcvAcceptAllReqSubseq] : (msg acceptAllReq(TID, EG, (TID1 LP MSID1 OL1), PN) from SENDER to THIS)}
\]
--- If we receive another accept request for this log position with a lower proposal number than the previous, we discard the message.

--- If we receive an accept request for an already logged transaction, discard the message.

--- Ignore an unexpected accept response.
> if (not inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS)) .

op inAcceptAll : TransId EntityGroupId LogPosition Propnum Configuration -> Bool .
ceq inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS) .
eq inAcceptAll(TID, EG, LP, PROPNUM, < TID : Transaction | paxosState : acceptAll(EG, (TID' LP SID OL), PROPNUM, SIS, 0) ; PSTATE > LOCALTRANS) = true .
eq inAcceptAll(TID, EG, LP, PROPNUM, < TID : Transaction | paxosState : PSTATE > LOCALTRANS) = false [owise] .

--- Only some replicas responded, but sufficient for a quorum. Send invalidate message to others

crl [A6initInvalidation] :
< THIS : Site |
  localTransactions : < TID : Transaction | paxosState : acceptAll(EG, (TID' LP SID OL), PROPNUM, SIS, 0) ; PSTATE, status : in-paxos > LOCALTRANS, entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS
>
=>
< THIS : Site |
  localTransactions : < TID : Transaction | paxosState : invalidating(EG, (TID' LP SID OL), PROPNUM, REPLICAS setminus SIS, defTimeout) ; PSTATE > LOCALTRANS, entityGroups : < EG : EntityGroup | > EGROUPS
>
(multiCast invalidateCoordinator(EG, LP) from THIS to REPLICAS setminus SIS)
if REPLICAS := getSites(EGRS) \ hasQuorum(size(SIS), REPLICAS) .

rl [A7invalidateCoordinator] :
  (msg invalidateCoordinator(EG, lpos(N)) from SENDER to THIS)
< THIS : Site |
  coordinator : CES >
=>
< THIS : Site |
  coordinator : applyInvalidate(EG, lpos(N), CES) >
  (uniCast invalidateConfirmed(EG, lpos(N)) from THIS to SENDER) .

op applyInvalidate : EntityGroupId LogPosition EntGroupLogPosPairSet -> EntGroupLogPosPairSet .
ceq applyInvalidate(EG, lpos(N), eglp[EG, lpos(N')] ; CES) = invalidCstate(EG, lpos(N)) ; CES if (N' <= N) .
ceq applyInvalidate(EG, lpos(N), invalidCstate(EG, lpos(N')) ; CES) = invalidCstate(EG, lpos(N)) ; CES if (N' < N) .
ceq applyInvalidate(EG, lpos(N), eglp[EG, lpos(N')] ; CES) = eglp[EG, lpos(N')] ; CES if (N' > N) .
eq applyInvalidate(EG, lpos(N), CES) = CES [owise] .

crl [A8rcvInvalidateConfirmed] :
  (msg invalidateConfirmed(EG, LP) from SENDER to THIS)
< THIS : Site |
  entityGroups : EGROUPS, localTransactions : < TID : Transaction |
  paxosState : invalidating(EG, (TID' LP SID OL), PROPNUM, SIS-Failed, EXP) ; PSTATE, reads : READS, writes : WRITES > LOCALTRANS
>
=>
< THIS : Site | localTransactions : LOCALTRANS
if COMPLETE then
  (if allEGSAccepted(PSTATE) then none
else
  < TID : Transaction | paxosState : acceptedPS(EG, (TID' LP SID OL), PROPNUM) ; PSTATE >
fi)
else
  < TID : Transaction |
    paxosState : invalidating(EG, (TID' LP SID OL), PROPNUM, SIS - FAILED setminus SENDER, EXP); PSTATE >
fi)

if (COMPLETE and allEGSAccepted(PSTATE)) then
  createApplyMessages(SID, EGROUPS, acceptedPS(EG, (TID' LP SID OL), PROPNUM); PSTATE)
  (if (TID == TID') then
    notifyCommit(THIS, TID, eglp(EG, LP); getEntGroupLogPosPair(PSTATE), READS, WRITES)
  else (notifyConflictAbort(THIS, TID, READS, WRITES))
  fi)
else none
fi)

if COMPLETE := ((SIS - FAILED setminus SENDER) == emptyOidSet).

crl [A8rcvInvalidateConfirmedObsolete] :
  (msg invalidateConfirmed(EG, LP) from SENDER to THIS)
  < THIS : Site |
    localTransactions : LOCALTRANS
  >
  =>
  < THIS : Site |
    localTransactions : LOCALTRANS
  >= if not inInvalidate(EG, LP, LOCALTRANS).

op inInvalidate : TransId LogPosition Configuration -> Bool.

    eq inInvalidate(EG, LP, < TID : Transaction | paxosState : invalidating(EG, (TID' LP MSID1 OL), PROPNUM, SIS, EXP); PSTATE > LOCALTRANS) = true .

    eq inInvalidate(EG, LP, LOCALTRANS) = false [wise].

rl [AF6resendInvalidate] :
  < THIS : Site |
    localTransactions : < TID : Transaction | paxosState : invalidating(EG, (TID' LP SID OL), PROPNUM, SIS, 0); PSTATE, status : in -- paxos > LOCALTRANS,
    entityGroups : < EG : EntityGroup | > EGROUPS
  >
  =>
  < THIS : Site |
    localTransactions : < TID : Transaction | paxosState : invalidating(EG, (TID' LP SID OL), PROPNUM, SIS, defTimeout); PSTATE > LOCALTRANS,
    entityGroups : < EG : EntityGroup | > EGROUPS

  (multiCast invalidateCoordinator(EG, LP) from THIS to SIS).

--- Timeout without quorum -- failure handling according to #3 in 4.6.3

crl [restartPrepare] :
  < THIS : Site |
    localTransactions : < TID : Transaction | paxosState : acceptAll(EG, (TID' LP MSID1 OL), PROPNUM, SIS, 0); PSTATE,
     status : in -- paxos > LOCALTRANS,
    entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS
  >
  =>
  < THIS : Site |
localTransactions : < TID : Transaction | paxosState : restartPrepare(EG, (TID', LP MSID1 OL1), N ; PSTATE > LOCALTRANS,
               entityGroups : < EG : EntityGroup | EGROUPS
               >
            if not hasQuorum(size(SIS), getSites(EGRS)) \ N ; NS := possibleBackoffs .

    crl [AF4initiatePrepare] :
    < THIS : Site |
             localTransactions : < TID : Transaction | paxosState : restartPrepare(EG, (TID', LP MSID1 OL1), 0 ; PSTATE,
            status : in-paxos > LOCALTRANS,
               entityGroups : < EG : EntityGroup | EGROUPS
               >
            =>
            < THIS : Site |
             localTransactions : < TID : Transaction | paxosState : prepare(EG, (TID', LP MSID1 OL1), PN, noPropnum, emptyOidSet, defTimeout ; PSTATE,
            status : in-paxos > LOCALTRANS,
               entityGroups : < EG : EntityGroup | EGROUPS
               >
            (multiCast prepareAllReq(TID, EG, LP, OL1, PN) from THIS to REPLICAS)
            if REPLICA := getSites(egr(THIS, PXSID, LP') ; EGRS) \ PN := createPropnum(getCurPropnum(LP, PROPSET), size(REPLICAS), PXSID) .

    ***( After Paxos-consensus: Apply ***)

    --- In case of some previous error, we allow processing "out of order"
    crl [APP3:iniUpdatesInvalidated] :
          (msg applyReq(TID, EG, LP, PROPNUM) from SENDER to THIS)
          < THIS : Site |
            coordinator : CES,
            entityGroups : < EG : EntityGroup | transactionLog : LEL, pendingWrites : PWL,
            proposals : accepted(SID, (TID2 LP MSID1 OL), PN') ; PROPSET >
            EGROUPS,
            localTransactions : LOCALTRANS
          >
          =>
          < THIS : Site |
            coordinator : CES,
            entityGroups : < EG : EntityGroup | transactionLog : insertLogEntrySorted((TID2 LP MSID1 OL), LEL),
            pendingWrites : pw(LP, idle, OL) ; PWL,
            proposals : removeProposals(LP, PROPSET) > EGROUPS,
            localTransactions : removeOthersForLogPosition(EG, LP, LOCALTRANS)
          >
          (sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, LP, LOCALTRANS)))
          if not containsEntityGroupId(EG, CES) \ ((PROPNUM == PN') or (TID == TID2)) .

    op insertLogEntrySorted : LogEntry LogEntryList -> LogEntryList .
    eq insertLogEntrySorted((TID1 lpos(N1) MSID1 OL1), (TID2 lpos(N2) MSID2 OL2) :: LEL) =
            if (N1 < N2) then
               (TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2 OL2) :: LEL
            else
               (TID2 lpos(N2) MSID2 OL2) :: insertLogEntrySorted((TID1 lpos(N1) MSID1 OL1), LEL) fi .
    eq insertLogEntrySorted(LE, noEntries) = LE .

    --- Here, we discover that this site is not up-to-date upon receiving an apply
    crl [APP3:iniOutOfOrder] :
          (msg applyReq(TID, EG, lpos(N1), PROPNUM) from SENDER to THIS)
Listing B.9 transaction-history.rtmaude

```maude
< THIS : Site |
  coordinator : eglp(EG, lpos(N2)) ; CES, 
  entityGroups : < EG : EntityGroup | transactionLog : LEL, 
                 pendingWrites : PWL, 
                 proposals : accepted(SID, (TID2 lpos(N1) MSID1 OL), PW') ; 
                 PROPSET > EGROUPS, 
  localTransactions : LOCALTRANS > 
> 
\Rightarrow
< THIS : Site |
  coordinator : applyInvalidate(EG, lpos(s N1), CES), 
  entityGroups : < EG : EntityGroup | transactionLog : insertLogEntrySorted((TID2 lpos(N1) 
                     MSID1 OL), LEL), 
                 pendingWrites : pw(lpos(N1), idle, OL) :: PWL, 
                 proposals : removeProposals(lpos(N1), PROPSET) > EGROUPS, 
  localTransactions : removeOthersForLogPosition(EG, lpos(N1), LOCALTRANS) > 
  (sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N1), LOCALTRANS))) 
if N1 > s N2 \ (\ (PROPNUM == PW') or (TID == TID2)) .
--- If we receive an apply req for which we do not have an accept, we invalidate the coordinator 
\cr[APP3.2InitUpdatesWithoutAccept] : 
  (msg applyReq(TID, EG, LP, PROPNUM) from SENDER to THIS) 
< THIS : Site |
  coordinator : CES, 
  entityGroups : < EG : EntityGroup | proposals : PROPSET > EGROUPS > 
=\Rightarrow
< THIS : Site | coordinator : applyInvalidate(EG, LP, CES) > 
if not containsAccept(SENDER, TID, LP, PROPNUM, PROPSET) .

\op containsAccept : SiteId TransId LogPosition Propnum PaxosProposalSet \rightarrow \Bool .
\eq containsAccept(SID, TID, LP, PN, accepted(SID, (TID LP MSID1 OL), PN) ; PROPSET) = true .
\eq containsAccept(SID, TID, LP, PN, accepted(SID, (TID LP MSID1 OL), PN') ; PROPSET) = true .
\eq containsAccept(SID, TID, LP, PW, PROPSET) = false [otherwise] .
--- With competing leaders, we might receive two apply messages for the same transaction 
\cr[APP4InitUpdatesForAppliedTrans] : 
  (msg applyReq(TID, EG, LP, PROPNUM) from SENDER to THIS) 
< THIS : Site |
  coordinator : CES, 
  entityGroups : < EG : EntityGroup | transactionLog : LEL, 
    proposals : PROPSET > EGROUPS > 
=\Rightarrow
< THIS : Site |
  coordinator : CES, 
  entityGroups : < EG : EntityGroup | transactionLog : LEL, 
    proposals : removeProposals(LP, PROPSET) > EGROUPS > 
  if containsLogPosition(LP, LEL) .

\op containsLogPosition : LogPosition LogEntryList \rightarrow \Bool .
\eq containsLogPosition(LP, LEL :: (TID LP MSID1 OL) :: LEL') = true .
\ceq containsLogPosition(lpos(N), (TID lpos(N') MSID1 OL) :: LEL') = true if (N < N') .
\eq containsLogPosition(LP, LEL) = false [otherwise] .
endtm)
```
There is a path from transaction T1 and T2 if and only if T1 and T2 are in conflict.

The basic definition of a conflict graph is:

- Whenever transaction T2 reads an item I:
  * create an edge from any previous updater of I
- Whenever a transaction T1 writes an item I:
  * create an edge from any previous updater TU of I to T1
  * create an edge from any previous reader TR of to T1

```
(tomod TRANSACTION—HISTORY is
  inc MEGASTORE—SETUP .
  var EID : EntityId .
  vars N N’ : Nat .
  vars SG SG’ : SerGraph .
  var LP : LogPosition .
  vars TID TID’ : TransId .
  vars TIS1 TIS2 TIS3 : TransIdSet .
  var TH : Oid .
  var T1 : Time .
  vars TOPS READERS WRITERS : TransOpSet .
  var E : Edge .

  class TransactionHistory |
    graph : SerGraph ,
    readers : TransOpSet ,
    writers : TransOpSet .

eq nte(< TH : TransactionHistory | >) = INF .
eq delta(< TH : TransactionHistory | >, T1) = < TH : TransactionHistory | > .

  sort SerGraph .
  sort Edge .
  subsort Edge < SerGraph .
  op <_,_> : TransId TransId -- Edge [ctor] .
  op emptyGraph : -- SerGraph [ctor] .
  op _-- : SerGraph SerGraph -- SerGraph [ctor assoc comm id: emptyGraph] .
  eq (SG ; E) ; (SG’ ; E) = (SG ; E) ; SG’ .

  sorts TransOp TransOpSet .
  op op : TransId EntityId LogPosition -- TransOp [ctor] .

  op predecessors : EntityId LogPosition TransOpSet -- TransIdSet .
  ceq predecessors(EID, lpos(N), op(TID, EID, lpos(N’)) ; TOPS) = TID ; predecessors(EID, lpos(N), TOPS)
    if N > N’ .
  ceq predecessors(EID, lpos(N), op(TID, EID, lpos(N’)) ; TOPS) = predecessors(EID, lpos(N), TOPS)
    if N <= N’ .
  eq predecessors(EID, lpos(N), TOPS) = emptyOidSet [owise] .

  op successors : EntityId LogPosition TransOpSet -- TransIdSet .
  ceq successors(EID, lpos(N), op(TID, EID, lpos(N’)) ; TOPS) = TID ; successors(EID, lpos(N), TOPS)
    if N < N’ .
  ceq successors(EID, lpos(N), op(TID, EID, lpos(N’)) ; TOPS) = successors(EID, lpos(N), TOPS)
    if N = N’ .
  eq successors(EID, lpos(N), TOPS) = emptyOidSet [owise] .
```
Listing B.10 network_model.mc.rtmaude

{tomod NETWORK—MODEL is
  \( \text{inc} \ \text{MSG—WRAPPERs} \)
  \( \text{inc} \text{ TIMED—BEHAVIOR} \).

  \( \text{var} \ N : \text{Nat} \).
  \( \text{var} \ NS : \text{NatSet} \).
  \( \text{var} \ MC : \text{MsgContent} \).
  \( \text{vars} \ SID \ \text{SID'} : \text{SiteId} \).

op getCreator : EntityId LogPosition TransOpSet --> TransId.
\( \text{eq} \ \text{getCreator}(\text{EID}, \text{LP}, \text{op}(\text{TID}, \text{EID}, \text{LP}); \text{TOPS}) = \text{TID} \).

op addWrite : TransId EntityId LogPosition Object --> Object.
op addRead : TransId EntityId LogPosition Object --> Object.
\( \text{eq} \ \text{addWrite}(\text{TID}, \text{EID}, \text{LP}, \text{< TH : TransactionHistory | graph : SG, readers : READERS, writers : WRITERS >}) = \)
\( \text{< TH : TransactionHistory | graph : addInEdges(\text{TID}, predecessors(\text{EID}, \text{LP}, \text{READERS}, \text{WRITERS}), \text{SG}),
\text{readers : op}(\text{TID}, \text{EID}, \text{LP}); \text{READERS} >}) .
\( \text{eq} \ \text{addRead}(\text{TID}, \text{EID}, \text{LP}, \text{< TH : TransactionHistory | graph : addInEdges(\text{TID}, \text{getCreator(\text{EID}, \text{LP}, \text{WRITERS})), addOutEdges(\text{TID}, successors(\text{EID}, \text{LP}, \text{WRITERS}), \text{SG}),
\text{readers : op}(\text{TID}, \text{EID}, \text{LP}); \text{READERS} >}) = \)
\( \text{< TH : TransactionHistory | graph : addInEdges(\text{TID}, \text{getCreator(\text{EID}, \text{LP}, \text{WRITERS})), addOutEdges(\text{TID}, successors(\text{EID}, \text{LP}, \text{WRITERS}), \text{SG}),
\text{readers : op}(\text{TID}, \text{EID}, \text{LP}); \text{READERS} >}) .
\( \text{op} \ \text{addInEdges : TransId TransIdSet SerGraph --> SerGraph} .
\( \text{ceq} \ \text{addInEdges}(\text{TID}, \text{TID'}, \text{TIS1}, \text{SG}) = \text{< TID ; TID'} ; \text{TID'} ; \text{TID'} ; \text{TID'}) \).
\( \text{ceq} \ \text{addInEdges}(\text{TID}, \text{TID'}, \text{TIS1}, \text{SG}) = \text{addInEdges}(\text{TID}, \text{TIS1}, \text{SG}) \text{ if TID'} =/= \text{TID} .
\( \text{eq} \ \text{addInEdges}(\text{TID}, \text{emptyOidSet}, \text{SG}) = \text{SG} .
\( \text{op} \ \text{addOutEdges : TransId TransIdSet SerGraph --> SerGraph} .
\( \text{ceq} \ \text{addOutEdges}(\text{TID}, \text{TID'}, \text{TIS1}, \text{SG}) = \text{< TID ; TID'} ; \text{TID'} ; \text{TID'}) \).
\( \text{ceq} \ \text{addOutEdges}(\text{TID}, \text{TID'}, \text{TIS1}, \text{SG}) = \text{addOutEdges}(\text{TID}, \text{TIS1}, \text{SG}) \text{ if TID'} =/= \text{TID} .
\( \text{eq} \ \text{addOutEdges}(\text{TID}, \text{emptyOidSet}, \text{SG}) = \text{SG} .
\( \text{op} \ \text{hasCycle : SerGraph --> Bool} .
\( \text{eq} \ \text{hasCycle}(\text{SG}) = \text{hasCycle(getTransIds(\text{SG}), \text{SG}, \text{emptyOidSet})} .
\end{verbatim}
vars \(T1\) \(T2\) : Time .
var \(M\) : Msg .

sort DlyMsg .
subsort Msg < DlyMsg < NEConfiguration .
op dly : Msg Time \rightarrow DlyMsg [ctor right id: 0] .

op possibleMsgDelays : SiteId SiteId \rightarrow NatSet [comm] .

eq delta(dly(M, T2), T1) = dly(M, T2 \setminus T1) .
eq mte(dly(M, T1)) = T1 .

crl [sendMsg] :
  (\text{uniCast MC from SID to SID'})
  \Rightarrow
  dly(\text{msg MC from SID to SID'}, N)
if SID =/= SID' \land NS := \text{possibleMsgDelays}(SID, SID') .
eq \text{uniCast MC from SID to SID} = \text{msg MC from SID to SID} .

Listing B.11 mc_fault_injection.rtmaude

(tomod FAULT-INJECTION is
  inc TIMED-BEHAVIOR .
  inc MEGASTORE-SETUP .
  inc MAJORITY-READ .
  inc UPDATES .
  inc UPDATE-FAULT-HANDLERS .
  inc NETWORK-MODEL .
  inc CLIENT-INTERFACE .

vars \(SID\) \(SENDER\) \(THIS\) : SiteId .
var \(EG\) : EntityGroupId .
var \(TID\) : TransId .
var \(LP\) : LogPosition .
var \(LPL\) : LogPositionList .
var \(LE\) : LogEntry .
var \(LEL\) : LogEntryList .
var \(PN\) : Propnum .
var \(MC\) : MsgContent .
var \(T\) : Time .
var \(M\) : Msg .
var \(REST\) : Configuration .
var \(OBJECT\) : Object .
var \(OL\) : OperationList .
var \(CES\) : EntGroupLogPosPair .

msg siteFailure : SiteId \rightarrow Msg .
msg siteRepair : SiteId \rightarrow Msg .
op ttf : \rightarrow Time .
op ttr : \rightarrow Time .
	op failed : Object \rightarrow Object [ctor frozen(1)] .

eq mte(failed(OBJECT)) = INF .
eq delta(failed(OBJECT), T) = failed(OBJECT) .
rl [takeSiteDown] :
    siteFailure(SID)
    < SID: Site | >
    =>
    failed(< SID: Site | >)
    dly(siteRepair(SID), ttr).

rl [bringSiteUp] :
    (siteRepair(SID))
    failed(< SID: Site | >)
    =>
    < SID: Site | >.

crl [msgWhenSiteFailure] :
    (msg MC from SENDER to SID)
    failed(< SID: Site | >)
    =>
    failed(< SID: Site | >)
    if not isInvalidateCoordinator(MC).

op isInvalidateCoordinator : MsgContent --> Bool .
eq isInvalidateCoordinator(invalidateCoordinator(EG, LP)) = true .
eq isInvalidateCoordinator(MC) = false [owise] .

rl [newTrans] :
    (newTrans(SID, TID, OL))
    failed(< SID: Site | >)
    =>
    failed(< SID: Site | >).

rl [invalidateCoordinator] :
    (msg invalidateCoordinator(EG, LP) from SENDER to THIS)
    failed(< THIS: Site | coordinator: CES >)
    =>
    failed(< THIS: Site | coordinator: applyInvalidate(EG, LP, CES) >)
    uniCast invalidateConfirmed(EG, LP) from THIS to SENDER).

dendom)
Appendix C
Real-Time Maude Model of Megastore-CGC

Listing C.1  timed_behavior.rtmaude

(tomod TIMED—BEHAVIOR is
pr TIME—DOMAIN .

var C : Configuration .
vars NEC NEC' : NEConfiguration .
var T : Time .

Var [tick] :
\{C\} => \{delta(C, mte(C))\} in time mte(C) if mte(C) > 0 /\ mte(C) /=/ = INF .

op delta : Configuration Time --> Configuration [format (r! o) frozen (1)] .
eq delta[none, T] = none .

op mte : Configuration --> TimeInf [format (r! o) frozen (1)] .
eq mte[none] = INF .

endtom)

Listing C.2  megastore_setup.rtmaude

(tomod MEGASTORE—SETUP is
inc TIMED—BEHAVIOR .
inc RANDOM .

vars SID SID' ORDERSITE : SiteId .
var SIS : SiteIdSet .
var EGRS : EntityGroupReplicaSet .
var LP : LogPosition .
var T : Time .
var TID : TransId .
vars N N' : Nat .
var SIL : SiteIdList .
vars OID O O' VH : Oid .
vars OS OS1 OS2 : OidSet .
var CES : EntGroupLogPosSet .
var EGLP : EntGroupLogPos .
var LP : LogPosition .
var EG : EntityGroupId .
var OL : OperationList .
vars LOCALTRANS EGROUPS : Configuration .
vars PREV—EGLP EGLP : EntGroupLogPos .
vars EGSET ORDCLASS : EntityGroupIdSet.
var EVERSIONS : EntityVersionList.
vars ES ORDERCLASSES : EntitySet.
var VAL : EntityValue.

var OCID : OrderClassId.

( Mapping from entity group to log position ***)
sorts EntGroupLogPos EntGroupLogPosSet DefEntGroupLogPos.
subsort EntGroupLogPos < EntGroupLogPosSet.
op noEntGroupLogPos : EntGroupLogPosSet [ctor].
op eglp : EntityGroupId LogPosition -> EntGroupLogPos [ctor].
op _!_ : EntGroupLogPosSet EntGroupLogPosSet -> EntGroupLogPosSet [ctor assoc comm id: noEntGroupLogPos].

sort EntGroupLogPosList.
subsort EntGroupLogPos < EntGroupLogPosList.
op emptyEntGroupLogPosList : -> EntGroupLogPosList.

--- Valid for coordinator
op invalidCstate : EntityGroupId LogPosition -> EntGroupLogPos [ctor].

op containsEntityGroupId : EntGroupLogPosSet -> Bool.
eq containsEntityGroupId(EG, eglp(EG, LP) ; CES) = true.
eq containsEntityGroupId(EG, CES) = false [owitz].

( Ordering extensions ***)
sorts OrderClassId OrderClass.
subsort EntityId < OrderClassId.
subsort OrderClass < EntityValue.
op _>: : SiteId EntityGroupIdSet -> OrderClass [ctor].
op noOrderClass : -> OrderClassId [ctor].

op OrderSites : -> EntityGroupId.

