GMU: Genuine Multiversion Update-Serializable Partial Data Replication

Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues

Abstract—In this article we introduce GMU, a genuine partial replication protocol for transactional systems, which exploits an innovative, highly scalable, distributed multiversioning scheme. Unlike existing multiversion-based solutions, GMU does not rely on any global logical clock, which may represent a contention point and a major impairment to system scalability. Also, GMU never aborts read-only transactions and spares them from undergoing distributed validation schemes. This makes GMU particularly efficient in presence of read-intensive workloads, as typical of a wide range of real-world applications. GMU guarantees the Extended Update Serializability (EUS) isolation level. This consistency criterion is particularly attractive as it is sufficiently strong to ensure correctness even for very demanding applications (such as TPC-C), but is also weak enough to allow efficient and scalable implementations, such as GMU. Further, unlike several relaxed consistency models proposed in literature, EUS shows simple and intuitive semantics, thus being an attractive consistency model for ordinary programmers. We integrated GMU in a popular open source in-memory transactional data grid, namely Infinispan. On the basis of a wide experimental study performed on heterogeneous platforms and using industry standard benchmarks (namely TPC-C and YCSB), we show that GMU achieves almost linear scalability and that it introduces reduced overhead, with respect to solutions ensuring non-serializable semantics, in a wide range of workloads.

Index Terms—Partial data replication, multiversioning, transactional systems, fault tolerance

1 Introduction

The advent of cloud computing has led to the proliferation of a new generation of in-memory, transactional data platforms, such as Cassandra [1], Sinfonia [2], or Infinispan [3]. In these platforms, data replication plays a fundamental role for both performance and fault-tolerance, since it allows for enhancing throughput by distributing the load and for ensuring data survival despite failures.

A common trait characterizing these platforms is the adoption of relaxed consistency models such as eventual consistency [4], restricted transactional semantics (e.g., single object transactions [1], or static transactions whose accessed dataset is pre-declared [2]), and non-serializable isolation levels [5]. The point is that classical transactional replication protocols ensuring strong consistency target fully replicated systems, where data is replicated synchronously across all nodes. As a consequence, these solutions are inherently non-scalable, as the synchronization overhead grows (at least) linearly with the number of nodes in the system. Also, full replication approaches based on two phase commit (2PC) are known to suffer from a rapid growth of the distributed deadlock rate as the number of nodes increases [6]. This issue is partially addressed by more recent transactional replication protocols (see, e.g., [7], [8]), which rely on total order communication primitives [9] precisely to avoid distributed deadlocks. However, as most of these protocols still adopt a full replication scheme, total order tends to become the scalability bottleneck.

Other proposals, e.g., [10], target partial replication scenarios, and are aimed at improving the level of scalability via the reliance on, so called, *genuine* replication. Genuineness maximizes scalability by ensuring that, for any transaction T, only the processes that replicate the data items read or written by T exchange messages to decide the final outcome (commit/abort) of T. Unfortunately, these solutions introduce a considerable overhead, as they require read-only transactions (that are largely predominant in typical applications' workloads [11]) to undergo expensive distributed validation phases.

In this paper, we present GMU, a Genuine Multiversion Update-serializable protocol which guarantees that read-only transactions are abort-free and never undergo any remote validation phase. The core of GMU is a distributed multiversion concurrency control algorithm, which relies on a novel vector clock [12] based synchronization mechanism to track, in a fully decentralized and scalable way, both data and causal dependencies among transactions.

GMU ensures the *Extended Update Serializability* (EUS) consistency criterion, originally introduced in [13] and further investigated in [11] (in the form of the *no-update-conflict-misses* property). EUS provides guarantees analogous to those offered by 1-Copy Serializability (1CS) for update transactions, thus ensuring consistent evolution of the system's state. Analogously to what happens with 1CS, with EUS read-only transactions are also guaranteed to observe a snapshot equivalent to some serial execution (formally, a linear extension [14]) of the history of update transactions. Further, EUS extends the guarantee of observing consistent