--- The entity group updates at each site is a list of sets, where
--- each element in the list represents one transaction
sorts EntGroupUpdate DefEntGroupUpdate EntGroupUpdateList EntGroupUpdateSet.
subsort EntGroupUpdate < EntGroupUpdateSet < DefEntGroupUpdate < EntGroupUpdateList.
op ___ : TransId EntityGroupId LogPosition Bool -> EntGroupUpdate [ctor].
op _!_ : EntGroupUpdateSet EntGroupUpdateSet -> EntGroupUpdateSet [ctor comm assoc id: emptyEntGroupUpdateSet].
op _!_ : EntGroupUpdateList EntGroupUpdateList -> EntGroupUpdateList [ctor assoc id: noEntGroupUpdate].
op noEntGroupUpdate : -> EntGroupUpdateList [ctor].
op emptyEntGroupUpdateSet : -> EntGroupUpdateSet [ctor].

op tentativeMarker : -> EntGroupUpdate [ctor].

sorts ReplicaMapEntry ReplicaPredMap DefReplicaPredMap.
subsort ReplicaMapEntry < ReplicaPredMap < DefReplicaPredMap.
op _!_ : Siteld EntGroupUpdateList -> ReplicaMapEntry [ctor].
op _!_ : ReplicaPredMap ReplicaPredMap -> ReplicaPredMap [ctor comm assoc id: noReplicaPredMap].
op noReplicaPredMap : -> DefReplicaPredMap [ctor].

sorts OrderClassUpdates.
op _!_ : OrderClassId EntityGroupIdSet EntGroupUpdateList -> OrderClassUpdates [ctor].
op _!_ : OrderClassUpdates OrderClassUpdates -> OrderClassUpdates [ctor comm assoc id: noOrderClassUpdates].
op noOrderClassUpdates : -> OrderClassUpdates [ctor].
C Real-Time Maude Model of Megastore-CGC

*** Utility functions for reading the OrderSites entity group

op osu : SiteId Nat --> TransId [ctor].
op osr : SiteId Nat --> TransId [ctor].

op validOrderSiteStatus : EntGroupLogPosSet --> Bool.
eq validOrderSiteStatus(egrp(OrderSites, LP); CES) = true.
eq validOrderSiteStatus(CES) = false [owise].

op getOrderClass : EntityGroupIdSet EntitySet --> OrderClassId.
eq getOrderClass(EG; EGSET, (OCID |--> EVERSIONS :: (LP (SID !>( EG ; ORDCLASS)))) ; ES) = OCID.
eq getOrderClass(EGSET, ES) = noOrderClass [owise].

op isOrderingSite : OrderClassId SiteId EntitySet --> Bool.
eq isOrderingSite(OCID, SID, (OCID |--> EVERSIONS :: (LP (SID !> ORDCLASS)))) ; ES) = true.
eq isOrderingSite(OCID, SID, ES) = false [owise].

op getOrderingClass : EntityGroupIdSet Configuration --> OrderClassId.
eq getOrderingClass(EGSET, < OrderSites : EntityGroup |
entitiesState : ((OCID |--> EVERSIONS :: (LP (SID !> ORDCLASS)))) ; ES > EGROUPS) = OCID.
eq getOrderingClass(EGSET, EGROUPS) = noOrderClass [owise].

op getOrderingSite : OrderClassId Configuration --> SiteId.
eq getOrderingSite(OCID, < OrderSites : EntityGroup |
entitiesState : ((OCID |--> EVERSIONS :: (LP (SID !> ORDCLASS)))) ; ES > EGROUPS) = SID.
eq getOrderingSite(OCID, EGROUPS) = noSiteId [owise].

op getOrderClasses : Configuration --> EntitySet.
eq getOrderClasses(< OrderSites : EntityGroup | entitiesState : ORDERCLASSES > EGROUPS) = ORDERCLASSES.

op getOrderingSite : OrderClassId EntitySet --> SiteId.
eq getOrderingSite(OCID, (OCID |--> EVERSIONS :: (LP (SID !> ORDCLASS)))) ; ES) = SID.

op getOrderingEGs : OrderClassId Configuration --> EntityGroupIdSet.
eq getOrderingEGs(OCID, < OrderSites : EntityGroup |
entitiesState : ((OCID |--> EVERSIONS :: (LP (SID !> ORDCLASS)))) ; ES > EGROUPS) = ORDCLASS.
eq getOrderingEGs(OCID, EGROUPS) = noSiteId [owise].

sorts AwaitingOrder AwaitingOrderSet.
subsort AwaitingOrder < AwaitingOrderSet.
op .__. -- : OrderClassId TransId EntityGroupId LogPosition --> AwaitingOrder [ctor].
op .__ -- : AwaitingOrderSet AwaitingOrderSet --> AwaitingOrderSet [ctor assoc id: noAwaitingOrderSet].
op noAwaitingOrderSet : --> AwaitingOrderSet [ctor].

*** ( Sites )***

sort SiteId.
subsort SiteId < Oid.

sort DefSiteId.
subsort SiteId < DefSiteId.
op .-- : DefSiteId --> DefSiteId [ctor].

class Site
  | coordinator : EntGroupLogPosSet,
  | egOrderings : OrderClassUpdates,
  | awaitingOrder : AwaitingOrderSet,
  | entityGroups : Configuration,
  | seqGen : Nat,
localTransactions : Configuration.

*** (Transaction log) ***
subsort LogPosition < LogPositionList.
subsort LogPosition < DefLogPosition.
op nlogPosition: --> DefLogPosition [ctor].
op emptyLPList: --> LogPositionList [ctor].
op lpos: Nat --> LogPosition [ctor].
op _:::LogPositionList LogPositionList --> LogPositionList [ctor assoc id: emptyLPlist].

op min: LogPosition DefLogPosition --> LogPosition.
eq min(lpos(N), lpos(N')) = if (N <= N') then lpos(N) else lpos(N') f1.
eq min(lpos(N), nlogPosition) = lpos(N).

sorts LogEntry LogEntryList.
subsort LogEntry < LogEntryList.
op noEntries: --> LogEntryList [ctor].
--- Usage: Transaction Logposition SiteId Leader -- replica Updates
op _::Trans: LogPosition SiteId OperationList --> LogEntry [ctor].
op _::LogEntryList LogEntryList --> LogEntryList [ctor assoc id: noEntries].

*** (Entities) ***
sorts EntityId EntityIdSet.
subsort EntityId < EntityIdSet.
op emptyEntityIdSet: --> EntityIdSet [ctor].
op entity: EntityGroupId Nat --> EntityId [ctor].
op _::EntityIdSet EntityIdSet --> EntityIdSet [ctor assoc comm id: emptyEntityIdSet].

sorts EntityEntitySet.
subsort Entity < EntitySet.
op emptyEntitySet: --> EntitySet [ctor].
op _::EntityId EntityVersionList --> Entity [ctor].
op _::EntitySet EntitySet --> EntitySet [ctor assoc comm id: emptyEntitySet].

sort EntityValue.
op v: Nat --> EntityValue [ctor].

sorts EntityVersion EntityVersionList.
subsort EntityVersion < EntityVersionList.
op noEntityVersions: --> EntityVersionList [ctor].
op _::LogPosition EntityValue --> EntityVersion [ctor].

*** (Transactions) ***
sorts TransId.
subsort TransId < Oid.
op initTrans: --> TransId [ctor].

sorts Operation OperationList.
subsort Operation < OperationList.
--- Custom operation to see the latest ordering site
op cr: EntityId --> Operation [ctor].
op w: EntityId EntityValue --> Operation [ctor].
op emptyOpList: --> OperationList [ctor].
op _::OperationList OperationList --> OperationList [ctor assoc id: emptyOpList].

op getWriteEGS: OperationList --> EntityGroupIdSet.
eq getWriteEGS(cr(entity(EG,N))::OL) = getWriteEGS(OL).
eq getWriteEGS(w(entity(EG,N),VAL)::DL) = EGS::getWriteEGS(OL).
eq getWriteEGS(emptyOpList) = emptyOidSet.
sort TransStatus.
ops idle in--paxos : --> TransStatus [ctor].
op executing : LogPosition Time --> TransStatus [ctor].
op awaitOrder : Time --> TransStatus [ctor].
op transTimer : Time --> TransStatus [ctor].
op defTimeout : --> Time.

***
Coordination state represents a mapping to allow a running transaction to keep metadata per replica, eg. while conducting a current read
***
-- -- Used to maintain the state of
sorts ReadState ReadStateSet.
subsort ReadState < ReadStateSet.
op emptyReadState : --> ReadStateSet [ctor].
op _:: : ReadStateSet ReadStateSet --> ReadStateSet [ctor assoc comm id: emptyReadState].
sorts PaxosState PaxosStateSet.
subsort PaxosState < PaxosStateSet.
op emptyPaxosState : --> PaxosStateSet [ctor].
op _:: : PaxosStateSet PaxosStateSet --> PaxosStateSet [ctor assoc comm id: emptyPaxosState].
class Transaction |
operations : OperationList,
reads : EntitySet,
writes : OperationList,
status : TransStatus,
readState : ReadStateSet,
paxosState : PaxosStateSet.
op createNewTrans : TransId OperationList --> Object.
eq createNewTrans(TID, OL) =
< TID : Transaction | operations : OL, status : idle,
readState : emptyReadState,
paxosState : emptyPaxosState,

*** (Applying updates ***)
sort PendingWriteState.
op idle : --> PendingWriteState [ctor].
op updating : Time --> PendingWriteState [ctor].
sorts PendingWrite PendingWriteList.
subsort PendingWrite < PendingWriteList.
op pw : LogPosition PendingWriteState OperationList --> PendingWrite [ctor].
op emptyPWLList : --> PendingWriteList [ctor].
op _:: : PendingWriteList PendingWriteList --> PendingWriteList [ctor assoc id: emptyPWLList].

*** (Update coordination ***)
sorts Propnum DefPropnum.
subsort Nat < Propnum.
subsort Propnum < DefPropnum.
op noPropnum : --> DefPropnum.
sorts PaxosProposal PaxosProposalSet.
subsort PaxosProposal < PaxosProposalSet.
op emptyProposalSet : --> PaxosProposalSet.
op proposal : SiteId TransId LogPosition OperationList Bool Propnum --> PaxosProposal [ctor].
op accepted : SiteId LogEntry Bool PropNum --> PaxosProposal [ctor] .

*** ( Replicas with metadata ***)
sorts EntityGroupReplica EntityGroupReplicaSet .
subsort EntityGroupReplica < EntityGroupReplicaSet .
op noEGR : --> EntityGroupReplicaSet [ctor] .

op getSites : EntityGroupReplicaSet --> SiteIdSet .
eq getSites( egr( SID, N, LP); EGRS) = SID ; getSites( EGRS) .
eq getSites( noEGR) = emptyOidSet .

*** ( EntityGroups ***)
sort EntityGroupId .
subsort EntityGroupId < Oid .
op OrderSite : --> EntityGroupReplica [ctor] .
class EntityGroup |
  entitiesState : EntitySet ,
  replicas : EntityGroupReplicaSet ,
  proposals : PaxosProposalSet ,
  pendingWrites : PendingWriteList ,
  transactionLog : LogEntryList .

*** ( Site id--lists ***)
sort SiteIdList .
subsort SiteId < SiteIdList .
op emptySiteIdList : --> SiteIdList [ctor] .
op _-_ : SiteIdList SiteIdList --> SiteIdList [ctor assoc id: emptySiteIdList] .
op length : SiteIdList --> Nat .
eq length( SID :: SIL) = 1 + length( SIL) .
eq length( emptySiteIdList) = 0 .

*** ( Necessary set constructs ***)
sort NatSet .
subsort Nat < NatSet .
sorts EmptyOidSet SiteIdSet TransIdSet EntityGroupIdSet OidSet .
subsort EmptyOidSet < TransIdSet EntityGroupIdSet SiteIdSet < OidSet .
subsort TransId < TransIdSet .
subsort EntityGroupId < EntityGroupIdSet .
subsort SiteId < SiteIdSet .
subsort Oid < OidSet .
op emptyOidSet : --> EmptyOidSet [ctor] .
op _-_ : EmptyOidSet EmptyOidSet --> EmptyOidSet [ctor assoc comm id: emptyOidSet] .
op _-_ : TransIdSet TransIdSet --> TransIdSet [ctor ditto] .
op _-_ : EntityGroupIdSet EntityGroupIdSet --> EntityGroupIdSet [ctor ditto] .
op _-_ : SiteIdSet SiteIdSet --> SiteIdSet [ctor ditto] .
op _-_ : OidSet OidSet --> OidSet [ctor ditto] .
eq 0; 0 = 0 .

op _-_ : OidSet OidSet --> OidSet [assoc] .
eq ( OS1; 0) setminus ( OS2; 0) = OS1 setminus ( OS2; 0) .
eq OS1 setminus OS2 = OS1 [owise] .
\begin{verbatim}
op _in_ : Oid OidSet -> Bool.
  eq 0 in (0 ; OS) = true.
  eq 0 in OS = false [owise].

op intersection : OidSet OidSet -> OidSet.
  eq intersection(0 ; OS1, 0 ; OS2) = 0 ; intersection(OS1, OS2).
  eq intersection(OS1, OS2) = emptyOidSet [owise].

(** Aggregates **)

op size : OidSet -> Nat.
  eq size (OID; OIS) = 1 + size (OIS setminus OID).
  eq size (emptyOidSet) = 0.
endom)

(omod MSG--WRAPPERS is
  inc MEGASTORE--SETUP.

  var MC : MsgContent.
  vars SID SID' : SiteId.
  var SIS : SiteIdSet.
  vars SYSTEM REST : Configuration.

  sort MsgContent.
  op msg_from_to_ : MsgContent SiteId SiteId -> Msg [ctor]. -- msg to be read/rcvd
  op uniCast_from_to_ : MsgContent SiteId SiteId -> Msg [ctor]. -- msg to be transmitted
  op multiCast_from_to_ : MsgContent SiteId SiteIdSet -> MsgConfiguration.

  --- Sometimes we need to store the set of message contents received,
  --- and we therefore define a sort for multisets of message contents:

  sort MsgContentSet.
  subsort MsgContent < MsgContentSet.
  op noMsgContent : -> MsgContentSet [ctor].
  op _ : MsgContentSet MsgContentSet -> MsgContentSet [ctor assoc comm id: noMsgContent].

  eq multiCast MC from SID to (SID'; SIS) =
    (uniCast MC from SID to SID')
    (multiCast MC from SID to SIS).
  eq multiCast MC from SID to emptyOidSet = none.
endom)

(omod CLIENT--INTERFACE is
  inc MEGASTORE--SETUP.

  inc MSG--WRAPPERS.

  msg newTrans : SiteId TransId OperationList -> Mag.
  msg notifyCommit : SiteId TransId EntGroupLogPosSet EntitySet OperationList -> Mag.
  msg notifyReadOnlyCommit : SiteId TransId EntitySet OperationList -> Mag.
  msg notifyReadOnlyAbort : SiteId TransId OperationList -> Mag.
  msg notifyConflictAbort : SiteId TransId EntitySet OperationList -> Mag.
  msg notifyValidationAbort : SiteId TransId EntitySet OperationList -> Mag.
  msg notifyAbort : SiteId TransId EntitySet OperationList -> Mag.
endom)

Listing C.3  current_read.rtmaude

(omod CURRENT--READ is
  inc CLIENT--INTERFACE.
  inc MEGASTORE--SETUP.
  inc MSG--WRAPPERS.

endom)
\end{verbatim}
vars TID TID': TransId.
vars SID SID' THIS SENDER : SiteId.
var TS : TransStatus.
vars SIS SIS' : SiteIdSet.
var EGRS : EntityGroupReplicaSet.
var EID : EntityId.
var EGID : EntityGroupId.
vars CNT N' N SEQ N1 N2 : Nat.
var EGIS : EntityGroupIdSet.
var LOCALTRANS : Configuration.
vars OL OL' : OperationList.
var ES BSTATE READS : EntitySet.
var EV : EntityVersion.
var DLP : DefLogPosition.
vars VAL1 VAL2 : EntityValue.
var EVER : EntityVersion.
vars EG EG' : EntityGroupId.
var CE : EntGroupLogPos.
var CES : EntGroupLogPosSet.
var EGROUPS : Configuration.
vars EVERSIONS EVERSIONS' : EntityVersionList.
var RSTATE : ReadStateSet.
var LEL : LogEntryList.
var T : Time.
var SIL : SiteIdList.
vars OL1 OL2 : OperationList.
var OCID : OrderClassId.
var EGIDS : EntityGroupIdSet.
var ORDERCLASSES : EntitySet.
var CATCHUP—GSS : Bool.

op readpos : EntityGroupId LogPosition -> ReadState [ctor].

ops readDelay : -> Time.
--- Proceed transaction locally
---
***{  
--- * Current read:
--- * * If local coordinator is up-to-date (e.g. an
--- * entry for the given entityid exists in the coordinator state): Read locally.
--- * * If local coordinator is not up-to-date, perform
--- * a majority read to find the maximum logposition. Once a given
--- * logposition has been received from a majority of sites, the
--- * most responsive replica is elected for a "catchup". See
--- * MAJORITY—READ for details In addition to the modelled delay for
--- * local access (representing the actual bigtable—lookup), we
--- * require the pending write queue to be empty We store the most
--- * recent log entry upon start of the read — this LP is maintained
--- * throughout the transaction. Any conflict with concurrent
--- * updates will then be detected upon commit.
***}

*** A: Non—faulty scenario: Perform a local read

crl [CRA1—startCurrentLocalRead] :
< SID : Site |
  coordinator : (eqlp(EG, LP) ; CES),
  seqGen : SEQ,
  entityGroups : < EG : EntityGroup |
    pendingWrites : emptyPWList > EGROUPS,
  localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, status : idle > LOCALTRANS >
Listing C.4 ordering.rtmaude

```maude
(omod ORDERING is
  inc CURRENT—READ .

  vars EG EG1 EG2 : EntityGroupId .
  var EID : EntityId .
  var SID : SiteId .

  endtom)
```

C Real-Time Maude Model of Megastore-CGC 175

```maude
=>
< SID : Site |
  localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, status :
    executing(LP, readDelay) > LOCALTRANS,
  seqGen : (if CATCHUP—as then s SEQ else SEQ fi) >
    (if CATCHUP—as then (newTrans(SID, osr(SID, SEQ), cr(OCEID)) else none fi)
      if not (containsUpdate(entity(EG,N), OL) and inConflictWithRunning(EG, LOCALTRANS)) /
        OCID := getOrderingClass(EG, < EG : EntityGroup | > EGROUPS) /\n        CATCHUP—as := not (validOrderSiteStatus(eglP(EG, LP) ; CES) or containsOSSCatchup(LOCALTRANS)) .

  op containsOSSCatchup : Configuration -> Bool .
  eq containsOSSCatchup(< osr(SID, N') : Transaction | > LOCALTRANS) = true .
  eq containsOSSCatchup(LOCALTRANS) = false [wise] .

  op inConflictWithRunning : EntityGroupId Configuration -> Bool .
  ceq inConflictWithRunning(EG, < TID : Transaction |
    status : TS, reads : READS, operations : GL1 :: w(entity(EG,N), VAL1) :: OL2 > LOCALTRANS) = true
    if not (TS == idle and filterReads(EG, READS) == emptyEntitySet) .
  eq inConflictWithRunning(EG, < TID : Transaction | writes : GL1 :: w(entity(EG,N), VAL1) :: OL2 >
    LOCALTRANS) = true .
  eq inConflictWithRunning(EG, LOCALTRANS) = false [wise] .

  op filterReads : EntityGroupId EntitySet -> EntitySet .
  eq filterReads(EG, (entity(EG,N) |--> EV) : ES) = (entity(EG,N) |--> EV) ; filterReads(EG, ES) .
  eq filterReads(EG, ES) = emptyEntitySet [wise] .

  op containsUpdate : EntityId OperationList -> Bool .
  eq containsUpdate(EID, OL1 :: w(EID, VAL1) :: OL2) = true .
  eq containsUpdate(EID, OL) = false [wise] .

  rl CRA2—endCurrentLocalRead :
    < THIS : Site |
      entityGroups : < EG : EntityGroup | entitiesState : (entity(EG,N) |--> EVERSIONS) ; BSTATE >
        EGROUPS ,
      localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL,
        status : executing(LP, 0), readState : RSTATE, reads : READS > LOCALTRANS >
    =>
    < THIS : Site |
      localTransactions : < TID : Transaction | operations : OL, status : idle,
        readState : readpos(EG, LP) ; RSTATE, reads : (entity(EG,N) |-->
          getVersion(LP, EVERSIONS)) ; READS > LOCALTRANS > .

  ceq getVersion(1pos(N), EVERSIONS :: (1pos(N1) VAL1) :: (1pos(N2) VAL2) :: EVERSIONS') = (1pos(N1) VAL1)
    if (N1 < N /\ N < N2) .
  ceq getVersion(1pos(N), EVERSIONS :: (1pos(N1) VAL1)) = (1pos(N1) VAL1) if (N1 <= N) .

  op hasQuorum : Nat SiteIdSet -> Bool .
  eq hasQuorum(N, SIS) = (N >= (size(SIS) quo 2 + 1)) .
```

Listing C.4 ordering.rtmaude
\begin{verbatim}
vars LP LP': LogPosition.
vars SIS : SiteIdSet.
vars EGS : EntityGroupIdSet.
vars ES : EntitySet.
vars EGROUPS : Configuration.
vars EVERSIONS : EntityVersionList.
vars ORDERCLASSES : EntitySet.
vars EGIDS ORDERCLASS : EntityGroupIdSet.
vars UPS1 UPS2 : EntGroupIdUpdateSet.
vars DEFUP : DefEntGroupIdUpdate.
vars UP1 : EntGroupIdUpdate.
vars PRED1 PRED2 UPDATELST1 UPDATELST2 : EntGroupIdUpdateList.
vars TID TID1 TID2 : TransId.
var N : Nat.
var EVAL : EntityValue.
var OL : OperationList.
var LEL : LogEntryList.
var OCID OCID1 OCID2 : OrderClassId.
var OCUPDATES : OrderClassUpdates.
vars Awaiting : AwaitingOrderSet.
vars TENTATIVE TENT1 TENT2 ; Bool.

--- For a transaction T updating entity groups EG1, EG2.
--- the initiator receives a set of updates preceding T in EG1 and EG2, respectively
op applyOrdering : OrderClassId EntGroupIdUpdateSet EntGroupIdUpdateList OrderClassUpdates ->
OrderClassUpdates.
eq applyOrdering(OCID, (TID EG LP TENT1), PRED1, (OCID - emptyOidSet |> UPDATELST1) ; OCUPDATES) =
(OCID - emptyOidSet |> updateOrder((TID EG LP TENT1),
applyPermanent(UPDATELST1, PRED1),
applyPermanent(PRED1 :: (TID EG LP TENT1), UPDATELST1))) ; OCUPDATES.
--- If the local ordering is invalid, simply apply the order when received
ceq applyOrdering(OCID, ((TID EG LP TENTATIVE) ; DEFUP), PRED1, (OCID - EGIDS |> UPDATELST1) ;
OCUPDATES) =
(OCID - emptyOidSet |> PRED1 :: ((TID EG LP TENTATIVE) ; DEFUP)) ; OCUPDATES if EGIDS /= emptyOidSet.

op applyPermanent : EntGroupIdUpdateList EntGroupIdUpdateList -> EntGroupIdUpdateList.
eq applyPermanent(PRED1 :: ((TID1 EG LP false) ; DEFUP) :: UPDATELST1, PRED2 :: ((TID2 EG LP true) ; DEFUP) :: UPDATELST2) =
applyPermanent(PRED1 :: ((TID1 EG LP false) ; DEFUP) :: UPDATELST1, PRED2 :: ((TID2 EG LP false) ;
DEFUP) :: UPDATELST2).
eq applyPermanent(UPDATELST1, UPDATELST2) = UPDATELST2 [owise].

op updateOrder : EntGroupIdUpdate EntGroupIdUpdateList EntGroupIdUpdateList -> EntGroupIdUpdateList.
eq updateOrder((TID EG LP false), PRED1, PRED2 :: ((TID EG LP true) ; DEFUP) :: UPDATELST1) =
updateOrder((TID EG LP false), PRED1, PRED2 :: UPDATELST1).

eq updateOrder((TID EG LP TENT1), PRED1 :: UPDATELST1, PRED2 :: UPDATELST1 :: ((TID EG LP TENT1) ; DEFUP) :: UPDATELST2) =
PRED2 :: UPDATELST1 :: ((TID EG LP TENT1) ; DEFUP) :: UPDATELST2.

cceq updateOrder((TID EG LP TENT1), noEntGroupIdUpdate, UPDATELST1) = UPDATELST1 :: (TID EG LP TENT1)
if not containsOrdering(TID, EG, LP, UPDATELST1).

cceq updateOrder((TID EG LP TENT1), PRED1 :: UPDATELST1 :: UPDATELST2, PRED2 :: UPDATELST1) =
(UPDATELST1 :: UPDATELST2 :: (TID EG LP TENT1))
if not containsOrdering(TID, EG, LP, UPDATELST1) \ UPDATELST1 =/= noEntGroupIdUpdate.

cceq updateOrder((TID EG LP TENT1), PRED1 :: UPDATELST1 :: UPDATELST2, PRED2 :: UPDATELST1 :: (TID2 EG LP true)) =
(PRED2 :: UPDATELST1 :: UPDATELST2 :: (TID EG LP TENT1))
\end{verbatim}
if not containsOrdering(TID, EG1, LP, UPDATELST1) \ / UPDATELST1 =/= noEntGroupUpdate .

op applyAwaiting : OrderClassId AwaitingOrderSet OrderClassUpdates -> OrderClassUpdates .

eq applyAwaiting(OCID, (OCID TID EG LP) ; Awaiting, OCUPDATES) =
applyAwaiting(OCID, Awaiting, updateOrdering(OCID, (TID EG LP true), OCUPDATES)) .

eq applyAwaiting(OCID1, (OCID2 TID EG LP) ; Awaiting, OCUPDATES) =
applyAwaiting(OCID1, Awaiting, OCUPDATES) if OCID1 =/= OCID2 .

eq applyAwaiting(OCID, noAwaitingOrderSet, OCUPDATES) = OCUPDATES .

op removeOrdered : AwaitingOrderSet OrderClassUpdates -> AwaitingOrderSet .

eq removeOrdered((OCID TID EG LP) ; Awaiting,
(OCID = emptyOidSet |> UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) :: UPDATELST2) ; OCUPDATES) =
removeOrdered(Awaiting, (OCID = emptyOidSet |> UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) :: UPDATELST2) ; OCUPDATES) .

eq removeOrdered(noAwaitingOrderSet, OCUPDATES) = noAwaitingOrderSet .

eq removeOrdered(Awaiting, OCUPDATES) = Awaiting [wise] .

--- Even if a transaction is applied as a "dummy", it might be ordered at the ordering site
--- and later aborted since the ordering message never arrived. In that case only
--- we allow changing the order list to remove this transaction.

op removeIfOrdered : TransId OrderClassUpdates -> OrderClassUpdates .

eq removeIfOrdered(TID, (OCID - EGIDS |> UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) :: UPDATELST2) ; OCUPDATES) =
((OCID - EGIDS |> UPDATELST1 :: UPDATELST2) ; OCUPDATES) .

eq removeIfOrdered(TID, OCUPDATES) = OCUPDATES [wise] .

op containsOrdering : TransId EntityGroupId LogPosition EntGroupUpdateList -> Bool .

eq containsOrdering(TID, EG, LP, UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) :: UPDATELST2) = true .

eq containsOrdering(TID, EG, LP, UPDATELST1) = false [wise] .

op containsTid : TransId EntGroupUpdateList -> Bool .

eq containsTid(TID, UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) :: UPDATELST2) = true .

eq containsTid(TID, UPDATELST1) = false [wise] .

op getFirstUpdate : OrderClassId OrderClassUpdates -> EntGroupUpdateSet .

eq getFirstUpdate(OCID, (OCID - emptyOidSet |> UPDATELST1 :: UPS1) ; OCUPDATES) = UPS1 .

op updateOrdering : OrderClassId EntGroupUpdateSet OrderClassUpdates -> OrderClassUpdates .

eq updateOrdering(OCID, UPS1, OCUPDATES) =
applyOrdering(OCID, UPS1, noEntGroupUpdate, OCUPDATES) .

op getUpdateList : OrderClassId OrderClassUpdates -> EntGroupUpdateList .

eq getUpdateList(OCID, (OCID - emptyOidSet |> UPDATELST1) ; OCUPDATES) = UPDATELST1 .

op removeTidIfPresent : OrderClassId TransId OrderClassUpdates -> OrderClassUpdates .

eq removeTidIfPresent(OCID, TID, (OCID - emptyOidSet |> UPDATELST1 :: UPS1 :: (TID EG LP TENTATIVE) ; DEFUP :: UPDATELST2) ; OCUPDATES) =
((OCID - emptyOidSet |> UPDATELST1 :: UPDATELST2) ; OCUPDATES) if UPS1 =/= tentativeMarker .

eq removeTidIfPresent(OCID, TID, (OCID - emptyOidSet |> UPDATELST1 :: tentativeMarker :: (TID EG LP TENTATIVE) ; DEFUP :: UPDATELST2) ; OCUPDATES) =
((OCID - emptyOidSet |> UPDATELST1 :: UPDATELST2) ; OCUPDATES) .

eq removeTidIfPresent(OCID, TID, (OCID - emptyOidSet |> (TID EG LP TENTATIVE) ; DEFUP :: UPDATELST1) ; OCUPDATES) =
(OCID - emptyOidSet |> UPDATELST1) .

eq removeTidIfPresent(OCID, TID, OCUPDATES) = OCUPDATES [wise] .

endom

{omod VALIDATION INTERFACE is
inc ORDERING .

--- Actual implementation in a separate module

op isValid? : TransId EntitySet EntGroupUpdateList Configuration -> Bool .

}
Listing C.5  updates.rtmaude

(tomod updates is
    inc CLIENT–INTERFACE .
    inc CURRENT–READ .
    inc ORDERING .
    inc VALIDATION–INTERFACE .

    var EID : EntityId .
    var EIDSET : EntityGroupIdSet .
    var N N' SEQ M1 M2 : Nat .
    var NS : NatSet .
    var T EXP : Time .
    var PN PN' PROPNUM : Propnum .
    var PASID : Nat .
    var DPN SEEN–PROPNUM : DefPropnum .
    var EG EG' : EntityGroupId .
    var EGROUPS : Configuration .
    var TID TID1 TID2 TID3 : TransId .
    var SID SID1 MSID1 MSID2 OSITE NEW–OSITE SENDER THIS ORDERSITE : SiteId .
    var SIS SIS' FAILED REPLICAS : SiteIdSet .
    var EGRS : EntityGroupReplicaSet .
    var RSTATE : ReadStateSet .
    var LOCALTRANS LTRANS1 LTRANS2 : Configuration .
    var WRITEOPS OL OL' OL1 OL2 : OperationList .
    var OP : Operation .
    var PROPSET : PaxosProposalSet .
    var VAL VAL' : EntityValue .
    var LP LP' : LogPosition .
    var ES : EntitySet .
    var EVERSIONS : EntityVersionList .
    var FwL : PendingWriteList .
    var RND : Oid .
    var CE PREV–EGLP : EntGroupLogPos .
    var CES : EntGroupLogPosSet .
    var BEG : EntitySet .
    var WRITES : OperationList .
    var ORDERCLASSES : EntitySet .
    var NEW–OCUPDATES CHANGED–OCUPDATES OCUPDATES PREDUPDATES : OrderClassUpdates .
    var NEW–PREDMAP RCVD–PREDMAP PREDMAP : DefReplicaPredMap .
    var PRELIST : EntGroupUpdateList .
    var UPDATE–LIST UPDATELST1 UPDATELST2 : EntGroupUpdateList .
    var UPS : EntGroupUpdateSet .
    var DEFUP : DefEntGroupUpdate .
    var OCID : OrderClassId .
    var AWAIT–ORDERSET : AwaitingOrderSet .
    var EGIDS : EntityGroupIdSet .

    op defPropExp : --- > Time .
    ops updateDelay : --- > Time .

    endmod)
C Real-Time Maude Model of Megastore-CGC

```plaintext
op initiatePaxosState : TransId EntityGroupId LogEntry EntitySet Propnum -> MagContent .
op signalConflict : TransId EntityGroupId LogPosition -> MagContent .
op signalValidationFail : TransId EntityGroupId LogPosition -> MagContent .
op acceptAllReq : TransId EntityGroupId LogEntry Bool Propnum -> MagContent .
op acceptAllRsp : TransId EntityGroupId LogPosition DefReplicaPredMap Propnum -> MagContent .
op applyReq : TransId EntityGroupId LogPosition Bool DefReplicaPredMap Propnum -> MagContent .
op abortTrans : TransId OrderClassId -> MagContent .

*** (Paxos states)***
op acceptLeader : EntityGroupId LogEntry SiteId Time -> PaxosState [ctor] .
--- Propnum == proposal number, SiteIdSet1 == sites responded yes
op acceptAll : EntityGroupId LogEntry Bool Propnum SiteIdSet DefReplicaPredMap Time -> PaxosState [ctor] .
--- SiteIdSet == sites who did not accept
op acceptedPS : EntityGroupId LogEntry Bool DefReplicaPredMap Propnum -> PaxosState [ctor] .

*** (Paxos—states involved in presence of errors, see UPDATE—FAULT—HANDLERS)***
op prepare : EntityGroupId LogEntry Bool Propnum SiteIdSet Time -> PaxosState [ctor] .
op restartPrepare : EntityGroupId LogEntry Bool Time -> PaxosState [ctor] .
op invalidateing : EntityGroupId LogEntry Bool Propnum SiteIdSet DefReplicaPredMap Time -> PaxosState [ctor] .

rl [bufferWriteOperation] :
< SID : Site |
  entityGroups : EGROUPS,
  localTransactions : < TID : Transaction | operations : w(EID, VAL) : OL, writes : WRITEOPS,
  status : idle > LOCALTRANS
>
=>
< SID : Site |
  localTransactions : < TID : Transaction | operations : OL, writes : WRITEOPS : w(EID, VAL) >
  LOCALTRANS
>
. >>> Initiate commit. If the initiator is the ordering site, we first order and validate the transaction.
crl [initiateCommit] :
< THIS : Site |
  entityGroups : EGROUPS,
  localTransactions : < TID : Transaction | operations : emptyOpList, writes : WRITEOPS,
  readState : RSTATE, reads : READS, paxosState : PSTATE, status : idle > LOCALTRANS
>
=>
< THIS : Site |
  localTransactions : < TID : Transaction | paxosState : NEW—PAXOS—STATE, status : in—paxos >
  LOCALTRANS
>
(createAcceptLeaderMessages(THIS, ORDERSITE, READS, NEW—PAXOS—STATE))
if WRITEOPS /= emptyOpList /
  ORDERCLASSES := getOrderClasses(EGROUPS) /
  OCID := getOrderIdClass(getWriteEGS(READS), ORDERCLASSES) /
  ORDERSITE := get OrderingSite(OCID, ORDERCLASSES) /
  EIDSET := getWriteEGS(WRITEOPS) /
  NEW—PAXOS—STATE := initiatePaxosState(EIDSET, TID, WRITEOPS, THIS, RSTATE, EGROUPS) .

op initiatePaxosState : EntityGroupIdSet TransId OperationList SiteId ReadStateSet Configuration -> PaxosStateSet .
```
eq initiatePaxosState(EG ; EIDSET, TID, WRITEOPS, SID, readpos(EG, lpos(N)) ; RSTATE.
< EG : EntityGroup | replicas : EGRS, transactionLog : LEL :: (TID’ lpos(N) MSID1 OL1) :: LEL’ >
EGROUPS =
acceptLeader(EG, (TID lpos(N) SID filterEGWrites(EG, WRITEOPS)), MSID1, defTimeout)
; initiatePaxosState(EIDSET, TID, WRITEOPS, SID, RSTATE, < EG : EntityGroup | > EGROUPS).
eq initiatePaxosState(emptyOidSet, TID, WRITEOPS, SID, RSTATE, EGROUPS) = emptyPaxosState.

eq createAcceptLeaderMessages(SiteId SiteId EntitySet PaxosStateSet —> Configuration.

eq createAcceptLeaderMessages(SID, ORDERSITE, READS, acceptLeader(EG, (TID LP MSID2 OL), MSID1, EXP) ;
PSTATE) =
(unicast acceptLeaderReq(TID, EG, (TID LP MSID2 OL), (not withinSingleEntityGroup(READS))) from
SID to MSID1)
createAcceptLeaderMessages(SID, ORDERSITE, READS, PSTATE) if (MSID1 != ORDERSITE).

eq createAcceptLeaderMessages(SID, ORDERSITE, READS, acceptLeader(EG, (TID LP MSID2 OL), MSID1, EXP) ;
PSTATE) =
(unicast acceptLeaderReq(TID, EG, (TID LP MSID2 OL), READS) from SID to MSID1)
createAcceptLeaderMessages(SID, ORDERSITE, READS, PSTATE) if (MSID1 == ORDERSITE).

eq createAcceptLeaderMessages(SID, ORDERSITE, READS, PSTATE) = none [otherwise].

op filterEGWrites : EntityGroupId OperationList —> OperationList.
eq filterEGWrites(EG, emptyOpList) = emptyOpList.
eq filterEGWrites(EG, w(entity(EG, N), VAL :: OL) = w(entity(EG, N), VAL) :: filterEGWrites(EG, OL).
eq filterEGWrites(EG, w(entity(EG, N), VAL :: OL) = filterEGWrites(EG, OL) [otherwise].

--- For read-only transactions accessing one entity group, we remove the transaction and commit
immediately

crl [initiateCommitReadOnly] :
< SID : Site |
  entityGroups : EGROUPS,
  egOrderings : OCUPDATES,
  localTransactions :
  < TID : Transaction | operations : emptyOpList, readState : RSTATE, status : idle,
  reads : READS, writes : emptyOpList > LOCALTRANS

> =
< SID : Site |
  localTransactions : LOCALTRANS
  (if TENTATIVE then
   < TID : Transaction | status : awaitOrder(defTimeout) >
  else none f1)
  >
  (if notSingleEntityGroup(READS) then
   (notifyReadOnlyCommit(SID, TID, READS))
  else
   (if not TENTATIVE then
    (if isValidReadOnly?(READS, UPDATE—LIST, EGROUPS) then
     (notifyReadOnlyCommit(SID, TID, READS))
    else
     (notifyReadOnlyAbort(SID, TID))
    fi)
   else
    none
   fi)
  fi)
if ORDERCLASSES := getOrderClasses(EGROUPS) /
OCID := getOrderClass(getWriteEGS(READS), ORDERCLASSES) /
UPDATE—LIST := getUpdateList(OCID, OCUPDATES) /
TENTATIVE := isTentative?(UPDATE—LIST).

op getWriteEGS : EntitySet —> EntityGroupIdSet.
eq getWriteEGS((entity(EG,N) —> EVERSIONS) ; ES) = EG ; getWriteEGS(ES).
eq getWriteEGS(emptyEntitySet) = emptyOidSet.


cceq withinSingleEntityGroup(entity(EG, N1) --> EVERSIONS); (entity(EG', N2) --> EVERSIONS') ; ES) =
false if EG /= EG'.
cceq withinSingleEntityGroup(ES) = true [owise].

*** (Section 4.6.3 Accept Leader. No conflicting proposal ***)

--- Note: In the "Fast write" scenario, we do not run the explicit prepare step. But it appears
--- correct to regard the present proposal as proposal number 0 and store this at the leader
--- (if the original proposer then fails, there is a chance its value will "survive" due to
--- this)

*** (Receive a leader–accept request without validation request ***)
crl [L2successfulLeaderAccept] :
  (msg acceptLeaderReq(TID, EG, (TID LP SID OL), VAL−REQ) from SENDER to THIS)
  < THIS : Site |
    entityGroups :< EG : EntityGroup | proposals : PROPSET, transactionLog : LEL, replicas : EGRS >
    EGROUPS
  >
  =>
  < THIS : Site |
    entityGroups :< EG : EntityGroup |
      proposals : accepted(SENDER, (TID LP SID OL), VAL−REQ, 0) ; PROPSET
    >
    EGROUPS
  >
  (uniCast acceptLeaderRsp(TID, EG, LP, noReplicaPredMap) from THIS to SENDER)
  if not (containsLPos(LP, LEL) or conflictingProposal(TID, LP, 0, PROPSET)) /\ 
  ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /\ 
  OCID := getOrderClass(EG, ORDERCLASSES).

*** (Receive a leader–accept request with read−set (representing a request for validation) ***)
crl [L2successfulLeaderAcceptWithValidation] :
  (msg acceptLeaderReq(TID, EG, (TID LP SID OL), READS) from SENDER to THIS)
  < THIS : Site |
    coordinator : CES,
    entityGroups :< EG : EntityGroup | proposals : PROPSET, transactionLog : LEL, replicas : EGRS >
    EGROUPS,
    egOrderings : OCUPDATES,
    awaitingOrder : AWAIT−ORDERSET
  >
  =>
  < THIS : Site |
    entityGroups :< EG : EntityGroup |
      proposals : (if VALID then accepted(SENDER, (TID LP SID OL), (not IN−SINGLE−EG), 0) ; PROPSET
      else PROPSET fi)
    >
    EGROUPS,
    egOrderings : (if (VALID and ORDERED) then NEW−OCUPDATES else OCUPDATES fi),
    awaitingOrder : (if (VALID and ORDERED) then noAwaitingOrderSet else AWAIT−ORDERSET fi)
  >
  (if (VALID) then
    (if ORDERED then
      (uniCast acceptLeaderRsp(TID, EG, LP,
createPredMap(EGRS, EG, getUpdateList(OCID, PREDUPDATES), < EG : EntityGroup | > EGGROUPS)) from THIS to SENDER)
else
  (uniCast acceptLeaderRsp(TID, EG, LP, noReplicaPredMap) from THIS to SENDER)
fi
else
  (uniCast signalValidationFail(TID, EG, LP) from THIS to SENDER)
fi
if not (containsPos(LP, LEL) or conflictingProposal(TID, LP, 0, PROPSET)) \/
ORDERCLASSES ::= getOrderClasses(< EG : EntityGroup | > EGGROUPS) \/
OCID ::= getOrderClass(EG, ORDERCLASSES) \/
ORDERED ::= (isOrderingSite(OCID, THIS, ORDERCLASSES) and isUpToDate(OCID, OCUPDATES, THIS, < EG : EntityGroup | > EGGROUPS, READS, CES)) \/
PREDUPDATES ::= applyAwaiting(OCID, AWAIT-ORDERSET, OCUPDATES) \/
NEW-OCUPDATES ::= updateOrdering(OCID, (TID EG LP true), PREDUPDATES) \/
IN-SINGLE-EG ::= withinSingleEntityGroup(READS) \/
UPDATE-LIST ::= getUpdateList(OCID, NEW-OCUPDATES) \/
VALID ::= ((not-SINGLE-EG or (not ORDERED)) or isValid(TID, READS, UPDATE-LIST, < EG : EntityGroup | > EGGROUPS)).

--- If all versions read in "EntitySet" are valid at least up to the log position
--- ... and the local coordinator state of the OrderSites--entity group is valid, this site is ready to order
op isUpToDate : OrderClassId OrderClassUpdates SiteId Configuration EntitySet EntGroupLogPosSet -- > Bool.

eq isUpToDate(OCID, OCUPDATES, SID, EGGROUPS, ES, CES) =
  hasValidOrder(OCID, OCUPDATES) and hasSeenAllUpdates(ES, CES) and
  (not acceptedNotOrdered(OCID, getOrderingEGs(OCID, EGGROUPS), getUpdateList(OCID, OCUPDATES), EGGROUPS)) and
  containsEntityGroupId(OrderSites, CES) and (getOrderingUpdate(OCID, SID, EGGROUPS) ==
  noLogPosition).

op acceptedNotOrdered : OrderClassId EntityGroupIdSet EntGroupUpdateList Configuration -- > Bool.
eq acceptedNotOrdered(OCID, EGIDS, UPDATELIST1, EGGROUPS) = true.

eq acceptedNotOrdered(OCID, EG, EGIDS, UPDATELIST1, < EG : EntityGroup | proposals : accepted(SID, (TID LP MSID1 OL1), VAL-REQ, PN); PROPSET > EGGROUPS) =
  true if not containsOrdering(TID, EG, LP, UPDATELIST1).
eq acceptedNotOrdered(OCID, EGIDS, UPDATELIST1, EGGROUPS) = false [owise].

op hasValidOrder : OrderClassId OrderClassUpdates -- > Bool.
eq hasValidOrder(OCID, (OCID EMptyIdSet | >> UPDATE-LIST) ; OCUPDATES) = true.
eq hasValidOrder(OCID, (OCID - EGIDS | >> UPDATE-LIST) ; OCUPDATES) = false [owise].

op getOrderingUpdate : OrderClassId SiteId Configuration -- > DefLogPosition.
ceq get Ordering Update(OCID, OSITE, < OrderSites : EntityGroup | proposals :
  accepted(SID, (TID LP MSID1 OL1 : w(OCID, NEW-SITE ! EGIDS ::
  OL2),
  VAL-REQ, PN); PROPSET > EGGROUPS) = LP if NEW-SITE /= OSITE
  .

eq getOrderingUpdate(OCID, SID, EGGROUPS) = noLogPosition [owise].

op hasSeenAllUpdates : EntitySet EntGroupLogPosSet -- > Bool.
ceq hasSeenAllUpdates(ENTITY(EG,N) | >> (LP VAL)); ES, eglp(EG, LP') ; CES) =
  hasSeenAllUpdates(ES, eglp(EG, LP') ; CES) if min(LP', LP) == LP.
eq hasSeenAllUpdates(emptyEntitySet, eglp(EG, LP) ; CES) = true.
eq hasSeenAllUpdates(ES, CES) = false [owise].

ceq conflictingProposal(TID, LP, PROPNUM, proposal(SID, TID', LP, OL, VAL-REQ, PN); PROPSET) =
  true if (PN >= PROPNUM).
C Real-Time Maude Model of Megastore-CGC

```maude
ceq conflictingProposal(TID, LP, PROPNUM, accepted(SID, LE, VAL—REQ, PN) ; PROPSET) =
  true if (PN >= PROPNUM).
ceq conflictingProposal(TID, LP, PROPNUM, PROPSET) = false [otherwise].

ceq containsLPos(LP, LEL :: (TID LP SID OL) :: LEL') = true.
ceq containsLPos(LP, LEL) = false [otherwise].

--- Task: For each site replicating this entity group, find the most
--- recent member of "OrderClassUpdates"

op createPredMap : EntityGroupReplicaSet EntityGroupReplicaSet EntityGroupUpdateList Configuration ->
  DefReplicaPredMap.
ceq createPredMap(eogr(SID, N, LP) ; EGREF, EG, UPDATE—LIST, EGGROUPS) =
  (SID createMapEntry(UPDATE—LIST, SID, EGGROUPS)) ; createPredMap(EGREF, EG, UPDATE—LIST, EGGROUPS).
ceq createPredMap(noEGR, EG, UPDATE—LIST, EGGROUPS) = noReplicaPredMap.

--- From "EntGroupUpdateList", create a projected ordering list for "SiteId".
--- Configuration is the set of all entity groups, used to determine which entity groups
--- SiteId replicates.

op createMapEntry : EntGroupUpdateList SiteId Configuration -> EntGroupUpdateList.
ceq createMapEntry(UPDATE—LIST :: UPS, SID, EGGROUPS) =
  createMapEntry(UPDATE—LIST, SID, EGGROUPS) ::
  (if (filterReplicatedEntries(UPS, SID, EGGROUPS) =/= emptyEntGroupUpdateSet) then
   filterReplicatedEntries(UPS, SID, EGGROUPS) else noEntGroupUpdate fi).
ceq createMapEntry(noEntGroupUpdate, SID, EGGROUPS) = noEntGroupUpdate.

op filterReplicatedEntries : EntGroupUpdateSet SiteId Configuration -> EntGroupUpdateSet.
ceq filterReplicatedEntries(emptyEntGroupUpdateSet, SID, EGGROUPS) = (emptyEntGroupUpdateSet).
ceq filterReplicatedEntries((TID EG LP TENTATIVE) ; UPS, SID, 
  < EG : EntityGroup | replicas : egr(SID, N, LP) ; EGREF > EGGROUPS) = (TID EG LP TENTATIVE)
  ; filterReplicatedEntries(UPS, SID, < EG : EntityGroup | replicas : egr(SID, N, LP') > EGGROUPS).
ceq filterReplicatedEntries(UPS, SID, EGGROUPS) = (emptyEntGroupUpdateSet) [otherwise].

*** (Section 4.6.3 Accept Leader/Invalidate. Paxos with conflicting proposals ***)
crl [LFircvAcceptLeaderReq] :
  (msg acceptLeaderReq(TID, EG, (TID LP MSID1 OL1), VAL—REQ) from SENDER to THIS)
  < THIS : Site |
  entityGroups : < EG : EntityGroup |
  transactionLog : LEL,
  proposals : PROPSET > EGGROUPS
  =>
  < THIS : Site |
  entityGroups : < EG : EntityGroup | > EGROUPS
  (uniCast signalConflict(TID, EG, LP) from THIS to SENDER)
  if (containsLPos(LP, LEL) or conflictingProposal(TID, LP, 0, PROPSET)).
crl [LFircvAcceptLeaderReqWithValidationRequest] :
  (msg acceptLeaderReq(TID, EG, (TID LP MSID1 OL1), READS) from SENDER to THIS)
  < THIS : Site |
  entityGroups : < EG : EntityGroup |
  transactionLog : LEL,
  proposals : PROPSET > EGROUPS
  =>
  < THIS : Site |
  entityGroups : < EG : EntityGroup | > EGROUPS
```
(uniCast signalConflict(TID, EG, LP) from THIS to SENDER)
if (containsLPos(LP, LEL) or conflictingProposal(TID, LP, 0, PROPSET)).

--- If we receive a conflict signal, we abort the transaction.
< THIS : Site |
  localTransactions : < TID : Transaction | reads : READS, writes : WRITES, paxosState : PSTATE |
  >
  LOCALTRANS

rl [UF2rcvDenyLeaderRsp] :
  (msg signalConflict(TID, EG, LP) from SENDER to THIS)
  < THIS : Site |
  localTransactions : < TID : Transaction | reads : READS, writes : WRITES, paxosState : PSTATE |
  >
  LOCALTRANS

--- We ignore a conflict signal for an already missing transaction.
< THIS : Site |
  localTransactions : LOCALTRANS

rl [UF2.1rcvDenyLeaderRsp] :
  (msg signalConflict(TID, EG, LP) from SENDER to THIS)
  < THIS : Site |
  localTransactions : LOCALTRANS
  \(\text{if not containsTrans(TID, LOCALTRANS)}\).

op containsTrans : TransId Configuration \(\rightarrow\) Bool .
eq containsTrans(TID, < TID : Transaction | > LOCALTRANS) = true .
eq containsTrans(TID, LOCALTRANS) = false [owise] .

--- If accept-leader message arrives delayed, ignore the message
< THIS : Site |
  localTransactions : LOCALTRANS

< THIS : Site |
  localTransactions : LOCALTRANS
  \(\text{if not acceptingLeader(TID, EG, LP, LOCALTRANS)}\).

op acceptingLeader : TransId EntityGroupId LogPosition Configuration \(\rightarrow\) Bool .
eq acceptingLeader(TID, EG, LP, \(\langle\ TID : Transaction |\ \text{paxosState} : \text{acceptLeader}(EG, \langle\ TID LP SID OL\rangle, SENDER, EXP) ; PSTATE \rangle\ |
  LOCALTRANS) = true .
eq acceptingLeader(TID, EG, LP, LOCALTRANS) = false [owise] .

***( Section 4.6.3 – Accept-step ***)

*** Received accept from leader, proceed with requesting accept from all
< THIS : Site |
  coordinator : CES,
  localTransactions : < TID : Transaction | paxosState : acceptLeader(EG, \langle\ TID LP SID OL\rangle, SENDER, EXP) ; PSTATE |
  status : in-paxos, reads : READS, writes : WRITES |
  LOCALTRANS,

entityGroups :
  \(\langle\ EG : EntityGroup |\ \text{proposals} : \text{PROPSET}, \text{replicas} : \text{EGRS}, \text{transactionLog} : \text{LEL} \rangle\ |
  EGROUPS,
egOrderings : OCUPDATES,
waitingOrder : AWAIT-ORDERSET
>
=>
--- Note: We now have accept from leader + this (which might be the same site)
< THIS : Site |
  entityGroups : EGROUPS
   < EG : EntityGroup |
     proposals : (if ((${\text{SEND}}ER =/= THIS) and (not (VALIDATED and (not VALID))) then
       accepted(TID, (TID LP SID OL), (not IN--SINGLE--EG), 0)
      else emptyProposalSet fi) ; PROPSET >,
    egOrderings : (if (not (VALIDATED and (not VALID))) then NEW--OCUPDATES else OCUPDATES fi),
    awaitingOrder : (if (VALID and ORDERED) then noAwaitingOrderSet else AWAIT--ORDERSET fi),
  localTransactions : LOCALTRANS
   (if (VALIDATED and ((not VALID) or (COMPLETE and VALID))) then
      none
    else
      (if ((getSites(EGRS) setminus (SEND\text{ER} ; THIS)) =/= emptyOidSet) then
        < TID : Transaction |
          paxosState : acceptAll(EG, (TID LP SID OL), (not IN--SINGLE--EG), 0, (THIS ; SEND\text{ER}), NEW--PREDMAP, defTimeout) ; PSTATE >
        else
          --- If we get here, TID is VALIDATED and VALID (since all replicas have accepted),
          --- but not COMPLETE
          < TID : Transaction |
          paxosState : acceptedPS(EG, (TID LP SID OL), (not IN--SINGLE--EG), NEW--PREDMAP, 0) ; PSTATE >
        fi)
      fi)
  )
> (if ((getSites(EGRS) setminus (SEND\text{ER} ; THIS)) =/= emptyOidSet) then
  (multiCast acceptAllReq(TID, EG, (TID LP SID OL), (not IN--SINGLE--EG), 0) from THIS to
  getSites(EGRS) setminus (SEND\text{ER} ; THIS ; ORDE\text{RSITE}))
  (if ((ORDERSITE =/= THIS) and (ORDERSITE =/= SEND\text{ER})) then
    (uniCast acceptAllReq(TID, EG, (TID LP SID OL), Reads, 0) from THIS to ORDERSITE)
  else
    none fi)
else
  (if (VALIDATED) then
    (if (COMPLETE and VALID) then
      createApplyMessages(THIS, < EG : EntityGroup | > EGROUPS,
        acceptedPS(EG, (TID LP SID OL), (not IN--SINGLE--EG), NEW--PREDMAP, 0) ; PSTATE)
      notifyCommit(THIS, TID, eglp(EG, LP) ; getEntGroupLogPos(PSTATE), READ, WRITES)
    else
      (if (not VALID) then
        (uniCast abortTrans(TID, OCID) from THIS to SEND\text{ER})
        notifyValidationAbort(THIS, TID, READ, WRITES)
      else
        none fi)
    else
      none
    fi)
  fi)
if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
OCID := getOr\text{derClass}(EG, ORDERCLASSES) /
ORDERSITE := getOr\text{derSite}(OCID, ORDERCLASSES) /
OCUPDATES := applyAwaiting(OCID, AWAIT--ORDERSET, OCUPDATES) /
ORDERED := (ORDERSITE =/= THIS) and isUpToDate(OCID, OCUPDATES, THIS, < EG : EntityGroup | >
EGROUPS, READ, CES) /
NEW--PREDMAP :=
 (if (RCVD--PREDMAP =/= noReplicaPredMap) then
   RCVD--PREDMAP

---
else
  (if (ORDERED) then
    createPredMap(EGRS, EG, getUpdateList(OCID, PREDUPDATES), < EG : EntityGroup | > EGROUPS
      )
    else noReplicaPredMap fi)
  fi) \/
NEW–OCUPDATES :=
  (if (ORDERED) then
    applyOrdering(OCID, (TID LP true), getLocalPred(TI)D, NEW–PREDMAP), PREDUPDATES
  else
    (if (RCVD–PREDMAP /= noReplicaPredMap) then
      applyOrdering(OCID, (TID LP true), getLocalPred(TI)D, NEW–PREDMAP), OCUPDATES
    else OCUPDATES fi)
  fi) \/
UPDATE–LIST := getUpdateList(OCID, NEW–OCUPDATES) /\nCOMPLETE := ((getSites(EGRS) setminus (SENDER ; THIS)) == emptyOidSet) and allEGSAccepted(PSTATE) /\nVALIDATED := (RCVD–PREDMAP /= noReplicaPredMap) or (ORDERSITE == THIS) /\nIN–SINGLE–EG := withinSingleEntityGroup(READS) /\nVALID :=
  ((IN–SINGLE–EG or (not ORDERED)) or
   isValid?(TID, READS, UPDATE–LIST, < EG : EntityGroup | > EGROUPS))
fi).

*** Common case: This is the first time we receive an accept for this log position***
crl [A2rcvAcceptAllReqWithoutOrderRequest] :=
  (msg acceptAllReq(TID, EG, (TID' LP SID OL), VAL–REQ, PROPNUM) from SENDER to THIS)
  < THIS : Site |
    coordinator : CES,
    seqGen : SEQ,
    entityGroups :
      < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL, replicas : EGRS > EGROUPS,
    localTransactions : LOCALTRANS
  >
  =>
  < THIS : Site |
    seqGen : (if CATCHUP–OSS then s SEQ else SEQ fi),
    entityGroups :
      < EG : EntityGroup | proposals : accepted(SENDER, (TID' LP SID OL), VAL–REQ, PROPNUM) ;
        removeProposal(LP, PROPSET) > EGROUPS
    >
  (uniCast acceptAllRsp(TID, EG, LP, noReplicaPredMap, PROPNUM) from THIS to SENDER)
  (if CATCHUP–OSS then (newTrans(THIS, oar(THIS, SEQ), cr(OCID))) else none fi)
if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET)) /\n  ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /\nOCID := getOrderId(EQ, ORDERCLASSES) /\nCATCHUP–OSS := not (validOrderSiteStatus(CES) or containsOSSCatchup(LOCALTRANS)).

*** Accept request at ordering site ***
crl [A2rcvAcceptAllReqWithOrderRequest] :=
  (msg acceptAllReq(TID, EG, (TID LP SENDER OL), READS, PROPNUM) from SENDER to THIS)
  < THIS : Site |
    coordinator : CES,
    entityGroups :
      < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL, replicas : EGRS > EGROUPS,
    egOrderings : OCUPDATES,
    awaitingOrder : AWAIT–ORDERSET
  >
  =>
< THIS : Site |
entityGroups :
  < EG : EntityGroup | proposals :
    (accepted(SENDER, (TID LP SENDER OL), (not IN—SINGLE—EG), PROPNUM) :
      removeProposal(LP, PROPSET)) > EGROUPS,
  egOrderings : (if (VALID and ORDERED) then NEW—OCUPDATES else OCUPDATES fi),
  awaitingOrder : (if (VALID and ORDERED) then noAwaitingOrderSet else AWAIT—ORDERSET fi)
>
(if VALID and ORDERED then
  (uniCast acceptAllRsp(TID, EG, LP,
    createPredMap(EGRS, EG, getUpdateList(OCID, PREDUPDATES), < EG : EntityGroup | >
      EGROUPS), PROPNUM)
    from THIS to SENDER)
else
  --- If validation fails, the response is returned without an ordering map. This signals that
  --- this transaction should be applied only if it does not require validation.
  (uniCast acceptAllRsp(TID, EG, LP, noReplicaPredMap, PROPNUM) from THIS to SENDER)
fi)
if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET)) \/
  OCID := getOrderingClass(EG, < EG : EntityGroup | > EGROUPS) \/
  ORDERSITE := getOrderingSite(OCID, < EG : EntityGroup | > EGROUPS) \/
  ORDERED := ((THIS == ORDERSITE) and isUpToDate(OCID, OCUPDATES, THIS, < EG : EntityGroup | >
    EGROUPS, READS, CES)) \/
  PREDUPDATES := applyAwaiting(OCID, WAIT—ORDERSET, OCUPDATES) \/
  NEW—OCUPDATES := updateOrdering(OCID, (TID EG LP true), PREDUPDATES) \/
  IN—SINGLE—EG := withinSingleEntityGroup(READS) \/
  UPDATE—LIST := getUpdateList(OCID, NEW—OCUPDATES) \/
  VALID := (IN—SINGLE—EG or (ORDERED and isValid?(TID, READS, UPDATE—LIST,
    (< EG : EntityGroup | proposals : PROPSET, transactionLog : LEL > EGROUPS))))).

crl [A2rcvAcceptAllReqWithOrderRequestForOtherTrans] :
  (msg acceptAllReq(TID, EG, (TID' LP SID OL), READS, PROPNUM) from SENDER to THIS)
  < THIS : Site |
    coordinator : CES,
    entityGroups :
      < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL, replicas : EGRS > EGROUPS
  >
  =>
  < THIS : Site |
    entityGroups :
      < EG : EntityGroup | proposals :
        (if IN—SINGLE—EG then (accepted(SENDER, (TID' LP SID OL), true, PROPNUM) :
          removeProposal(LP, PROPSET))
          else PROPSET fi) > EGROUPS
  >
  (if IN—SINGLE—EG then
    (uniCast acceptAllRsp(TID, EG, LP, noReplicaPredMap, PROPNUM) from THIS to SENDER)
  else
    (uniCast signalValidationFail(TID, EG, LP) from THIS to SENDER)
  fi)
if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET)) \/
  TID :=/= TID' \/
  IN—SINGLE—EG := withinSingleEntityGroup(READS).

op hasAcceptedForPosition : LogPosition PaxosProposalSet —> Bool .
eq hasAcceptedForPosition(LP, accepted(SID, (TID LP SID' OL'), VAL—REQ, PW) ; PROPSET) = true .
eq hasAcceptedForPosition(LP, PROPSET) = false [owise] .

eq removeProposal(LP, proposal(SID, TID, LP, OL, VAL—REQ, PW) ; PROPSET) = removeProposal(LP, PROPSET) .
eq removeProposal(LP, PROPSET) = PROPSET [\text{wise}].

\begin{verbatim}
--- Log the accept\-vote. If this was the last, proceed the transaction (validation
--- is implicit, if the transaction comes this far without abort, it has passed the validation step)
crl [4arcvAcceptAllRsp]:
  (msg acceptAllRsp(TID, EG, LP, RCVD\-PREDMAP, PROPNUM) from SENDER to THIS)
< THIS: Site |
  entityGroups :
    < EG : EntityGroup | transactionLog : LEL, replicas : EGRS > EGROUPS,
    localTransactions : < TID : Transaction | paxosState : acceptAll(EG, (TID' LP SID OL),
    VAL\-REQ, PROPNUM, SIS, PREDMAP, EXP) : PSTATE,
      reads : READS, writes : WRITES, status : in\->paxos > LOCALTRANS,
    egOrderings : OCUPDATES
  >
  :=>
< THIS: Site |
  egOrderings : NEW\->OCUPDATES,
  localTransactions :
    LOCALTRANS
    (if ((SENDER : SIS) \=/= getSites(EGRS)) then
      < TID : Transaction |
      paxosState : acceptAll(EG, (TID' LP SID OL), VAL\-REQ, PROPNUM, SIS ; SENDER, NEW\->PREDMAP,
        EXP) : PSTATE >
    else
      (if (allEGSAccepted(PSTATE)) then
        none
      else
        ( < TID : Transaction | paxosState : acceptedPS(EG, (TID' LP SID OL), VAL\-REQ, NEW\->PREDMAP,
          PROPNUM) ; PSTATE > )
      fi)
    fi)
<>
(if COMPLETE then
  createApplyMessages(THIS, < EG : EntityGroup | > EGROUPS, acceptedPS(EG, (TID' LP SID OL), VAL\-REQ,
    NEW\->PREDMAP, PROPNUM) ; PSTATE)
  (if (TID == TID') then
    (if ((NEW\->PREDMAP /= noReplicaPredMap) or (VAL\-REQ == false)) then
      --- If the transaction is ordered, or it does not require
      --- ordering, register it as committed.
      notifyCommit(THIS, TID, eglp(EG, LP); getEntGroupLogPos(PSTATE), READS, WRITES)
    else
      --- If the transaction is not ordered and requires validation,
      --- register an abort. Since VAL\-REQ is true and NEW\->PREDMAP == noReplicaPredMap
      --- the transaction will be aborted by all recipients.
      notifyValidationAbort(THIS, TID, READS, WRITES)
      fi)
    else notifyConflictAbort(THIS, TID, READS, WRITES)
    fi))
else none
fi)
if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
  OCID := getOrderClass(EG, ORDERCLASSES) /
  NEW\->PREDMAP := (if (RCVD\-PREDMAP /= noReplicaPredMap and PREDMAP == noReplicaPredMap) then
    RCVD\-PREDMAP else PREDMAP) /
  COMPLETE := ((SIS ; SENDER) == getSites(EGRS) and allEGSAccepted(PSTATE)) /
  NEW\->OCUPDATES :=
    (if ((RCVD\-PREDMAP /= noReplicaPredMap) and (TID' == TID)) then
      applyOrdering(DCID, (TID EG LP true), getLocalPred(THIS, NEW\->PREDMAP, OCUPDATES)
    else OCUPDATES)
  fi).
op getLocalPred : SiteId ReplicaPredMap \rightarrow EntGroupUpdateList
\end{verbatim}
eq getLocalPred(SID, (SID PREDLIST) ; PREDMAP) = PREDLIST.

op createApplyMessages : SiteId Configuration PaxosState --> Configuration.
eq createApplyMessages(SID, < EG : EntityGroup | replicas : EGRS > EGROUPS, acceptedPS(EG, (TID LP SID OL), VAL-REQ, PREDMAP, PROPNUM) ; PSTATE) = (multiCast applyReq(TID, EG, LP, VAL-REQ, setOrderPermanent(TID, EG, LP, PREDMAP), PROPNUM) from SID to getSites(EGRS)) createApplyMessages(SID, EGROUPS, PSTATE).
eq createApplyMessages(SID, EGROUPS, emptyPaxosState) = none.

op setOrderPermanent : TransId EntityGroupId LogPosition DefReplicaPredMap --> DefReplicaPredMap.
eq setOrderPermanent(TID, EG, LP, (SID (UPDATELST1 :: ((TID EG LP true) ; DEFUP) :: UPDATELST2)) ; PREDMAP) =
setOrderPermanent(TID, EG, LP, (SID (UPDATELST1 :: ((TID EG LP false) ; DEFUP) :: UPDATELST2)) ; PREDMAP).
eq setOrderPermanent(TID, EG, LP, PREDMAP) = PREDMAP [owise].

op getEntGroupLogPos : PaxosState --> EntGroupLogPosSet.
eq getEntGroupLogPos(acceptedPS(EG, (TID LP SID OL), VAL-REQ, PREDMAP, PROPNUM) ; PSTATE) = eglp(EG, LP); getEntGroupLogPos(PSTATE).
eq getEntGroupLogPos(emptyPaxosState) = noEntGroupLogPos.

op createAbortMessages : TransId SiteId OrderClassId Configuration PaxosState --> Configuration.
eq createAbortMessages(TID, SID, OCID, < EG : EntityGroup | replicas : EGRS > EGROUPS, acceptedPS(EG, (TID LP SID OL), VAL-REQ, PREDMAP, PROPNUM) ; PSTATE) = (multiCast abortTrans(TID, OCID) from SID to getSites(EGRS)) createAbortMessages(TID, SID, OCID, EGROUPS, PSTATE).
eq createAbortMessages(TID, SID, OCID, < EG : EntityGroup | replicas : EGRS > EGROUPS, acceptLeader(EG, (TID LP SID OL), MSID, EXP) ; PSTATE) = (multiCast abortTrans(TID, OCID) from SID to getSites(EGRS)) createAbortMessages(TID, SID, OCID, EGROUPS, PSTATE).
eq createAbortMessages(TID, SID, OCID, < EG : EntityGroup | replicas : EGRS > EGROUPS, acceptAll(EG, (TID LP SID OL), VAL-REQ, PN, SIS, PREDMAP, EXP) ; PSTATE) = (multiCast abortTrans(TID, OCID) from SID to getSites(EGRS)) createAbortMessages(TID, SID, OCID, EGROUPS, PSTATE).
eq createAbortMessages(TID, SID, OCID, EGROUPS, emptyPaxosState) = none.
eq createAbortMessages(TID, SID, OCID, none, PSTATE) = none.

*** (Handle validation aborts) ***
crl [receiveAbortSignalAtInitiatorWithinAcceptLeaderStep]:
(msg signalValidationFail(TID, EG, LP) from SENDER to THIS)
  < THIS : Site |
  entityGroups : EGROUPS,
  localTransactions :
    < TID : Transaction | reads : READS, writes : WRITES, paxosState : acceptLeader(EG, LE, SENDER, EXP) ; PSTATE > LOCALTRANS
  >
  =>
  < THIS : Site |
  localTransactions : LOCALTRANS
  =>
notifyValidationAbort(THIS, TID, READS, WRITES)
if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /\
  OCID := getOrderClass(EG, ORDERCLASSES).

crl [processTransactionAbort]:
(msg abortTrans(TID, OCID) from SENDER to THIS)
  < THIS : Site |
  egOrderings : OCUPDATES,
  entityGroups : EGROUPS
  >
==>
< THIS : Site |
  egOrderings : NEW–OCUPDATES,
  entityGroups : removePState(TID, EGROUPS)
>
if NEW–OCUPDATES := removeTidIfPresent(OCID, TID, OCUPDATES).

crl

op removePState : TransId Configuration -> Configuration.
eq removePState(TID, < EG : EntityGroup | proposals : accepted(SENDER, (TID LP MSID1 OL1), VAL–REQ, PN) ; PROPSET > EGROUPS) = removePState(TID, < EG : EntityGroup | proposals : PROPSET > EGROUPS).
eq removePState(TID, EGROUPS) = EGROUPS [wise].

*** (Section 4.6.3 — Apply step ***)

crl | APP3initUpdates :

op allEGSAccepted : PaxosStateSet -> Bool.
eq allEGSAccepted(acceptedPS(EG, LE, VAL–REQ, PREDMAP, N) ; PSTATE) = allEGSAccepted(PSTATE).
eq allEGSAccepted(emptyPaxosState) = true.
eq allEGSAccepted(PSTATE) = false [wise].

--- Apply a valid transaction at site which previously accepted a proposal for TID2

crl | [APP3initUpdates] :

(msg applyReq(TID2, EG, lpos(N2), VAL–REQ1, (THIS PREDLIST) ; PREDMAP, PN) from SENDER to THIS)
< THIS : Site |
  coordinator : eglp(EG, lpos(N1)) ; CES,
  entityGroups :
    < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1),
    pendingWrites : PWL,
    proposals : accepted(SID, (TID2 lpos(N2) MSID2 OL2), VAL–REQ2, PROPNUM) ;
    PROPSET > EGROUPS,
    localTransactions : LOCALTRANS,
    egOrderings : OCUPDATES,
    awaitingOrder : WAIT–ORDERSET
>
==>
< THIS : Site |
  coordinator : eglp(EG, lpos(N2)) ; CES,
  entityGroups :
    < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2 OL2),
    pendingWrites : pw(lpos(N2), idle, OL2) :: PWL,
    proposals : removeProposals(lpos(N2), PROPSET) ; EGROUPS,
    localTransactions : removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS),
    egOrderings : (if (THIS /= SENDER) then applyOrdering(OCID, (TID2 EG lpos(N2) false),
    PREDLIST, OCUPDATES) else OCUPDATES f1),
    awaitingOrder : removeOrdered(WAIT–ORDERSET, NEW–OCUPDATES)
>
(sendDateNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)))
if N2 == s N1 /
ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
OCID := getOrderClass(EG, ORDERCLASSES) /
NEW–OCUPDATES := (if (THIS /= SENDER) then applyOrdering(OCID, (TID2 EG lpos(N2) false),
    PREDLIST, OCUPDATES) else OCUPDATES f1) .

*** Receive an apply request for a transaction not requiring validation, but without a correct order.

crl | [APP3initUpdatesWithoutAndNotRequiringValidation] :

(msg applyReq(TID2, EG, lpos(N2), false, noReplicaPredMap, PN) from SENDER to THIS)
< THIS : Site |
  coordinator : eglp(EG, lpos(N1)) ; CES,
  entityGroups :
    < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1),
pendingWrites : PWL,
proposals : accepted(SID, (TID2 lpos(N2) MSID2 OL2), false, PROPNUM) ;
PROPSET > EGROUPS,
localTransactions : LOCALTRANS,
awaitingOrder : AWAIT-ORDERSET
>
=>
< THIS : Site |
  coordinator : eglp(EG, lpos(N2)) ; CES,
  entityGroups : 
    < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2
      emptyOpList),
    pendingWrites : pw(lpos(N2), idle, OL2) : PWL,
    proposals : removeProposals(lpos(N2), PROPSET) > EGROUPS,
localTransactions : removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS),
awaitingOrder : AWAIT-ORDERSET ; (OCID TID2 EG lpos(N1))
>
(sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)))
if N2 == = s N1 /
ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
OCID := getOrderClass(EG, ORDERCLASSES).

*** Receive an apply request for a transaction requiring a validation, but without a correct order.
*** We apply this as a dummy, effectively aborting it.
crl [APP3initUpdatesWithoutAndRequiringValidation] :=
(msg applyReq(TID2, EG, lpos(N2), true, noReplicaPredMap, PW) from SENDER to THIS)
< THIS : Site |
  coordinator : eglp(EG, lpos(N1)) ; CES,
  entityGroups : 
    < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1),
    proposals : accepted(SID, (TID2 lpos(N2) MSID2 OL2), VAL-REQ2, PROPNUM) ;
    PROPSET > EGROUPS,
localTransactions : LOCALTRANS,
egOrderings : OCUPDATES
>
=>
< THIS : Site |
  coordinator : eglp(EG, lpos(N2)) ; CES,
  entityGroups : 
    < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2
      emptyOpList),
    proposals : removeProposals(lpos(N2), PROPSET) > EGROUPS,
localTransactions : removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS),
egOrderings : removeIfOrdered(TID2, OCUPDATES)
>
(sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)))
if N2 == = s N1 .
removeOthersForLogPosition(EG, LP, LOCALTRANS).
eq removeOthersForLogPosition(EG, LP, < TID : Transaction | paxosState : acceptLeader(EG, (TID3 LP MSID1 OL), MSID2, EXP) ; PSTATE > LOCALTRANS) = removeOthersForLogPosition(EG, LP, LOCALTRANS).
eq removeOthersForLogPosition(EG, LP, LOCALTRANS) = LOCALTRANS [owise].

\[\text{op removeProposals : LogPosition PaxosProposalSet} \rightarrow \text{PaxosProposalSet} .\]
\[\text{eq removeProposals(LP, accepted(SID, (TID LP MSID1 OL), VAL - REQ, PN) ; PROPSET)} = \text{removeProposals(LP, PROPSET)} .\]
\[\text{eq removeProposals(LP, proposal(SID, TID, LP, OL, VAL - REQ, PN) ; PROPSET)} = \text{removeProposals(LP, PROPSET)} .\]
\[\text{eq removeProposals(LP, PROPSET)} = \text{PROPSET [owise]}.\]

\[\text{op sendNotifyAbort : SiteId Configuration Configuration} \rightarrow \text{Configuration} .\]
\[\text{eq sendNotifyAbort(SID, < TID : Transaction | > LTRANS1, < TID : Transaction | > LTRANS2)} = \text{sendNotifyAbort(SID, LTRANS1, LTRANS2)} .\]
\[\text{eq sendNotifyAbort(SID, none, LTRANS1)} = \text{none} .\]
\[\text{eq sendNotifyAbort(SID, < TID : Transaction | reads : READS, writes : WRITES > LTRANS1, LTRANS2)} = \text{(notifyConflictAbort(SID, TID, READS, WRITES)) sendNotifyAbort(SID, LTRANS1, LTRANS2) [owise]}.\]

r1 [APP4beginPendingWrite]:
< THIS : Site |
  entityGroups : < EG : EntityGroup | pendingWrites : PWL :: pw(LP, idle, OL) > EGROUPS
> =>
< THIS : Site |
  entityGroups : < EG : EntityGroup | pendingWrites : PWL :: pw(LP, updating(updateDelay), OL) > EGROUPS
> .

r1 [APP5sendPendingWrite]:
< THIS : Site | entityGroups : < EG : EntityGroup | entitiesState : ES, pendingWrites : PWL :: pw(LP, updating(0), OL :: OP) > EGROUPS >

=>
< THIS : Site | entityGroups :
  < EG : EntityGroup | entitiesState : applyUpdates(OL, LP, ES), pendingWrites : updatePWListUponComplete(LP, OL, PWL) > EGROUPS
> .

\[\text{op updatePWListUponComplete : LogPosition OperationList PendingWriteList} \rightarrow \text{PendingWriteList} .\]
\[\text{eq updatePWListUponComplete(LP, emptyOpList, PWL)} = \text{PWL} .\]
\[\text{eq updatePWListUponComplete(LP, OL, PWL)} = \text{pw(LP, idle, OL) [owise]}.\]

\[\text{op applyUpdates : OperationList LogPosition EntitySet} \rightarrow \text{EntitySet} .\]
\[\text{eq applyUpdates(w[EID, VAL] :: OL, LP, (EID | -> EVERSIONS) ; ES)} = \text{(EID | -> insertEntityValSorted((LP VAL), EVERSIONS)) ES} .\]
\[\text{eq applyUpdates(\emptyset, LP, ES)} = \text{ES}.\]

\[\text{op insertEntityValSorted : EntityVersion EntityVersionList} \rightarrow \text{EntityVersionList} .\]
\[\text{eq insertEntityValSorted((lpos(N) VAL), (lpos(N) VAL'))} = \text{EVERSIONS} = \text{if} (N < N) \text{then} \text{(lpos(N) VAL') :: insertEntityValSorted(lpos(N) VAL), EVERSIONS}) \text{else} \text{(lpos(N) VAL) :: (lpos(N) VAL') :: EVERSIONS) else insertEntityValSorted(lpos(N) VAL), noEntityVersions) = (lpos(N) VAL).} \]
Listing C.6  ordering_fault_tolerance.rtmaude

{(omod ORDERING--FAULT--TOLERANCE is
  inc UPDATES .
  var OCID : OrderClassId .
  vars OCID : OrderClassId .
  var EGROUPS : Configuration .
  var SIS : SiteIdSet .
  var EVERSIONS : EntityVersionList .
  var EGIDS : EntityGroupIdSet .
  var LP : LogPosition .
  var ES : EntitySet .
  var EG : EntityGroupId .
  var EGIDS : EntityGroupReplicaSet .

  vars TID1 TID2 TID3 : TransId .
  var EG : EntityGroupId .
  vars PN PN’ : Propnum .
  var PWL : PendingWriteList .
  vars N1 N2 : Nat .
  var SENDER THIS MSID1 MSID2 : SiteId .
  var LEL : LogEntryList .
  var GL1 GL2 : OperationList .
  var CES : EntGroupLogPosSet .
  var LOCALTRANS : Configuration .
  var AWAIT--ORDERSET : AwaitingOrderSet .
  var PROPSET : PaxosProposalSet .
  var ORDERCLASSES : EntitySet .
  var VAL--REQ : Bool .
  var N : Nat .

  op initOrderSiteUpdate : OrderClassId SiteId SiteId Nat Configuration -- > Msg .
  eq initOrderSiteUpdate(OSI, SID, CURSID, N, EGROUPS) =
    (newTrans(SID, osu(SID, N), chooseNewOrderSite(OSI, SID, CURSID, EGROUPS))) .

  op chooseNewOrderSite : OrderClassId SiteId SiteId Configuration -- > OperationList .
  eq chooseNewOrderSite(OSI, SID, CURSID, < OrderSites : EntityGroup | entitiesState : ES > EGROUPS) =
    cr(OSI) :: createUpdate(OSI, getEntityGroupIdSet(OSI, ES, CURSID, EGROUPS)) .

  op getEntityGroupIdSet : OrderClassId EntitySet -- > EntityGroupIdSet .
  eq getEntityGroupIdSet(OSI, (OCID |-- EVERSIONS : (LP (SID !> EGIDS))) ; ES) = EGIDS .

  op createUpdate : OrderClassId EntityGroupIdSet SiteId Configuration -- > Operation .
  eq createUpdate(OSI, EG : EGIDS, CURSID, < EG : EntityGroup | replicas : EGRS > EGROUPS) =
    w(OSI, chooseSite(orderSiteCandidates(getSites(EGRS), EGIDS, EGROUPS) setminus CURSID) ; (EG ; EGIDS)) .

  op chooseSite : SiteIdSet -- > SiteId .
  eq chooseSite(SID ; SIS) = SID .

  op orderSiteCandidates : SiteIdSet EntityGroupIdSet Configuration -- > SiteIdSet .
  eq orderSiteCandidates(SIS, EG : EGIDS, < EG : EntityGroup | replicas : EGRS > EGROUPS) =
    orderSiteCandidates(intersection(SIS, getSites(EGRS)), EGIDS, EGROUPS) .
  eq orderSiteCandidates(SIS, emptyUIdSet, EGROUPS) = SIS .

  ***( Handle apply for trans without order, and where the transaction does not require validation ***)
  crl [APP3initUpdates] :
    (msg applyReq(TID2, EG, lpos(N2), false, noReplicaPredMap, PN) from SENDER to THIS)
    < THIS : Site
     coordinator : eglp(EG, lpos(N1)) ; CES,
     entityGroups :
     < EG : EntityGroup | transactionLog : LEL :: (TID1 lpos(N1) MSID1 OL1),
     pendingWrites : PWL,}
proposals : accepted(SID, (TID2 lpos(N2) MSID2 OL2), VAL−REQ, PN) ;
PROPSET > EGROUPS,
localTransactions : LOCALTRANS,
awaitingOrder : AWAIT−ORDERSET
>
=>
< THIS : Site |
  coordinator : eglp(EG, lpos(N2)) ; CES,
  entityGroups :
    < EG : EntityGroup | transactionLog : LEL : (TID1 lpos(N1) MSID1 OL1) : (TID2 lpos(N2) MSID2 OL2),
      pendingWrites : pv(lpos(N2), idle, OL2) ; PWL,
      proposals : removeProposals(lpos(N2), PROPSET) > EGROUPS,
    localTransactions : removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS),
    awaitingOrder : AWAIT−ORDERSET ; (OCID TID2 EG lpos(N2))
>
(sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)))
if N2 == s N1 /
ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
OCID := getOrderClass(EG, ORDERCLASSES).

***(
Handle apply for trans without order, and where the transaction does require validation. Here, we simply ignore the updates and treat this update as a "dummy" for the given log position.
***)

\[\text{crl}\ [\text{APP3initUpdates}] :\]
\[\text{(msg applyReq(TID2, EG, lpos(N2)), true, noReplicaPredMap, PN) from SENDER to THIS)}\]
< THIS : Site |
  coordinator : eglp(EG, lpos(N1)) ; CES,
  entityGroups :
    < EG : EntityGroup | 
      transactionLog : LEL : (TID1 lpos(N1) MSID1 OL1),
      proposals : accepted(SID, (TID2 lpos(N2) MSID2 OL2), VAL−REQ, PN) ;
      PROPSET > EGROUPS,
    localTransactions : LOCALTRANS
>
=>
< THIS : Site |
  coordinator : eglp(EG, lpos(N2)) ; CES,
  entityGroups :
    < EG : EntityGroup |
      transactionLog : LEL : (TID1 lpos(N1) MSID1 OL1) : (TID2 lpos(N2) MSID2 emptyOpList),
      proposals : removeProposals(lpos(N2), PROPSET) > EGROUPS,
    localTransactions : removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)
>
(sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, lpos(N2), LOCALTRANS)))
if N2 == s N1 /
ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
OCID := getOrderClass(EG, ORDERCLASSES) .
endom)

Listing C.7  updates_fault_handling.rtaude

\{(tomod UPDATE−FAULT−HANDLERS is
  inc UPDATES .
  inc ORDERING−FAULT−TOLERANCE .

var EID : EntityId .
var N N' N2 : Nat .
var NS : NatSet .\}
C Real-Time Maude Model of Megastore-CGC

vars T EXP : Time.
vars PN PN’ PROPNUM : Propnum.
var PXSID : Nat.
vars DPN SEEN—PROPNUM : DefPropnum.
var EG : EntityGroupId.
var EGROUPS : Configuration.
vars TID TID’ TID1 TID2 TID3 : TransId.
var TIS : TransIdSet.
vars SID SID’ MSID1 MSID2 SENDER THIS ORDERSITE : SiteId.
vars SIS SIS—FAILED REPLICAS : SiteIdSet.
var EGRS : EntityGroupReplicaSet.
var PSTATE : PaxosStateSet.
var LOCALTRANS : Configuration.
vars OL OL’ OL1 OL2 : OperationList.
var OP : Operation.
var PROPSET : PaxosProposalSet.
var LEL LEL’ : LogEntryList.
var LE LE’ NEW—LE : LogEntry.
var LP LP’ : LogPosition.
var PWL : PendingWriteList.
var CES : EntityGroupLogPosSet.
var READS : EntitySet.
var WRITES : OperationList.
vars COMPLETE ORDERED VALID WITHIN—SINGLE—EG AWAITING—ORDER : Bool.
var PREDMAP : DefReplicaPredMap.
var PRED : DefEntGroupUpdate.
var PREDLIST : EntGroupUpdateList.
vars NEW—OCCUPDATES OCCUPDATES PREDUPDATES : OrderClassUpdates.
var ORDERCLASSES : EntitySet.
var OCID : OrderClassId.
var UPDATE—LIST : EntGroupUpdateList.
var AWAIT—ORDERSET : AwaitingOrderSet.
vars VAL—REQ VAL—REQ1 VAL—REQ2 NEW—VAL—REQ ELECT—ORDERSITE : Bool.
var EGIDS : EntityGroupIdSet.

*** (Messages involved in presence of errors)

op prepareAllReq : TransId EntityGroupId LogPosition OperationList Bool Propnum \rightarrow MsgContent.

op prepareAllRsp : TransId EntityGroupId LogEntry Bool Propnum Propnum \rightarrow MsgContent.

op invalidateCoordinator : EntityGroupId LogPosition \rightarrow MsgContent.

op invalidateConfirmed : EntityGroupId LogPosition \rightarrow MsgContent.

*** (Paxos phase 1: Leader election)

--- We did not get any response from the leader. Run phase 1 of Paxos.
--- Megastore—CEG: Check if we should also initiate a new ordering site

crl [PiaceptLeaderFailureRsp]:
< THIS : Site |
localTransactions : < TID : Transaction | paxosState : acceptLeader(EG, (TID LP MSID1 OL1), MSID2, 0) ; PSTATE,
status : in—paxos, reads : READS > LOCALTRANS,
entityGroups : < EG : EntityGroup | proposals : PROPSET,
replicas : egr(THIS, PXSID, LP’) ; EGRS >
EGROUPS
>
=>
< THIS : Site |
localTransactions : < TID : Transaction |
paxosState : prepare(EG, (TID LP MSID1 OL1), VAL—REQ, PN, noPropnum, emptyGidSet, defTimeout) ; PSTATE,
status : in—paxos > LOCALTRANS
\[
\begin{align*}
\text{if } \text{PROPNUM} = \langle \text{crl} \rangle \\
\text{ORDERCLASS} := \text{ordGetClass}(\text{EG}, \text{ORDERCLASS}) \\
\text{AWAITING•ORDER} := (\text{MSID} = \text{ORDERSITE}) \\
\text{VAL•REQ} := (\text{not withinSingleEntityGroup(READS)}) .
\end{align*}
\]

\[
\begin{align*}
\text{op getCurPropnum} &: \text{LogPosition PaxosProposalSet} \rightarrow \text{DefPropnum} \\
\text{eq getCurPropnum}(\text{LP}, \text{proposal}(\text{SID}, \text{TID}, \text{LP}, \text{OL}, \text{VAL•REQ}, \text{PN}) ; \text{PROPSET}) &= \\
\text{maxPN}(\text{PN}, \text{getCurPropnum}(\text{LP}, \text{PROPSET})) .
\end{align*}
\]

\[
\begin{align*}
\text{--- Use the method described in footnote on page 4 og Chandra 2007 ("Paxos made live")} \\
\text{--- to ensure every proposal has a unique PN} \\
\text{op createPropnum} &: \text{DefPropnum Nat Nat} \rightarrow \text{Propnum} .
\end{align*}
\]

\[
\begin{align*}
\text{eq createPropnum}(\text{PN}, \text{N}, \text{PXSID}) &= \\
\text{if } (\text{PN} \mod \text{N}) \geq \text{PXSID} \text{ then} \\
(\text{PN} \mod (\text{PN} \mod \text{N}) + (\text{PXSID} \mod \text{N})) \text{ else} \\
(\text{PN} + \text{sd}((\text{PXSID}, (\text{PN} \mod \text{N})))) .
\end{align*}
\]

\[
\begin{align*}
\text{op maxPN} &: \text{Propnum DefPropnum} \rightarrow \text{Propnum} .
\end{align*}
\]

\[
\begin{align*}
\text{--- Receive a prepare•message with a previous proposal for this log position} \\
\text{crl [P2cvPrepareAllReq]} &: \\
\text{(msg prepareAllReq(TID1, EG, LP, OL1, VAL•REQ1, PROPNUM) from SENDER to THIS) } \\
< \text{THIS : Site} | \\
\text{entityGroups} : < \text{EG : EntityGroup} | \text{transactionLog : LEL, proposals : accepted(SID, (TID2 LP MSID1 OL2), VAL•REQ2, PN)} ; \text{PROPSET} > \text{EGROUPS} \\
\text{>} \\
\text{=} \\
< \text{THIS : Site} | \\
\text{entityGroups} : < \text{EG : EntityGroup} | \text{transactionsLog : LEL, proposals : accepted(SID, (TID2 LP MSID1 OL2), VAL•REQ2, PROPNUM)} ; \text{PROPSET} > \text{EGROUPS} \\
\text{>} \\
(\text{uniCast prepareAllRsp(TID1, EG, (TID2 LP MSID1 OL2), VAL•REQ2, PROPNUM, PN) from THIS to SENDER}) \\
\text{if PROPNUM} > \text{PN} .
\end{align*}
\]

\[
\begin{align*}
\text{crl [P2cvPrepareAllReq]} &: \\
\text{(msg prepareAllReq(TID1, EG, LP, OL, VAL•REQ1, PROPNUM) from SENDER to THIS) } \\
< \text{THIS : Site} | \\
\text{entityGroups} : < \text{EG : EntityGroup} | \text{transactionsLog : LEL, proposals : proposal(SID, TID, LP, OL2, VAL•REQ2, PN)} ; \text{PROPSET} > \text{EGROUPS} \\
\text{>} \\
\text{=} \\
< \text{THIS : Site} | \\
\text{entityGroups} : < \text{EG : EntityGroup} | \text{transactionsLog : LEL, proposals : proposal(SID, TID, LP, OL2, VAL•REQ2, PROPNUM)} ; \text{PROPSET} > \text{EGROUPS} \\
\text{>} \\
(\text{uniCast prepareAllRsp(TID1, EG, (TID LP MSID OL2), VAL•REQ2, PROPNUM, PN) from THIS to SENDER}) \\
\text{if PROPNUM} > \text{PN} .
\end{align*}
\]
--- If we receive a proposal with an obsolete number, then we can safely ignore it

crl \([PF2.lrcvPrepareAllReqWithObsoletePropnum]:\)

\((\text{msg prepareAllReq}(\text{TID}, \text{EG}, \text{LP}, \text{OL}, \text{VAL} = \text{REQ}, \text{PROPNUM}) \text{ from SENDER to THIS})\)

\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | \text{transactionLog} : \text{LEL}, \text{proposals} : \text{PROPSET} > \text{EGROUPS}\)

\(=\)\n
\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | > \text{EGROUPS}\)

\>\n
\(\text{if conflictingProposal}(\text{EG}, \text{LP}, \text{PROPNUM}, \text{PROPSET})\).

crl \([PF2.lrcvPrepareAllReqForApplied]:\)

\((\text{msg prepareAllReq}(\text{TID}, \text{EG}, \text{LP}, \text{OL}, \text{VAL} = \text{REQ}, \text{PROPNUM}) \text{ from SENDER to THIS})\)

\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | \text{transactionLog} : \text{LEL}, \text{proposals} : \text{PROPSET} > \text{EGROUPS}\)

\(=\)\n
\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | > \text{EGROUPS}\)

\>\n
\(\text{(unicast signalConflicts}(\text{TID}, \text{EG}, \text{LP}) \text{ from THIS to SENDER})\)

\(\text{if containsLogPosition}(\text{LP}, \text{LEL})\).

\(\text{op conflictingProposal} : \text{EntityGroupId LogPosition Propnum PaxosProposalSet} \rightarrow \text{Bool} .\)

\(\text{ceq conflictingProposal}(\text{EG}, \text{LP}, \text{PN}, \text{accepted}(\text{SID}, (\text{TID} \text{LP MSID} 1 \text{OL}), \text{VAL} = \text{REQ}, \text{PN}')) \text{ PROPSET} = \)

\(\text{true if PN'} >= \text{PN} .\)

\(\text{ceq conflictingProposal}(\text{EG}, \text{LP}, \text{PN}, \text{proposal}(\text{SID}, \text{TID}, \text{LP}, \text{OL}, \text{VAL} = \text{REQ}, \text{PN}')) \text{ PROPSET} = \)

\(\text{true if PN'} >= \text{PN} .\)

\(\text{eq conflictingProposal}(\text{EG}, \text{LP}, \text{PN}, \text{PROPSET}) = \text{false} \text{ [ovise]} .\)

--- Receive a prepare-message without a previous proposal for this log position

crl \([P3rcvPrepareAllReq]:\)

\((\text{msg prepareAllReq}(\text{TID}, \text{EG}, \text{LP}, \text{OL}, \text{VAL} = \text{REQ}, \text{PROPNUM}) \text{ from SENDER to THIS})\)

\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | \text{transactionLog} : \text{LEL}, \text{proposals} : \text{PROPSET} > \text{EGROUPS}\)

\(=\)\n
\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | > \text{EGROUPS}\)

\>\n
\(\text{(unicast prepareAllRsp}(\text{TID}, \text{EG}, \text{LP}, \text{PROPNUM}) \text{ from THIS to SENDER})\)

\(\text{if not (containsProposal}(\text{EG}, \text{LP}, \text{PROPSET}) \text{ or containsLogPosition}(\text{LP}, \text{LEL}))\).

\(\text{op containsProposal} : \text{EntityGroupId LogPosition PaxosProposalSet} \rightarrow \text{Bool} .\)

\(\text{eq containsProposal}(\text{EG}, \text{LP}, \text{accepted}(\text{SID}, (\text{TID} \text{LP MSID} 1 \text{OL}), \text{VAL} = \text{REQ}, \text{PN}')) \text{ PROPSET} = \)

\(\text{true} .\)

\(\text{eq containsProposal}(\text{EG}, \text{LP}, \text{proposal}(\text{SID}, \text{TID}, \text{LP}, \text{OL}, \text{VAL} = \text{REQ}, \text{PN}')) \text{ PROPSET} = \)

\(\text{true} .\)

\(\text{eq containsProposal}(\text{EG}, \text{LP}, \text{PROPSET}) = \text{false} \text{ [ovise]} .\)

\(\text{op removePreviousProposal} : \text{LogPosition Propnum PaxosProposalSet} \rightarrow \text{PaxosProposalSet} .\)

\(\text{eq removePreviousProposal}(\text{LP}, \text{PN}, \text{proposal}(\text{SID}, \text{TID}, \text{LP}, \text{OL}, \text{VAL} = \text{REQ}, \text{PN}')) \text{ PROPSET} = \text{PROPSET} .\)

\(\text{eq removePreviousProposal}(\text{LP}, \text{PN}, \text{PROPSET}) = \text{PROPSET} \text{ [ovise]} .\)

crl \([P4rcvPrepareAllRspWithValue]:\)

\((\text{msg prepareAllRsp}(\text{TID}, \text{EG}, (\text{TID2 LP MSID} 1 \text{OL}1), \text{VAL} = \text{REQ}1, \text{PROPNUM}, \text{PN}) \text{ from SENDER to THIS})\)

\(< \text{THIS} : \text{Site} \)

\(\text{entityGroups} : < \text{EG}: \text{EntityGroup} | \text{replicas} : \text{EGRS} > \text{EGROUPS},\)
localTransactions : < TID : Transaction | 
  paxosState : prepare(EG, (TID3 LP MSID2 OL2), VAL—REQ2, PROPNUM, SEEN—PROPNUM, SIS, EXP) ; PSTATE, 
  status : in—paxos, reads : READS > LOCALTRANS >
=>
< THIS : Site |
  localTransactions : LOCALTRANS |
  (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
    < TID : Transaction |
      paxosState : acceptAll(EG, NEW—LE, NEW—VAL—REQ, PROPNUM, THIS, noReplicaPredMap, defTimeout) ; PSTATE >
  else
    < TID : Transaction |
      paxosState : prepare(EG, NEW—LE, NEW—VAL—REQ, PROPNUM, maxPn(PN, SEEN—PROPNUM), (SIS ; SENDER), EXP) ; PSTATE >
  fi)
>
(if hasQuorum(size(SIS ; SENDER), REPLICAS) then
  (multiCast acceptAllReq(TID, EG, (TID2 LP MSID1 OL1), VAL—REQ1, PROPNUM) from THIS to (REPLICAS setminus ORDERSITE))
  (uniCast acceptAllReq(TID, EG, (TID2 LP MSID1 OL1), READS, PROPNUM) from THIS to ORDERSITE)
else none fi)

  if REPLICAS := getSites(EGRS) /
    NEW—LE := chooseValue(PN, SEEN—PROPNUM, (TID2 LP MSID1 OL1), (TID3 LP MSID2 OL2)) /
    NEW—VAL—REQ := chooseValReq(PN, SEEN—PROPNUM, VAL—REQ1, VAL—REQ2) /
    ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /
    OCID := getOrderClass(EG, ORDERCLASSES) /
    ORDERSITE := getOrderingSite(OCID, ORDERCLASSES).

  op chooseValue : Propnum DefPropnum LogEntry LogEntry —> LogEntry.
  eq chooseValue(PN, noPropnum, LE, LE') = LE .
  eq chooseValue(PN, PN', LE, LE') = if PN > PN' then LE else LE' fi .

  op chooseValReq : Propnum DefPropnum Bool Bool —> Bool.
  eq chooseValReq(PN, noPropnum, VAL—REQ1, VAL—REQ2) = VAL—REQ1 .
  eq chooseValReq(PN, PN', VAL—REQ1, VAL—REQ2) = if PN > PN' then VAL—REQ1 else VAL—REQ2 fi .

  ***(
    If we have a prepare—quorum, start the accept—phase. If the accept—message contains some other transaction
    than the original, we do not request ordering.
  ***)

  crl [PsrcvPrepareAllRspWithoutValue] :
  (msg prepareAllRsp(TID, EG, LP, PN) from SENDER to THIS)
< THIS : Site |
  entityGroups :
    < EG : EntityGroup | replicas : EGRS >
  EGROUPS,
localTransactions : < TID : Transaction | paxosState : prepare(EG, (TID2 LP MSID1 OL1), VAL—REQ, PN , SEEN—PROPNUM, SIS, EXP) ; PSTATE, 
  status : in—paxos, reads : READS > LOCALTRANS >
=>
< THIS : Site |
  localTransactions : LOCALTRANS |
  (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
    < TID : Transaction |
      paxosState : acceptAll(EG, (TID2 LP MSID1 OL1), VAL—REQ, PN, THIS, noReplicaPredMap, defTimeout) ; PSTATE >
  else
    < TID : Transaction |
C Real-Time Maude Model of Megastore-CGC

\[\text{paxosState} : \text{prepare}(\text{EG}, (\text{TID2 LP MSID1 OL1}), \text{VAL} = \text{REQ}, \text{PN}, \text{SEEN} = \text{PROPNUM}, (\text{SIS} ; \text{SENDER}), \text{EXP}) \]
\[; \text{PSTATE} > \]
\[\text{fi} \]
\[> \]
\[\text{if hasQuorum(size(SIS ; SENDER), REPLICA)} \text{then} \]
\[\text{(if (TID == TID2) then} \]
\[\text{(multiCast acceptAllReq(TID, EG, (TID2 LP MSID1 OL1), VAL = REQ, PN) from THIS to (REPLICAS setminus ORDERSITE)) \]
\[\text{(unicast acceptAllReq(TID, EG, (TID2 LP MSID1 OL1), reads, PN) from THIS to ORDERSITE) \]
\[\text{else} \]
\[\text{(multiCast acceptAllReq(TID, EG, (TID2 LP MSID1 OL1), VAL = REQ, PN) from THIS to REPLICAS) \}
\[\text{fi) \}
\[\text{else none fi)} \]
\[\text{if REPLICA} := \text{getSites(EGRS)} \]
\[\text{ORDERCLASSES} := \text{getOrderClass}(\text{EG, ORDERCLASSES}) \]
\[\text{ORDERSITE} := \text{getOrderingSite(OCID, ORDERCLASSES)} . \]
\[\text{crl [PF1rcvObsoletePrepareRspWithoutValue]} : \]
\[\text{(msg prepareAllRsp(TID, EG, LP, PN) from SENDER to THIS)} \]
\[\text{< THIS : Site} \]
\[\text{localTransactions : LOCALTRANS} \]
\[> \]
\[\Rightarrow \]
\[\text{< THIS : Site | localTransactions : LOCALTRANS >} \]
\[\text{if (not inPrepare(TID, EG, LP, PN, LOCALTRANS)).} \]
\[\text{crl [PF2rcvObsoletePrepareRspWithValue]} : \]
\[\text{(msg prepareAllRsp(TID, EG, (TID' LP MSID1 OL1), VAL = REQ, PROPNUM, PN) from SENDER to THIS)} \]
\[\text{< THIS : Site} \]
\[\text{localTransactions : LOCALTRANS} \]
\[> \]
\[\Rightarrow \]
\[\text{< THIS : Site | localTransactions : LOCALTRANS >} \]
\[\text{if (not inPrepare(TID, EG, LP, PROPNUM, LOCALTRANS)).} \]
\[\text{op inPrepare : TransId EntityGroupId LogPosition Propnum Configuration} \rightarrow \text{Bool} . \]
\[\text{ceq inPrepare(TID, EG, LP, PN, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).} \]
\[\text{eq inPrepare(TID, EG, LP, PN, \}
\[\text{< TID : Transaction | paxosState : prepare(EG, (TID LP MSID1 OL1), VAL = REQ, PN, SEEN = PROPNUM, SIS, EXP) ; PSTATE > LOCALTRANS) = true .} \]
\[\text{eq inPrepare(TID, EG, LP, PN, < TID : Transaction | paxosState : PSTATE > LOCALTRANS) = false [wise].} \]

--- NOTE: This is not included in the Megastore paper, but is our interpretation
\[\text{crl [PF3failedPrepareAllReq]} : \]
\[\text{< THIS : Site} \]
\[\text{entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,} \]
\[\text{localTransactions : < TID : Transaction |} \]
\[\text{paxosState : prepare(EG, (TID2 LP MSID1 OL1), VAL = REQ, PROPNUM, SEEN = PROPNUM, SIS, 0) ; PSTATE,} \]
\[\text{status : in - paxos > LOCALTRANS} \]
\[> \]
\[\Rightarrow \]
\[\text{< THIS : Site} \]
\[\text{localTransactions : < TID : Transaction | paxosState : restartPrepare(EG, (TID2 LP MSID1 OL1), VAL = REQ, N) ; PSTATE,} \]
\[\text{status : in - paxos > LOCALTRANS,} \]
\[\text{entityGroups : < EG : EntityGroup | EGROUPS} \]
\[> \]
\[\text{if (not hasQuorum(size(SIS), getSites(EGRS)))) / \\ N ; NS := possibleBackoffs .} \]
op possibleBackoffs : NatSet.

*** (Paxos phase 2: Accept **)

--- If we receive another accept request for this log position, accept it if and only if it is the same (re-sent) proposal, or
--- the new proposal number is higher than the previous

\[
\text{crl[A3rcvAcceptAllReqSubseqNonOrderingSite]} : \text{(msg acceptAllReq(TID, EG, (TID1 LP MSID1 OL1), VAL—REQ, PW) from SENDER to THIS)}
\]

\[
\text{< THIS : Site | entityGroups :}
\]

\[
\text{< EG : EntityGroup | proposals : accepted(SID, (TID2 LP MSID2 OL2), VAL—REQ, PW’) ; PROPSET,}
\]

\[
\text{replicas : EGRS > EGROUPS,}
\]

\[
\text{egOrderings : OCUPDATES}
\]

\[
\text{>}
\]

\[
\text{< THIS : Site | entityGroups :}
\]

\[
\text{< EG : EntityGroup | proposals : accepted(SID, (TID1 LP MSID1 OL1), VAL—REQ, PW) ; PROPSET > EGROUPS}
\]

\[
\text{> (uniCast acceptAllRsp(TID, EG, LP, noReplicaPredMap, PW) from THIS to SENDER)}
\]

if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /

\[
\text{OCID := getOrderClass(EG, ORDERCLASSES) /}
\]

\[
\text{(TID1 == TID2 and PW == PW’) or (PW > PW’).}
\]

\[
\text{crl[A3rcvAcceptAllReqSubseqOrderingSite]} : \text{(msg acceptAllReq(TID, EG, (TID1 LP MSID1 OL1), READS, PW) from SENDER to THIS)}
\]

\[
\text{< THIS : Site | coordinator : CES,}
\]

\[
\text{entityGroups :}
\]

\[
\text{< EG : EntityGroup | proposals : accepted(SID, (TID2 LP MSID2 OL2), VAL—REQ, PW’) ; PROPSET,}
\]

\[
\text{replicas : EGRS > EGROUPS,}
\]

\[
\text{egOrderings : OCUPDATES,}
\]

\[
\text{awaitingOrder : AWAIT—ORDERSET}
\]

\[
\text{>}
\]

\[
\text{< THIS : Site | entityGroups :}
\]

\[
\text{< EG : EntityGroup | proposals : accepted(SID, (TID1 LP MSID1 OL1), (not WITHIN—SINGLE—EG), PW) ;}
\]

\[
\text{PROPSET > EGROUPS,}
\]

\[
\text{egOrderings : (if (VALID and ORDERED) then NEW—OCUPDATES else OCUPDATES fi),}
\]

\[
\text{awaitingOrder : (if (VALID and ORDERED) then noAwaitingOrderSet else AWAIT—ORDERSET fi)}
\]

\[
\text{> (if VALID and ORDERED then}
\]

\[
\text{(uniCast acceptAllRsp(TID, EG, LP,}
\]

\[
\text{createPredMap(EGRS, EG, getUpdateList(OCID, PREDUPDATES), < EG : EntityGroup | >}
\]

\[
\text{EGROUPS), PW)
\]

\[
\text{from THIS to SENDER)}
\]

\[
\text{else}
\]

\[
\text{(uniCast acceptAllRsp(TID, EG, LP, noReplicaPredMap, PW) from THIS to SENDER)}
\]

\[
\text{fi)
\]

if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) /

\[
\text{OCID := getOrderClass(EG, ORDERCLASSES) /}
\]

\[
\text{(TID1 == TID2 and PW == PW’) or (PW > PW’).}
\]

\[
\text{PREDUPDATES := applyAwaiting(OCID, AWAIT—ORDERSET, OCUPDATES) /}
\]

\[
\text{NEW—OCUPDATES := updateOrdering(OCID, (TID EG true), PREDUPDATES) /}
\]

\[
\text{ORDERED := (TID == TID1 and isOrderingSite(OCID, THIS, ORDERCLASSES) and}
\]

\[
\text{isUpToDate(OCID, OCUPDATES, THIS, < EG : EntityGroup | > EGROUPS, READS, CES)) /}
\]

\[
\text{WITHIN—SINGLE—EG := withinSingleEntityGroup(READS) /}
\]

\[
\text{UPDATE—LIST := getUpdateList(OCID, NEW—OCUPDATES) /}
\]
VALID := ((WITHIN−SINGLE−EG or (not ORDERED)) or isValid?(TID1, READS, UPDATE−LIST, (< EG : EntityGroup | > EGROUPS))).

--- If we receive another accept request for this log position with a lower proposal number than the previous, we discard the message
cr1 [AF2rcvAcceptAllReqObsolete] :
(msg acceptAllReq(TID, EG, (TID1 LP MSID1 OL1), VAL−REQ1, PN) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | proposals : accepted(SID, (TID2 LP MSID2 OL2), VAL−REQ2, PN') ;
PROPSET > EGROUPS
>
< THIS : Site |
entityGroups : < EG : EntityGroup | > EGROUPS
>
if (PN < PN') or (TID1 /= TID2 and PN == PN').

cr1 [AF2rcvAcceptAllReqObsoleteWithValidation] :
(msg acceptAllReq(TID, EG, (TID1 LP MSID1 OL1), READS, PN) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | proposals : accepted(SID, (TID2 LP MSID2 OL2), VAL−REQ, PN') ;
PROPSET > EGROUPS
>
< THIS : Site |
entityGroups : < EG : EntityGroup | > EGROUPS
>
if (PN < PN') or (TID1 /= TID2 and PN == PN').

--- If we receive an accept request for an already logged transaction, discard the message
cr1 [AF2.2rcvAcceptAllRspObsolete] :
(msg acceptAllRsp(TID, EG, (TID1 LP MSID1 OL1), VAL−RSP, PN, PROPNUM) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | transactionLog : LEL :: (TID2 LP MSID2 OL2) :: LEL' > EGROUPS
>
< THIS : Site |
entityGroups : < EG : EntityGroup | > EGROUPS
>
(if (TID2 /= TID1) then (uniCast signalConflict(TID1, EG, LP) from THIS to SENDER) else none fi).

cr1 [AF2.2rcvAcceptAllRspObsoleteWidthValidation] :
(msg acceptAllRsp(TID, EG, (TID1 LP MSID1 OL1), VAL−RSP, PN) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | transactionLog : LEL :: (TID2 LP MSID2 OL2) :: LEL' > EGROUPS
>
< THIS : Site |
entityGroups : < EG : EntityGroup | > EGROUPS
>
(if (TID2 /= TID1) then (uniCast signalConflict(TID1, EG, LP) from THIS to SENDER) else none fi).

--- Ignore an unexpected accept response
cr1 [AF3rcvAcceptAllRspObsolete] :
(msg acceptAllRsp(TID, EG, LP, PREDMAP, PROPNUM) from SENDER to THIS)
< THIS : Site |
localTransactions : LOCALTRANS
hasAllEntityGroups

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).}
\]

--- Only some replicas responded, but sufficient for a quorum. Send invalidate—message to others

crl [A6initInvalidation] :

\[
\text{op inAcceptAll : TransId EntityGroupId LogPosition Propnum Configuration --> Bool.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : acceptAll(EG, (TID’ LP SID OL), VAL—REQ, PROPNUM, SIS,}
\]

\[
\text{PREDMAP, EXP) ; PSTATE > LOCALTRANS) = true.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : PSTATE > LOCALTRANS) = false [owise].}
\]

--- Only some replicas responded, but sufficient for a quorum. Send invalidate—message to others

crl [A6initInvalidation] :

\[
\text{op inAcceptAll : TransId EntityGroupId LogPosition Propnum Configuration --> Bool.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : acceptAll(EG, (TID’ LP SID OL), VAL—REQ, PROPNUM, SIS,}
\]

\[
\text{PREDMAP, EXP) ; PSTATE > LOCALTRANS) = true.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : PSTATE > LOCALTRANS) = false [owise].}
\]

--- Only some replicas responded, but sufficient for a quorum. Send invalidate—message to others

crl [A6initInvalidation] :

\[
\text{op inAcceptAll : TransId EntityGroupId LogPosition Propnum Configuration --> Bool.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : acceptAll(EG, (TID’ LP SID OL), VAL—REQ, PROPNUM, SIS,}
\]

\[
\text{PREDMAP, EXP) ; PSTATE > LOCALTRANS) = true.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : PSTATE > LOCALTRANS) = false [owise].}
\]

--- Only some replicas responded, but sufficient for a quorum. Send invalidate—message to others

crl [A6initInvalidation] :

\[
\text{op inAcceptAll : TransId EntityGroupId LogPosition Propnum Configuration --> Bool.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM, LOCALTRANS) = false if not containsTrans(TID, LOCALTRANS).}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : acceptAll(EG, (TID’ LP SID OL), VAL—REQ, PROPNUM, SIS,}
\]

\[
\text{PREDMAP, EXP) ; PSTATE > LOCALTRANS) = true.}
\]

\[
\text{eq inAcceptAll(TID, EG, LP, PROPNUM,}
\]

\[
\text{< TID : Transaction | paxosState : PSTATE > LOCALTRANS) = false [owise].}
\]
cdef applyInvalidate(EG, lpos(N), eglp(E, lpos(N')) ; CES) = invalidCstate(EG, lpos(N)) ; CES if (N' <= N)
.
cdef applyInvalidate(EG, lpos(N), invalidateCstate(EG, lpos(N')) ; CES) = invalidCstate(EG, lpos(N)) ; CES
if (N' < N).
cdef applyInvalidate(EG, lpos(N), eglp(E, lpos(N')) ; CES) = eglp(E, lpos(N')) ; CES if (N' > N). 
eq applyInvalidate(EG, lpos(N), CES) = CES [owise] .

crl [A8rcvInvalidateConfirmed] :
(msg invalidateConfirmed(EG, LP) from SENDER to THIS)
< THIS : Site |
   entityGroups : EGROUPS,
   localTransactions : < TID : Transaction |
      paxosState : invalidating(EG, (TID' LP SID OL), VAL−REQ, PROPNUM, SIS−FAILED, PREDMAP, EXP) ;
      PSTATE,
      reads : READS, writes : WRITES > LOCALTRANS
> => 
< THIS : Site | localTransactions : LOCALTRANS (if COMPLETE then
   (if allE8SAccepted(PSTATE) then none
   else
      < TID : Transaction | paxosState : acceptedPS(EG, (TID' LP SID OL), VAL−REQ, PREDMAP, PROPNUM) ;
      PSTATE >
   fi)
else
   < TID : Transaction |
      paxosState : invalidating(EG, (TID' LP SID OL), VAL−REQ, PROPNUM, SIS−FAILED setminus SENDER,
      PREDMAP, EXP) ; PSTATE >
   fi)
> (if (COMPLETE and allE8SAccepted(PSTATE)) then
   createApplyMessages(SID, EGROUPS, acceptedPS(EG, (TID' LP SID OL), VAL−REQ, PREDMAP, PROPNUM) ;
   PSTATE)
   (if (TID == TID') then
      (if ((PREDMAP /= noReplicaPredMap) or (VAL−REQ == false)) then
         notifyCommit(THIS, TID, eglp(E, LP) ; getEntGroupLogPos(PSTATE), READS, WRITES)
      else
         notifyValidationAbort(THIS, TID, READS, WRITES)
      fi)
   else
      notifyConflictAbort(THIS, TID, READS, WRITES)
   fi)
else
   none
fi)
if COMPLETE := ((SIS−FAILED setminus SENDER) == emptyOidSet).

crl [A8rcvInvalidateConfirmedObsolete] :
(msg invalidateConfirmed(EG, LP) from SENDER to THIS)
< THIS : Site |
localTransactions : LOCALTRANS
>
=>
< THIS : Site |
localTransactions : LOCALTRANS
> if not inInvalidate(EG, LP, LOCALTRANS) .

op inInvalidate : TransId LogPosition Configuration -> Bool.

eq inInvalidate(EG, LP, < TID : Transaction | paxosState : invalidating(EG, (TID' LP MSID OL), VAL−REQ, PROPNUM, SIS,
   PREDMAP, EXP) ; PSTATE > LOCALTRANS) = true .

eq inInvalidate(EG, LP, LOCALTRANS) = false [owise].
r1 \{ AF4\{resend\}Invalid\} :< THIS : Site |
localTransactions : < TID : Transaction | paxosState : invalidating(EG, (TID' LP SID OL), VAL-REQ, PROPNUM, SIS, PREDMAP, 0) ; PSTATE, 
status : in-paxos > LOCALTRANS, 
entityGroups : < EG : EntityGroup | EG >
=><< THIS : Site |
localTransactions : < TID : Transaction | paxosState : invalidating(EG, (TID' LP SID OL), VAL-REQ, PROPNUM, SIS, PREDMAP, defTimeout) ; PSTATE > LOCALTRANS, 
entityGroups : < EG : EntityGroup | EG >
> (multiCast invalidateCoordinator(EG, LP) from THIS to SIS).

--- Timeout without quorum - failure handling according to #3 in 4.6.3
crl [ restart\{Prepare\} ] :< THIS : Site |
localTransactions : < TID : Transaction | paxosState : acceptAll(EG, (TID' LP MSID1 OL), VAL-REQ, PROPNUM, SIS, PREDMAP, 0) ; PSTATE, 
status : in-paxos > LOCALTRANS, 
entityGroups : < EG : EntityGroup | replicas : EG >
=><< THIS : Site |
localTransactions : < TID : Transaction | paxosState : restartPrepare(EG, (TID' LP MSID1 OL), VAL-REQ, N) ; PSTATE > LOCALTRANS, 
entityGroups : < EG : EntityGroup | EG >
> if not hasQuorum(size(SIS), getSites(EGRS)) \ N ; NS := possibleBackoffs .
crl [ AF4\{init\}\{Validate\} ] :< THIS : Site |
localTransactions : < TID : Transaction | paxosState : restartPrepare(EG, (TID' LP MSID1 OL1), VAL-REQ, 0) ; PSTATE, 
status : in-paxos > LOCALTRANS, 
entityGroups : < EG : EntityGroup | proposals : PROPSET, 
replicas : egr(THIS, PAXID, LP) ; EGRS >
=><< THIS : Site |
localTransactions : < TID : Transaction | paxosState : prepare(EG, (TID' LP MSID1 OL1), VAL-REQ, PW, noPropnum, emptyDidSet, defTimeout) ; PSTATE, 
status : in-paxos > LOCALTRANS, 
entityGroups : < EG : EntityGroup | EG >
> (multiCast prepareAllReq(TID, EG, LP, OL1, VAL-REQ, PW) from THIS to REPLICAS)
if REPLICAS := getSites(egr(THIS, PAXID, LP) ; EGRS) /=
PW := createPropnum(getCurPropnum(LP, PROPSET), size(REPLICAS), PAXID).

*** ( After Paxos-\{consensus\}: Apply ***)

--- In case of some previous error, we allow processing "out of order"
crl [ APP3.l\{init\}\{Updates\}Invalid\{ated\} ] :
(msg apply\{Req\}(TID, EG, LP, VAL-REQ, (THIS PRED\{LIST\}) ; PREDMAP, PROPNUM) from SENDER to THIS)
< THIS : Site |
coordinator : CES, 
entityGroups :
< EG : EntityGroup | transactionLog : LEL,
   pendingWrites : PWL,
   proposals : accepted(SID, (TID2 LP MSID1 OL), VAL–REQ2, PW’) ; PROPSET >
   EGROUPS,
localTransactions : LOCALTRANS,
egOrderings : OCUPDATES,
awaitingOrder : AWAITS–ORDERSET
>

=>
< THIS : Site |
   coordinator : invalidateUnlessUpToDate(EG, LP, CES),
entityGroups : 
< EG : EntityGroup | transactionLog : LEL,
   pendingWrites : PWL,
   proposals : accepted(SID, (TID2 LP MSID1 OL), VAL–REQ2, PW’) ; PROPSET >
   EGROUPS,
localTransactions : LOCALTRANS,
egOrderings : OCUPDATES,
awaitingOrder : (if (TID2 == TID) then AWAITS–ORDERSET ; (OCID2 TID2 EG LP)
   else removeOrdered(AWAITS–ORDERSET, NEW–OCUPDATES) fi)
>

(sendNotifyAbort(THIS, LOCALTRANS, removeOthersForLogPosition(EG, LP, LOCALTRANS))
if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGROUPS) \/
OCID := getOrderClass(EG, ORDERCLASSES) /\ 
NEW–OCUPDATES := (if ((THIS =/= SENDER) and (TID2 == TID)) then
   applyOrdering(OCID, (TID2 EG LP false), PREDLIST, OCUPDATES) else OCUPDATES fi) /\ 
((PROPNUM == PW’) or (TID == TID2)) /\ 
not (containsEntityGroupId(EG, CES) and isNext(LP, LEL)) .

--- In case of some previous error, we allow processing "out of order"

crl [APP3.limitUpdatesInvalidatedNotOrdered] :
(msg applyReq(TID, EG, LP, VAL–REQ, noReplicaPredMap, PROPNUM) from SENDER to THIS)
< THIS : Site |
   coordinator : CES,
entityGroups : 
< EG : EntityGroup | transactionLog : LEL,
   pendingWrites : PWL,
   proposals : accepted(SID, (TID2 LP MSID1 OL), VAL–REQ2, PW’) ; PROPSET >
   EGROUPS,
localTransactions : LOCALTRANS,
awaitingOrder : AWAITS–ORDERSET
>

=>
< THIS : Site |
   coordinator : invalidateUnlessUpToDate(EG, LP, CES),
entityGroups : 
< EG : EntityGroup | transactionLog : (if (VAL–REQ2 == false) then
   insertLogEntrySorted((TID2 LP MSID1 OL), LEL) else
   insertLogEntrySorted((TID2 LP MSID1 emptyOpList), LEL) fi),
   pendingWrites : (if (VAL–REQ2 == false) then pw(LP, idle, OL) ; PWL else PWL fi),
   proposals : removeProposals(LP, PROPSET) > EGROUPS,
localTransactions : LOCALTRANS,
egOrderings : removeOthersForLogPosition(EG, LP, LOCALTRANS),
awaitingOrder : (if (VAL–REQ2 == false) then (OCID TID2 EG LP) ; AWAITS–ORDERSET else AWAITS–ORDERSET f)
(sendNotifyAbort(THEachine, LOCALTRANS, removeOthersForLogPosition(EG, LP, LOCALTRANS)))
if ORDERCLASSES := getOrderClasses(< EG : EntityGroup | > EGGROUPS) /
OCID := getOrderClass(EG, ORDERCLASSES) /
((PROPNUM == PN) or (TID == TID2)) /
not (containsEntityGroupId(EG, CES) and isNext(LP, LEL)).

op isNext : LogPosition LogEntryList --> Bool.
eq isNext(lpos(a N), LEL :: (TID lpos(N) MSID1 OL)) = true.
eq isNext(LP, LEL) = false [wise].

op invalidateUnlessUpToDate : EntityGroupId LogPosition EntGroupLogPosSet --> EntGroupLogPosSet.
ceq invalidateUnlessUpToDate(EG, lpos(N1), eglp(EG, lpos(N2)) ; CES) =
applyInvalidate((EG, lpos(N1), eglp(EG, lpos(N2))) ; CES) if N1 > (a N2).
eq invalidateUnlessUpToDate(EG, LP, CES) = CES [wise].

op insertLogEntrySorted : LogEntry LogEntryList --> LogEntryList.
insertLogEntrySorted((TID1 lpos(N1) MSID1 OL1), (TID2 lpos(N2) MSID2 OL2) :: LEL) =
if (N1 < N2) then
(TID1 lpos(N1) MSID1 OL1) :: (TID2 lpos(N2) MSID2 OL2) :: LEL
else
(TID2 lpos(N2) MSID2 OL2) :: insertLogEntrySorted((TID1 lpos(N1) MSID1 OL1), LEL) fi.
eq insertLogEntrySorted(LE, noEntries) = LE.

------- If we receive an apply req for which we do not have an accept, we invalidate the coordinator

  crl [APP3.InitUpdatesWithoutAccept] ::
  (msg applyReq(TID, EG, LP, VAL--REQ, PREDMAP, PROPNUM) from SENDER to THIS)
  < THIS : Site |
  coordinator : CES,
  entityGroups : < EG : EntityGroup | proposals : PROPSET > EGGROUPS

  >
  =>
  < THIS : Site | coordinator : applyInvalidate(EG, LP, CES) >
  if not containsAccept(SENDER, TID, LP, PROPNUM, PROPSET).

op containsAccept : SiteId TransId LogPosition Propnum PaxosProposalSet --> Bool.
eq containsAccept(SID, TID, LP, PN, accepted(SID, (TID' LP MSID1 OL), VAL--REQ, PN') ; PROPSET) = true.
eq containsAccept(SID, TID, LP, PN, accepted(SID, (TID LP MSID1 OL), VAL--REQ, PN') ; PROPSET) = true.
eq containsAccept(SID, TID, LP, PN, PROPSET) = false [wise].

------- With competing leaders, we might receive two apply messages for the same transaction

  crl [APP4.InitUpdatesForAppliedTrans] ::
  (msg applyReq(TID, EG, LP, VAL--REQ, PREDMAP, PROPNUM) from SENDER to THIS)
  < THIS : Site |
  coordinator : CES,
  entityGroups : < EG : EntityGroup | transactionLog : LEL,
  proposals : PROPSET > EGGROUPS

  >
  =>
  < THIS : Site |
  coordinator : CES,
  entityGroups : < EG : EntityGroup | transactionLog : LEL,
  proposals : removeProposals(LP, PROPSET) > EGGROUPS

  > if containsLogPosition(LP, LEL).

eq containsLogPosition(LP, LEL :: (TID LP MSID1 OL) :: LEL') = true.
ccq containsLogPosition(lpos(N), (TID lpos(N') MSID1 OL :: LEL') = true if (N < N')
eq containsLogPosition(LP, LEL) = false [wise].
endcon)
This module implements "catchup", see step 3 and 4 of section 4.6.2

Due to some previous fault, the local coordinator is not up to date. Perform a majority read crl [CRB1-initMajorityRead]:
< SID: Site |
  coordinator: CES,
seqGen : SEQ,
entityGroups : < EG : EntityGroup | replicas : EGRS >> EGROUPS,
egOrderings : OLISTS,
localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL, readState : RSTATE, status : idle > LOCALTRANS >
=>
< SID : Site |
localTransactions : < TID : Transaction | operations : cr(entity(EG,N)) :: OL,
readState : majorityRead(entity(EG,N), noLogPosition, emptySiteIdList) ; RSTATE,
status : transTimer(defTimeout) > LOCALTRANS,
egOrderings : invalidateOrdering(EG, OCID, OLISTS),
seqGen : (if CATCHUP =/= seq then seq else seq fi) >
(multiCast majorityRead(EG, TID) from SIF to getsites(EG) setminus SID)
(if CATCHUP =/= seq then (newTrans(SID, osr(SID, SEQ), cr(OCID))) else none fi)
if not (inConflictWithRunning(EG, LOCALTRANS) or containsEntityGroupId(EG, CES)) /
OCID := getOrderingClass(EG, < EG : EntityGroup | > EGROUPS) /
CATCHUP := (EG =/= OrderSites) and (not (validOrderSiteStatus(CES) or containsOSSCatchup(
LOCALTRANS))) .

op invalidateOrdering : EntityGroupId OrderClassId OrderClassUpdates -> OrderClassUpdates .
eq invalidateOrdering(EG, OCID, (OCID =EGIDS | >> UPDATELST1) ; OLISTS) = (OCID = (EGIDS ; EG) | >> UPDATELST1) ; OLISTS .

op setValidOrdering : EntityGroupId OrderClassId OrderClassUpdates -> OrderClassUpdates .
eq setValidOrdering(EG, OCID, (OCID = EG ; EGIDS >> UPDATELST1) ; OLISTS) = (OCID = EGIDS >> UPDATELST1) ; OLISTS .
eq setValidOrdering(EG, OCID, (OCID = emptyOidSet >> UPDATELST1) ; OLISTS) = (OCID = emptyOidSet | >> UPDATELST1) ; OLISTS .

r1 [CRB2–rcvMajorityReadRequest] :
(msg majorityRead(EG,TID) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | transactionLog : LEL :: (TID’ lpos(N) SID OL) > EGROUPS >
=>
< THIS : Site >
(unicast majorityReadResponse(EG, TID, lpos(N)) from THIS to SENDER) .

cr1 [CRB3–rcvMajorityReadResponse] :
(msg majorityReadResponse(EG, TID, LP) from SENDER to THIS)
< THIS : Site |
entityGroups : < EG : EntityGroup | transactionLog : LEL, replicas : EGRS > EGROUPS,
localTransactions : < TID : Transaction |
operations : cr(entity(EG,N)) :: OL, status : transTimer(T).
readState : majorityRead(entity(EG,N), DLP, SIL) ; RSTATE > LOCALTRANS >
=>
< THIS : Site | localTransactions : LOCALTRANS
(if not HAS–QUORUM) then
< TID : Transaction |
readState : majorityRead(entity(EG,N), LATEST, SIL’ :: SID) ; RSTATE >
else
< TID : Transaction |
status : transTimer(defTimeout).
readState : catchingUp(entity(EG,N), SIL’, LPL–MISSING, false) ; RSTATE >
fi
> (if HAS–QUORUM then
(unicast catchupRequest(EG, TID, LPL–MISSING) from THIS to SIF)
else none fi)
if HAS–QUORUM := hasQuorum(length(SENDER :: SIL), getSites(EG)) /
lpos(N1) := getMostRecentLPos(LEL) /
}
C Real-Time Maude Model of Megastore-CGC

\[
\text{majorityRead}(\text{entity}(\text{EG}, N), \text{LATEST}, \text{SIL}' \,: \text{SID}) := \text{updateMreadState}(\text{SENDER}, \text{entity}(\text{EG}, N), \text{LP}, \\
\text{majorityRead}(\text{entity}(\text{EG}, N), \text{DLP}, \text{SIL}) /\ \backslash \\
\text{LPL-MISSING} := (\text{getLogHoles}(\text{LEL}) :: \text{addLogPositionsBetween}(\text{lpos}(s, N1), \text{LATEST})) \\
\]

\[
\begin{align*}
\text{op} \ & \text{updateMreadState} : \text{SiteId EntityId LogPosition ReadState} \rightarrow \text{ReadState} . \\
\text{eq} \ & \text{updateMreadState}(\text{SENDER}, \text{EID}, \text{lpos}(N1), \text{majorityRead}(\text{EID}, \text{lpos}(N2), \text{SIL})) = \\
& \quad \text{if} (N2 > N1) \text{then} \\
& \qquad (\text{majorityRead}(\text{EID}, \text{lpos}(N2), \text{SIL} :: \text{SENDER})) \\
& \quad \text{else} (\text{majorityRead}(\text{EID}, \text{lpos}(N1), \text{SENDER} :: \text{SIL})) \\
& \quad \text{fi} . \\
\end{align*}
\]

\[
\begin{align*}
\text{eq} \ & \text{updateMreadState}(\text{SENDER}, \text{EID}, \text{LP}, \text{majorityRead}(\text{EID}, \text{noLogPosition}, \text{emptySiteIdList})) = \\
& \quad \text{majorityRead}(\text{EID}, \text{LP}, \text{SENDER}) . \\
\]

\[
\begin{align*}
\text{crl} \ & [\text{rcvLateMajorityReadResponse}] : \\
& \quad (\text{msg} \text{majorityReadResponse}(\text{EG}, \text{TID}, \text{LP}) \text{from} \text{SENDER} \text{to} \text{THIS}) \\
& \quad < \text{THIS} : \text{Site} | \\
& \quad \text{localTransactions} : \text{LOCALTRANS} > \\
& \quad => \\
& \quad < \text{THIS} : \text{Site} | > \\
& \quad \text{if} (\text{not} \text{inMajorityRead}(\text{TID}, \text{EG}, \text{LOCALTRANS})) . \\
\end{align*}
\]

\[
\begin{align*}
\text{op} \ & \text{inMajorityRead} : \text{TransId EntityGroupId Configuration} \rightarrow \text{Bool} . \\
\text{eq} \ & \text{inMajorityRead}(\text{TID}, \text{EG}, < \text{TID} : \text{Transaction} | \text{readState} : \text{majorityRead}(\text{entity}(\text{EG}, N), \text{DLP}, \text{SIL}) ; \\
& \quad \text{RSTATE} > \text{LOCALTRANS}) = \text{true} . \\
\text{eq} \ & \text{inMajorityRead}(\text{TID}, \text{EG}, \text{LOCALTRANS}) = \text{false} [\text{wise}] . \\
\end{align*}
\]

\[
\begin{align*}
\text{op} \ & \text{hasQuorum} : \text{Nat SiteIdSet} \rightarrow \text{Bool} . \\
\text{eq} \ & \text{hasQuorum}(N, \text{SIS}) = (N \geq (\text{size}(\text{SIS}) \text{quo} 2 + 1)) . \\
\end{align*}
\]

\[
\begin{align*}
\text{op} \ & \text{getVersion} : \text{LogPosition EntityVersionList} \rightarrow \text{EntityVersion} . \\
\text{ceq} \ & \text{getVersion}(\text{lpos}(N), \text{EVERSIONS} :: \text{lpos}(N1) \text{VAL1} :: \text{lpos}(N2) \text{VAL2} :: \text{EVERSIONS}' = (\text{lpos}(N1) \text{VAL1}) \\
& \quad \text{if} (N1 < N / N < N2) . \\
\text{ceq} \ & \text{getVersion}(\text{lpos}(N), \text{EVERSIONS} :: \text{lpos}(N1) \text{VAL1}) = (\text{lpos}(N1) \text{VAL1}) \text{if} (N1 <= N) . \\
\end{align*}
\]

--- If majority--read timed out, restart

\[
\begin{align*}
\text{rl} \ & [\text{restartCatchup}] : \\
& \quad < \text{THIS} : \text{Site} | \text{entityGroups} : < \text{EG} : \text{EntityGroup} | \text{replicas} : \text{EGRS} > \text{EGROUPS}, \\
& \quad \text{localTransactions} : < \text{TID} : \text{Transaction} | \\
& \quad \text{operations} : \text{cr}(\text{entity}(\text{EG}, N)) :: \text{OL}, \text{status} : \text{transTimer}(U), \\
& \quad \text{readState} : \text{majorityRead}(\text{entity}(\text{EG}, N), \text{DLP}, \text{SIL}) ; \text{RSTATE} > \text{LOCALTRANS} > \\
& \quad => \\
& \quad < \text{THIS} : \text{Site} | \text{entityGroups} : < \text{EG} : \text{EntityGroup} | > \text{EGROUPS}, \\
& \quad \text{localTransactions} : < \text{TID} : \text{Transaction} | \\
& \quad \text{operations} : \text{cr}(\text{entity}(\text{EG}, N)) :: \text{OL}, \text{status} : \text{transTimer}(\text{defTimeout}), \\
& \quad \text{readState} : \text{majorityRead}(\text{entity}(\text{EG}, N), \text{noLogPosition}, \text{emptySiteIdList}) ; \\
& \quad \text{RSTATE} > \text{LOCALTRANS} > \\
& \quad (\text{multiCast} \text{majorityRead}(\text{EG}, \text{TID}) \text{from} \text{THIS} \text{to} \text{getSites}(\text{EGRS}) \text{setminus} \text{THIS}) . \\
\end{align*}
\]

\[
\begin{align*}
\text{***( Perform catchup ***)} \\
\end{align*}
\]

--- Determine missing entries in a given log

\[
\begin{align*}
\text{op} \ & \text{getLogHoles} : \text{LogEntryList} \rightarrow \text{LogPositionList} . \\
\text{ceq} \ & \text{getLogHoles}((\text{TID1} \text{lpos}(N1) \text{MSID1 OL1}) :: (\text{TID2} \text{lpos}(N2) \text{MSID2 OL2}) :: \text{LEL}) = \\
& \quad \text{getLogHoles}((\text{TID2} \text{lpos}(N2) \text{MSID2 OL2}) :: \text{LEL}) \text{if} N2 =/= N1 . \\
\text{ceq} \ & \text{getLogHoles}((\text{TID1} \text{lpos}(N1) \text{MSID1 OL1}) :: (\text{TID2} \text{lpos}(N2) \text{MSID2 OL2}) :: \text{LEL}) = \\
& \quad \text{addLogPositionsBetween}(\text{lpos}(s, N1), \text{lpos}(\text{ad}(N2, 1))) :: \text{getLogHoles}((\text{TID2} \text{lpos}(N2) \text{MSID2 OL2}) :: \text{LEL} \\
\text{eq} \ & \text{getLogHoles}(\text{LEL}) = \text{emptyLPlist} .
\end{align*}
\]
210  C Real-Time Maude Model of Megastore-CGC

--- Utility function: Return a log-position list between the two positions N1 and N2 (including then both)
ceq addLogPositionsBetween(lpos(N1), lpos(N2)) =
  lpos(N1) \cup addLogPositionsBetween(lpos(N1), lpos(N2)) if N1 <= N2 .

--- Utility function: Get the most recent log entry.
--- Invariant: The log entry list always has one element
op getMostRecentLPos : LogEntryList -- LogPosition .
eq getMostRecentLPos =
  \{ logEntryList lpos(N1), lpos(N2) = emptyLPositionList \} [otherwise].

--- Upon receiving a catchup request, traverse the log position list representing missing entries, and respond with all entries present at this site
crl [rcvCatchupRequest] :
  (msg catchupRequest(EG, TID, LEL--RESP,)
   createMapEntry(getUpdateList(OCID, OLISTS), SENDER, < EG : EntityGroup | > EGROUPS)) from
   \(\text{THIS to SENDER}\)
if LEL--RESP = getPresentEntries(LPL, LEL) /\
  OCID = getOrderingClass(EG, < EG : EntityGroup | > EGROUPS) .

crl [rcvCatchupResponse] :
  (msg catchupResponse(EG, TID, LEL--RECEIVED, PREDLIST) from SENDER to \(\text{THIS}\))
  \(\text{LOCALTRANS} \rightarrow \) \(\text{LOCALTRANS}\) .
  \(\text{LOCALTRANS} \rightarrow \) \(\text{LOCALTRANS}\) .
  if \(\text{not catchingUp(TID, \text{LOCALTRANS})}\) .

op catchingUp : TransId Configuration -- Bool .
eq catchingUp(TID, TID : Transaction | readState : catchingUp(EID, SIL, LPL, MISSING--ORDERS) ; RSTATE > \(\text{LOCALTRANS}\) = true .
eq catchingUp(TID, \(\text{LOCALTRANS}\) = false [otherwise].
C Real-Time Maude Model of Megastore-CGC

removeObsoleteProposals

removeIfPresent

getNextSite

\text{NEW} := \text{NEXT} \text{NEW} \text{CATCHUP} \text{else none fi}

\text{if} \text{sendNotifyAbort} (\text{THIS, LOCALTRANS, removeObsoleteTrans(TID, EG, LEL=RECEIVED, LOCALTRANS)}) \text{if CATCHUP=COMPLETE} \text{then}
\text{(unicast catchupRequest(EG, TID, LPL=MISSING) from THIS to NEXT-SITE)}
\text{else}
\text{multicast majorityRead(EG, TID) from THIS to getSites(EGRS) setminus THIS)}
\text{fi}
\text{fi)

LPL=MISSING := getLogHoles(applyRemoteLogEntries(LEL=RECEIVED, LEL) \setminus
\text{CATCHUP=COMPLETE} := (LPL=MISSING =\setminus emptySiteIdList) \setminus
\text{NEW=TRANS=LOG} := applyRemoteLogEntries(LEL=RECEIVED, LEL) \setminus
\text{NEXT-SITE} := getNextSite(SIL) \setminus
\text{OCCID := getOrderingClass(EG, \text{nextSiteIdList} \setminus THIS)}
\text{\setminus
\text{AWAIT=ORDERS} := addNotOrdered(LEL=RECEIVED, EG, OCID, OCUPDATES) \setminus

\text{op} getSiteIdList \rightarrow \text{SiteIdList}.
\text{eq} \text{getNextSite(SIL \setminus SID) = SID}.
\text{eq} \text{getNextSite(\text{emptySiteIdList}) = emptySiteIdList}.

\text{op removeIfPresent: SiteIdIdList SiteIdIdList \rightarrow SiteIdIdList}.
\text{eq removeIfPresent(SID, SIL \setminus SID \setminus SIL') = SIL \setminus SIL'.}
\text{eq removeIfPresent(SID, SIL) = SIL \setminus \text{otherwise}}.

\text{op removeObsoleteProposals: LogEntryList PaxosProposalSet \rightarrow PaxosProposalSet}.
\text{eq removeObsoleteProposals((TID1 \text{ MSID1 OL1} :: LEL, proposal(SID, TID2, LP, OL, VAL=REQ, PN) ; PROPSET}) = removeObsoleteProposals(LEL, PROPSET}).
\text{eq removeObsoleteProposals((TID1 \text{ MSID1 OL1} :: LEL, accepted(SID, (TID2 LP MSID2 OL2), VAL=REQ, PN) ; PROPSET}) = removeObsoleteProposals(LEL, PROPSET)}.
\text{eq removeObsoleteProposals(noEntries, PROPSET) = PROPSET}.
\text{eq removeObsoleteProposals(\text{tID1 MSID1 OL1} :: LEL, PROPSET) = removeObsoleteProposals(LEL, PROPSET) \setminus \text{otherwise}}.
op removeObsoleteTrans: TransId EntityGroupId LogEntryList Configuration -> Configuration.
eq removeObsoleteTrans(TID1, EG, (TID2 LP MSID1 OL) :: LEL, LOCALTRANS) =
removeObsoleteTrans(TID1, EG, LEL, removeOthersForLogPosition(EG, LP, LOCALTRANS)).
eq removeObsoleteTrans(TID1, EG, noEntries, LOCALTRANS) = LOCALTRANS.

op merge: AwaitingOrderSet AwaitingOrderSet -> AwaitingOrderSet.
eq merge((Ocid TID EG LP) :: AOS1, (Ocid TID EG LP) :: AOS2) = merge(AOS1, (Ocid TID EG LP) :: AOS2).
eq merge(AOS1, AOS2) = AOS1 ; AOS2 [owise].
--- If some entry in "EntGroupUpdateList" exists in the "AwaitingOrderSet",
--- return "true". Otherwise, return false.
op checkMissingOrders: AwaitingOrderSet EntGroupUpdateList -> Bool.
eq checkMissingOrders((Ocid TID EG LP) :: AOS1, UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) :: UPDATELST2) = true.
eq checkMissingOrders(AOS1, UPDATELST1) = false [owise].

op applyRemoteLogEntries: LogEntryList LogEntryList -> LogEntryList.
--- We might receive multiple catchup—response (due to failures). If we already have the log entry,
simply ignore it.
eq applyRemoteLogEntries((TID lpos(N) SID OL) :: LEL—RECEIVED, LEL :: (TID lpos(N) SID OL) :: LEL') =
applyRemoteLogEntries(LEL—RECEIVED, LEL :: (TID lpos(N) SID OL) :: LEL').
--- Common case: Find the right hole in the log, and insert missing entry
ceq applyRemoteLogEntries((TID1 lpos(N1) MSID1 OL1) :: LEL—RECEIVED, LEL :: (TID2 lpos(N2) MSID2 OL2) ::
(TID3 lpos(N3) MSID3 OL3) :: LEL') =
applyRemoteLogEntries(LEL—RECEIVED, LEL :: (TID2 lpos(N2) MSID2 OL2) :: (TID1 lpos(N1) MSID1 OL1) ::
(TID3 lpos(N3) MSID3 OL3) :: LEL') if (N2 < N1 \ N1 < N3).
--- We're at the end of the local log, append all entries
ceq applyRemoteLogEntries((TID1 lpos(N1) MSID1 OL1) :: LEL—RECEIVED, LEL :: (TID2 lpos(N2) MSID2 OL2)) =
LEL :: (TID2 lpos(N2) MSID2 OL2) :: (TID1 lpos(N1) MSID1 OL1) :: LEL—RECEIVED if N1 > N2.
--- All entries applied, we are done
eq applyRemoteLogEntries(noEntries, LEL) = LEL.

op addPendingWrites: LogEntryList PendingWriteList -> PendingWriteList.
eq addPendingWrites((TID LP SID OL) :: LEL, PWL) = addPendingWrites(LEL, px(LP, idle, OL) :: PWL) if OL
= \= emptyOpList.
eq addPendingWrites((TID LP SID emptyOpList) :: LEL, PWL) = addPendingWrites(LEL, PWL).
eq addPendingWrites(noEntries, PWL) = PWL.

op addNotOrdered: LogEntryList EntityGroupId OrderClassId OrderClassUpdates -> AwaitingOrderSet.
eq addNotOrdered((TID LP SID OL) :: LEL, EG, Ocid, (Ocid — EGIDS |> UPDATELST1 :: ((TID EG LP TENTATIVE)
 | DEFUP) :: UPDATELST2) ; OCUPDATES) =
addNotOrdered(LEL, EG, Ocid, (Ocid — EGIDS |> UPDATELST1 :: ((TID EG LP TENTATIVE) ; DEFUP) ::
UPDATELST2) ; OCUPDATES).
eq addNotOrdered(noEntries, EG, Ocid, OCUPDATES) = noAwaitingOrderSet.
eq addNotOrdered((TID LP SID OL) :: LEL, EG, Ocid, OCUPDATES) =
(Ocid TID EG LP) ; addNotOrdered(LEL, EG, Ocid, OCUPDATES) [owise].
--- If catchup—response timed out and we have sites to try, try the next site
rl [restartCatchup]:
< THIS : Site | entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
localTransactions : < TID : Transaction |
operations : cr(entity(EG,N)) :: OL, status : transTimer(0),
readState : catchingUp(entity(EG, N), SIL :: SID, LPL, MISSING—ORDERS) ; RSTATE
> =
< THIS : Site | entityGroups : < EG : EntityGroup > EGROUPS,
localTransactions : < TID : Transaction |
\begin{verbatim}
operations : cr(entity(EG,N)) :: OL, status : transTimer(defTimeout),
            readState : catchingUp(entity(EG, N), SIL, LPL, MISSING--ORDERS) ; RSTATE >
            LOCALTRANS

           (uniCast catchupRequest(EG, TID, LPL) from THIS to SID).

--- If no sites are available, restart the majority read
rl [restartCatchup] : < THIS : Site | entityGroups : < EG : EntityGroup |
          localTransactions : < TID : Transaction |
          operations : cr(entity(EG,N)) :: OL, status : transTimer(0),
          readState : catchingUp(entity(EG, N), emptySiteIdList, LPL, MISSING--ORDERS) ;
          RSTATE > LOCALTRANS

            (multiCast majorityRead(EG, TID) from THIS to getSites(EGRS) setminus THIS).

  op setValidated : EntityGroupId LogPosition EntGroupLogPosSet --> EntGroupLogPosSet .
  ceq setValidated(EG, lpos(N1), eglp(EG, lpos(N2))) ; CES = eglp(EG, lpos(N2)) ; CES if (N2 >= N1).
  ceq setValidated(EG, lpos(N1), eglp(EG, lpos(N2))) ; CES = eglp(EG, lpos(N1)) ; CES if (N2 < N1).
  ceq setValidated(EG, lpos(N1), invalidCstate(EG, lpos(N2))) ; CES = invalidCstate(EG, lpos(N2)) ; CES if (N1 < N2).
  ceq setValidated(EG, lpos(N1), invalidCstate(EG, lpos(N2))) ; CES = eglp(EG, lpos(N1)) ; CES if (N2 <= N1).
  eq setValidated(EG, lpos(N1), CES = eglp(EG, lpos(N1)) ; CES [ovise].

endtm)
\end{verbatim}

\section*{Listing C.9 \ megastore\_timing\_rtmaude}

\begin{verbatim}
(tomod MEGASTORE--TIMING is
  inc MEGASTORE--SETUP .
  inc UPDATES .

  var SID : SiteId .
  var TID : TransId .
  var EG : EntityGroupId .
  var SIS : SiteIdSet .
  var EGRS : EntityGroupReplicaSet .

  var T1 T2 T3 EXP : Time .
  var T1 T12 : TimeInf .
  var N : Nat .
  var OL : OperationList .

  var EGROUPS LOCALTRANS REMAININGTRANS : Configuration .
  var TS : TransStatus .
  var PREDMAP : DefReplicaPredMap .

  var PN PN' : Propnum .
  var DPN : DefPropnum .
  var PWL : PendingWriteList .
  var LE : LogEntry .

\end{verbatim}
var EG : EntityGroupId.
var LP : LogPosition.
var CES : EntGroupLogPosSet.
var VAL—REQ : Bool.
var PSTATE : PaxosStateSet.

***(Sites)***
eq mte(<SID : Site | coordinator : CES, entityGroups : EGROUP, localTransactions : LOCALTRANS>) =
  min([mte(EGROUP), mteTrans(EGROUP, LOCALTRANS, LOCALTRANS)]).
eq delta(<SID : Site | entityGroups : EGROUP, localTransactions : LOCALTRANS, T1>) =
  <SID : Site | entityGroups : delta(EGROUP, T1), localTransactions : delta(LOCALTRANS, T1)>

***(Transactions)***
op mteTrans : Configuration Configuration Configuration --> TimeInf.

--- Determine mte if TS == idle
cceq mteTrans(<EG : EntityGroup | pendingWrites : emptyPList > EGROUP, LOCALTRANS,
  <TID : Transaction | operations : cr(entropy(EG,N)) : OL, status : idle > REMAININGTRANS) = 0 if not
  inConflictWithRunning(EG, removeTid(TID, LOCALTRANS))

eq mteTrans(EGROUP, LOCALTRANS,
  <TID : Transaction | operations : w(EG:EntityId, EVAL:EntityValue) : OL, status : idle >
  REMAININGTRANS) = 0

eq mteTrans(EGROUP, LOCALTRANS,
  <TID : Transaction | operations : emptyOpList, status : idle > REMAININGTRANS) = 0.

eq mteTrans(EGROUP, LOCALTRANS,
  <TID : Transaction | status : executing(LP, T1) > REMAININGTRANS) = min(T1, mteTrans(EGROUP,
  LOCALTRANS, REMAININGTRANS)).

eq mteTrans(EGROUP, LOCALTRANS,
  <TID : Transaction | status : transTimer(T1) > REMAININGTRANS) =
  min(T1, mteTrans(EGROUP, LOCALTRANS, REMAININGTRANS)).

eq mteTrans(EGROUP, LOCALTRANS,
  <TID : Transaction | status : awaitOrder(T1) > REMAININGTRANS) =
  min(T1, mteTrans(EGROUP, LOCALTRANS, REMAININGTRANS)).

eq mteTrans(EGROUP, LOCALTRANS,
  <TID : Transaction | status : paxosState : PSTATE, status : in—paxos > REMAININGTRANS) =
  min([mte(PSTATE), mteTrans(EGROUP, LOCALTRANS, REMAININGTRANS)]).

eq mteTrans(EGROUP, LOCALTRANS, REMAININGTRANS) = INF [owise].
op mte : PaxosStateSet --> Time.
eq mte(acceptLeader(EG, LE, SID, T1) : PSTATE) = min(T1, mte(PSTATE)).
eq mte(acceptAll(EG, LE, VAL—REQ, PN, SIS, PREDMAP, T1) : PSTATE) = min(T1, mte(PSTATE)).
eq mte(restartPrepare(EG, LE, VAL—REQ, T1) : PSTATE) = min(T1, mte(PSTATE)).
eq mte(invalidating(EG, LE, VAL—REQ, PN, SIS, PREDMAP, T1) : PSTATE) = min(T1, mte(PSTATE)).
eq mte(PSTATE) = INF [owise].
op hasPrepareQuorum : Configuration PaxosState --> Bool.
ccq hasPrepareQuorum(<EG : EntityGroup | replicas : EGROUP, prepare(EG, LE, VAL—REQ, PN, PN',
  SIS, EXP) : PSTATE) =
  true if REPLICA—SiteIdSet := getSites(EGROPS) \ hasQuorum(size(SIS), REPLICA—SiteIdSet).
eq hasPrepareQuorum(EGROUP, PSTATE) = false [owise].
op removeTid : TransId Configuration --> Configuration.
eq removeTid(TID, <TID : Transaction | > REMAININGTRANS) = REMAININGTRANS.
eq removeTid(TID, LOCALTRANS) = LOCALTRANS [owise].
eq delta(<TID : Transaction | status : executing(LP, T2) >, T1) =
  <TID : Transaction | status :
  executing(LP, T2 minus T1) >.
\[
\begin{align*}
\text{eq:} & \quad \text{delta} \left( \langle \text{TID} : \text{Transaction} | \text{status} : \text{transTimer(T2)}, \text{T1} \rangle \right) = \langle \text{TID} : \text{Transaction} | \text{status} : \text{transTimer(T2) monus \text{T1}} \rangle . \\
\text{eq:} & \quad \text{delta} \left( \langle \text{TID} : \text{Transaction} | \text{status} : \text{awaitOrder(T2)}, \text{T1} \rangle \right) = \langle \text{TID} : \text{Transaction} | \text{status} : \text{awaitOrder(T2) monus \text{T1}} \rangle . \\
\text{eq:} & \quad \text{delta} \left( \langle \text{TID} : \text{Transaction} | \text{paxosState} : \text{PSTATE}, \text{status} : \text{in-paxos} \rangle, \text{T1} \right) = \langle \text{TID} : \text{Transaction} | \text{paxosState} : \text{delta(PSTATE, \text{T1})} \rangle . \\
\text{eq:} & \quad \text{delta} \left( \langle \text{TID} : \text{Transaction} | \text{status} : \text{TS}, \text{T1} \rangle \right) = \langle \text{TID} : \text{Transaction} | \text{status} : \text{TS monus \text{T1}} \rangle . \\

\text{op:} & \quad \text{delta} : \text{PaxosStateSet} \rightarrow \text{PaxosStateSet} . \\
\text{eq:} & \quad \text{delta} \left( \langle \text{EG} : \text{EntityGroup} | \text{pendingWrites} : \text{PWL} \rangle \right) = \text{mte}(\text{PWL}) . \\
\text{eq:} & \quad \text{delta} \left( \langle \text{EG} : \text{EntityGroup} | \text{pendingWrites} : \text{PWL}, \text{T1} \rangle \right) = \langle \text{EG} : \text{EntityGroup} | \text{pendingWrites} : \text{delta(PWL, \text{T1})} \rangle . \\

\text{op:} & \quad \text{mte} : \text{PendingWriteList} \rightarrow \text{Time} . \\
\text{eq:} & \quad \text{mte} \left( \langle \text{PWL} \rangle \right) = \text{INF} . \\
\text{eq:} & \quad \text{mte} \left( \langle \text{PWL} \rangle : \text{pw}((\text{LP}, \text{idle, OL})) \right) = 0 . \\
\text{eq:} & \quad \text{mte} \left( \langle \text{PWL} \rangle : \text{pw}((\text{LP}, \text{idling}, \text{OL})) \right) = \text{T1} . \\
\end{align*}
\]

\[\text{Listing C.10 validation.rtaude}\]

\[
\text{(omod VALIDATION is} \\
\text{inc UPDATES.)} \\
\text{var EG EG2 : EntityGroupId.} \\
\text{var EID : EntityId.} \\
\text{var SID SID1 SID2 : SiteId.} \\
\text{var SIS : SiteIdSet.} \\
\text{var PROPS : PaxosProposalSet.} \\
\text{var LP LP1 LP2 : LogPosition.} \\
\text{var SIS : SiteIdSet.} \\
\text{var EGS : EntityGroupIdSet.} \\
\text{var ES : EntitySet.} \\
\text{var EGROUPS : Configuration.} \\
\text{var EVERSIONS : EntityVersionList.} \\
\text{var OBDCLASS : EntityGroupIdSet.} \\
\text{var UPS1 UPS2 : EntGroupUpdateSet.} \\
\text{var DEFUP : DefEntGroupUpdate.} \\
\text{var UP1 : EntGroupUpdate.)} \\
\]
vars UPDATELST1 UPDATELST2 : EntGroupUpdateList.
var TID : TransId.
var N : Nat.
var T : Time.
var PN : Propnum.
var EVAL : EntityValue.
var OL OL1 OL2 : OperationList.
var LEL : LogEntryList.
var OCID : OrderClassId.
var OCCUPDATES : OrderClassUpdates.
vars VAL—REQ TENTATIVE : Bool.

***
Validation procedure:

1. Find the most recent eg/logpos—pair in readset according to the local order
2. Verify that every other eg/logpos—pair in the readset is the most recent according to the local order

***

cq isValid?(TID, ES, UPDATELST1 : (TID EG LP TENTATIVE ; DEFUP) :: UPDATELST2, EGROUPS) =
verifyMostRecentReadVersions(ES, UPDATELST1, EGROUPS)
if not isTentative?(UPDATELST1).
cq isValid?(TID, ES, UPDATELST1, EGROUPS) =
verifyMostRecentReadVersions(ES, UPDATELST1, EGROUPS)
if not (containsTid(TID, UPDATELST1) or isTentative?(UPDATELST1)).
cq isValidReadOnly?(ES, UPDATELST1, EGROUPS) =
verifyMostRecentReadVersions(ES,
getPrefix(getMaximum(ES, noEntGroupUpdate, UPDATELST1), UPDATELST1), EGROUPS)
if not isTentative?(UPDATELST1).

op isTentative? : EntGroupUpdateList —> Bool.
eq isTentative?(UPDATELST1 :: tentativeMarker :: UPDATELST2) = true.
eq isTentative?(UPDATELST1) = false [wise].

op verifyMostRecentReadVersions : EntitySet EntGroupUpdateList Configuration —> Bool.
eq verifyMostRecentReadVersions((entity(EG.N) —> (LP EVAL)) : ES, UPDATELST1,
(< EG : EntityGroup | proposals : PROPSET, transactionLog : LEL > EGROUPS)) =
(if (not containsRunningUpdate(entity(EG.N), UPDATELST1, PROPSET)) and
isMostRecentInSnapshot?(entity(EG.N), LP, UPDATELST1, LEL)
then
verifyMostRecentReadVersions(ES, UPDATELST1, < EG : EntityGroup | transactionLog : LEL > EGROUPS )
else
false [f1].
eq verifyMostRecentReadVersions(ES, noEntGroupUpdate, EGROUPS) = true.
eq verifyMostRecentReadVersions(emptyEntitySet, UPDATELST1, EGROUPS) = true.

op containsRunningUpdate : EntityId EntGroupUpdateList PaxosProposalSet —> Bool.
eq containsRunningUpdate(ES, UPDATELST1 :: (TID EG LP TENTATIVE ; DEFUP) :: UPDATELST2,
accepted(SID1, (TID LP2 SID2 (GL1 :: w(entity(EG.N), EVAL) :: OL2)), VAL—REQ, PN :: PROPSET) =
true.
eq containsRunningUpdate(ES, UPDATELST1, PROPSET) = false [wise].

op getPrefix : EntGroupUpdateSet EntGroupUpdateList —> EntGroupUpdateList.
eq getPrefix(TID EG LP TENTATIVE, UPDATELST1 :: (TID EG LP TENTATIVE ; UPS1) :: UPDATELST2) =
UPDATELST1 :: (TID EG LP TENTATIVE ; UPS1).
eq getPrefix(noEntGroupUpdate, UPDATELST1) = noEntGroupUpdate.

op isMostRecentInSnapshot? : EntityId LogPosition EntGroupUpdateList LogEntryList —> Bool.
cq isMostRecentInSnapshot?(entity(EG.N), LP, UPDATELST1, LEL) = true
if (getMostRecentUpdate(entity(EG, N), UPDATELST1, LEL) == LP or
   getMostRecentUpdate(entity(EG, N), UPDATELST1, LEL) == noLogPosition).
eq isMostRecentInSnapshot?(entity(EG, N), LP, UPDATELST1, LEL) = false [owise].

op getMostRecentUpdate : EntityId EntGroupUpdateList LogEntryList -> LogPosition.
ceq getMostRecentUpdate(entity(EG, N), UPDATELST1, LEL) = (TID LP SID OL)
   if not (containsUpdate(entity(EG, N), OL) and containsEntry(EG, LP, UPDATELST1)).
ceq getMostRecentUpdate(entity(EG, N), UPDATELST1, LEL) = LP
   if containsUpdate(entity(EG, N), OL) and containsEntry(EG, LP, UPDATELST1).
eq getMostRecentUpdate(entity(EG, N), UPDATELST1, noEntries) = noLogPosition.

op containsEntry : EntityGroupId LogPosition EntGroupUpdateList -> Bool.
eq containsEntry(EG, LP, UPDATELST1) = ((TID EG LP Tentative) ; UPS1) :: UPDATELST2 = true.
eq containsEntry(EG, LP, UPDATELST1) = false [owise].

op getMaximum : EntitySet DefEntGroupUpdate EntGroupUpdateList -> DefEntGroupUpdate.
eq getMaximum((entity(EG, N) -> (LP EVAL)) ; ES, DEFUP, UPDATELST1 = ((TID EG LP Tentative) ; UPS1) ::
   UPDATELST2) =
   getMaximum(ES, (TID EG LP Tentative), UPDATELST2).
eq getMaximum(emptyEntitySet, DEFUP, UPDATELST1) = DEFUP.
eq getMaximum((entity(EG, N) -> (LP EVAL)) ; ES, DEFUP, UPDATELST1)
   [owise].

endom

Listing C.11 predicates.rtmaude

***
Things to verify:

- Liveness: That the system keeps running

- Safety:
  1) Given no failures, all sites will eventually contain the same entity state

  2) Serializability: For any pair of transactions T1, T2 where T2
     reads an entity E updated by T1 in EG. There is no other
     transaction T3 who as updated E, and where T1 < T3 < T2 according to the transaction order of EG.

***

**( Predicates ***)
(tomod PREDICATES -- FOR -- MODEL -- CHECKING is
  inc INIT - STATE.
  inc TIMED - MODEL -- CHECKER.

vars SID1 SID2 : SiteId.
var EG1 : EntityGroupId.
var ES1 ES2 : EntitySet.
var SYSTEM REST LOCALTRANS EGROUPS1 EGROUPS2 : Configuration.
var M : Msg.
var T : Time.
vars TLOG1 TLOG2 : LogEntryList.
var TID : TransId.
var GRAPH : SerGraph.
var LP : LogPosition.
var EGLP : EntGroupLogPosSet.
Listing C.12  fault_injection.rtmaude

\begin{verbatim}
{\texttt{\textbf{op} allTransFinished : \texttt{-> Prop [ctor].}}}
  \texttt{eq \{ initTransactions REST \} = allTransFinished = false.}
  \texttt{eq \{ < SID1 : Site \localTransactions : < TID : Transaction \> LOCALTRANS > REST \} = allTransFinished = false.}
  \texttt{eq \{ \texttt{diy(M, T) REST} \} = allTransFinished = false.}
  \texttt{eq \{ SYSTEM \} = allTransFinished = true [wise].}

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

\texttt{op entityGroupsEqualOrInvalid : \texttt{-> Prop [ctor].}}
  \texttt{ceq \{ < SID1 : Site \| coordinator : eglp(EG1, LP) ; EGLP, entityGroups : < EG1 : EntityGroup \entitiesState : ES1 > EGROUPS1 > \} \| < SID2 : Site \| coordinator : eglp(EG1, LP) ; EGLP, entityGroups : < EG1 : EntityGroup \entitiesState : ES2 > EGROUPS2 > \} = entityGroupsEqual = false if ES1 =/= ES2.}

\texttt{op transLogsEqualOrInvalid : \texttt{-> Prop [ctor].}}
  \texttt{ceq \{ < SID1 : Site \| coordinator : eglp(EG1, LP) ; EGLP, entityGroups : < EG1 : EntityGroup \transactionLog : TLOG1 > EGROUPS1 > \} \| < SID2 : Site \| coordinator : eglp(EG1, LP) ; EGLP, entityGroups : < EG1 : EntityGroup \transactionLog : TLOG2 > EGROUPS2 > \} = transLogsEqual = false if TLOG1 =/= TLOG2.}

\texttt{endtom)
\end{verbatim}

\texttt{\textbf{inc} TIMED\texttt{--BEHAVIOR.}}
\texttt{inc MEGASTORE\texttt{--SETUP.}}
\texttt{inc UPDATES.}}
\texttt{inc UPDATE\texttt{--FAULT\texttt{--HANDLERS.}}
\texttt{inc NETWORK\texttt{--MODEL.}}
\texttt{inc CLIENT\texttt{--INTERFACE.}}
var REST : Configuration.
var OBJECT : Object.
var O : Oid.
var OL : OperationList.
var CES : EntGroupLogPosSet.

op failed : Object --> Object [ctor frozen(1)].

eq mte(failed(OBJECT)) = INF.
eq delta(failed(OBJECT), T) = failed(OBJECT).

class MsgShark | start : Time, end : Time.

eq mte(< O : MsgShark | start : T1, end : T2 >) = T2.
eq delta(< O : MsgShark | start : T1, end : T2 >, T) =
  < O : MsgShark | start : T1 monus T, end : T2 monus T >.

*** Failure injection for model checking ***

op ttf : --> Time.
op ttr : --> Time.

msg siteFailure : SiteId --> Msg.
msg siteRepair : SiteId --> Msg.

rl [takeSiteDown]:
siteFailure(SID)
  < SID : Site | >
  => failed(< SID : Site | >)
  dly(siteRepair(SID), ttr).

rl [bringSiteUp]:
  (siteRepair(SID))
  failed(< SID : Site | >)
  =>
  < SID : Site | >.

crl [msgWhenSiteFailure]:
  (msg MC from SENDER to SID)
  failed(< SID : Site | >)
  =>
  failed(< SID : Site | >)
  if not isInvalidateCoordinator(MC).

rl [expireMsgShark]:
  < 0 : MsgShark | end : 0 > --> none.

rl [captureMessage]:
  < 0 : MsgShark | start : 0 >
  (dly((msg MC from SENDER to SID), T2))
  =>
  none.

op isInvalidateCoordinator : MsgContent --> Bool.
eq isInvalidateCoordinator(invalidateCoordinator(EG, LP)) = true.
eq isInvalidateCoordinator(MC) = false [owise].

rl [newTrans]:
  (newTrans(SID, TID, OL))
  failed(< SID : Site | >)
  =>
  failed(< SID : Site | >).
A test setup for model checking Megastore-CGC with five transactions.

Listing C.13 mc_cgc_five_transactions.rtmaude

```plaintext
in validation.rtmaude
in megastore_timing.rtmaude

(tomod WORKLOAD is
  inc CLIENT-INTERFACE.
  inc NETWORK-MODEL.
  inc TESTSETUP-IDS.

  vars SID SID' : SiteId.
  eq possibleMsgDelays(SID, SID') = (30 ; 80).

  op value : Nat --> EntityValue [ctor].

  op transStartTime : --> NatSet.
  eq transStartTime = (180 ; 210).

  *** (Test transactions ***)
  op initTransactions : --> NEConfiguration.
  eq mte(initTransactions) = 0.

  vars N1 N2 N3 N4 N5 N6 : Nat.
  vars NS1 NS2 NS3 NS4 NS5 NS6 : NatSet.

  crl [delayTransactions] :
  initTransactions
  =>
    dly(newTrans(PARIS, T_L, cr(entity(EG1.0)) :: w(entity(EG1.0),v(3))), 150)
    dly(newTrans(NEW-YORK, T_M, cr(entity(EG2.0)) :: w(entity(EG2.0),value(4))), 150)
    dly(newTrans(PARIS, T_N, cr(entity(EG2.0)) :: w(entity(EG2.0),v(4))), 150)
    dly(newTrans(NEW-YORK, T_R, cr(entity(EG1.0)) :: cr(entity(EG2.0)) :: w(entity(EG2.0), v(2))), N1)
    dly(newTrans(PARIS, T_Q, cr(entity(EG1.0)) :: cr(entity(EG2.0)) :: w(entity(EG1.0), v(4))), N2)
  if (N1 ; NS1) := transStartTime \ (N2 ; NS2) := transStartTime .
endtom)

(tomod INIT-STATE is
  inc TIME-DOMAIN .
  inc UPDATES .
  inc VALIDATION .
  inc TRANSACTION-HISTORY .
  inc CLIENT-FOR-MODEL-CHECKING .
  inc MEGASTORE-TIMING .
  inc NETWORK-MODEL .
  inc TESTSETUP-IDS .
  inc WORKLOAD .
```

```
C Real-Time Maude Model of Megastore-CGC

*** (For random generation ***)
var SEED : Nat.
op rnd : => Oid.

*** (Simulation parameters ***)
---- Local reads
eq readDelay = 10.
---- Local update cost
eq updateDelay = 100.
---- Timeout for a Paxos–step
eq defTimeout = 500.
---- Time for a proposal to expire — if no decision, invalidate coordinator
eq defPropExp = 500.

*** (Entity group ids ***)
ops entitySetEG1-updated entitySetEG2-updated : => EntitySet.
eq entitySetEG1-updated =
  (entity(EG1,0) => (lpos(0) value(0)));
  (entity(EG1,1) => (lpos(0) value(0))).
eq entitySetEG2-updated =
  (entity(EG2,0) => (lpos(0) value(0)));
  (entity(EG2,1) => (lpos(0) value(0))).

*** (Sites ***)
op orderSitesReplicas : => EntityGroupReplicaSet.
eq orderSitesReplicas = (egr(LONDON, 0, lpos(0)) ; egr(PARIS, 1, lpos(0)) ; egr(NEW–YORK, 2, lpos(0))).

op eg1Replicas : => EntityGroupReplicaSet.
eq eg1Replicas = (egr(LONDON, 0, lpos(0)) ; egr(PARIS, 1, lpos(0)) ; egr(NEW–YORK, 2, lpos(0))).

op eg2Replicas : => EntityGroupReplicaSet.
eq eg2Replicas = (egr(PARIS, 0, lpos(0)) ; egr(NEW–YORK, 1, lpos(0))).

op entitySetOrderSites : => EntitySet.
eq entitySetOrderSites =
  (entity(OrderSites,1) => (lpos(0) (PARIS !> EG1 !> EG2)));
  (entity(OrderSites,2) => (lpos(0) (NEW–YORK !> OrderSites)))).

op entityGroupReplicaSets : => Configuration.
eq entityGroupReplicaSets =

ops entityGroupsLondon entityGroupsParis entityGroupsNewYork : => Configuration.
eq entityGroupsLondon =
  entityGroupReplicaSets
  < EG1 : EntityGroup | entitiesState : entitySetEG1-updated, replicas : eg1Replicas, proposals : emptyProposalSet, pendingWrites : emptyPWList, transactionLog : (initTrans lpos(0) PARIS emptyOpList) >.

eq entityGroupsParis =
  entityGroupReplicaSets
  < EG1 : EntityGroup | entitiesState : entitySetEG1-updated, replicas : eg1Replicas, proposals : emptyProposalSet, pendingWrites : emptyPWList, transactionLog : (initTrans lpos(0) PARIS emptyOpList) >.

eq entityGroupsNewYork =
  entityGroupReplicaSets
  < EG2 : EntityGroup | entitiesState : entitySetEG2-updated, replicas : eg2Replicas,
proposals : emptyProposalSet,
pendingWrites : emptyPWLList,
transactionLog : (initTrans lpos(0) NEW–YORK emptyOplList) > .

eq entityGroupsNewYork =

entityGroupReplicaSets
< EG1 : EntityGroup | entitiesState : entitySetEG1–updated, replicas : eg1Replicas,
proposals : emptyProposalSet,
pendingWrites : emptyPWLList,
transactionLog : (initTrans lpos(0) PARIS emptyOplList) >

< EG2 : EntityGroup | entitiesState : entitySetEG2–updated, replicas : eg2Replicas,
proposals : emptyProposalSet,
pendingWrites : emptyPWLList,
transactionLog : (initTrans lpos(0) NEW–YORK emptyOplList) > .

op initSites : -> Configuration .

eq initSites =

< PARIS : Site | coordinator : eglp(EG1, lpos(0)) ; eglp(EG2, lpos(0)) ; eglp(OrderSites, lpos(0)),
  awaitingOrder : noAwaitingOrderSet, seqGen : 0,
  egOrderings : ((entity(OrderSites,1) -- emptyOidSet |> (initTrans EG1 lpos(0) false) ;
  initTrans EG2 lpos(0) false)) ;
  (entity(OrderSites,2) -- emptyOidSet |> (initTrans OrderSites lpos(0) false))) ,
  entityGroups : entityGroupsParis, localTransactions : none >

< LONDON : Site | coordinator : eglp(EG1, lpos(0)) ; eglp(OrderSites, lpos(0)),
  awaitingOrder : noAwaitingOrderSet, seqGen : 0,
  entityGroups : entityGroupsLondon, localTransactions : none,
  egOrderings : ((entity(OrderSites,1) -- emptyOidSet |> (initTrans EG1 lpos(0) false)) ;
  (entity(OrderSites,2) -- emptyOidSet |> (initTrans OrderSites lpos(0) false))) >

< NEW–YORK : Site | coordinator : eglp(EG1, lpos(0)) ; eglp(EG2, lpos(0)) ; eglp(OrderSites, lpos(0))
  ,
  awaitingOrder : noAwaitingOrderSet, seqGen : 0,
  egOrderings : ((entity(OrderSites,1) -- emptyOidSet |> (initTrans EG1 lpos(0) false)) ;
  (initTrans EG2 lpos(0) false)) ;
  (entity(OrderSites,2) -- emptyOidSet |> (initTrans OrderSites lpos(0) false))) ,
  entityGroups : entityGroupsNewYork, localTransactions : none > .

op th : -> 0id .

op initTransHist : -> Configuration .

eq initTransHist =

< th : TransactionHistory | graph : emptyGraph,
  readers : emptyTransOpSet,
  writers : op(initTrans, entity(EG1,0), lpos(0)) ;
  op(initTrans, entity(EG1,1), lpos(0)) ;
  op(initTrans, entity(EG2,0), lpos(0)) ;
  op(initTrans, entity(EG2,1), lpos(0)) > .

eq initMegastore =

{ initSites
  initTransactions
  initTransHist
} .

endtom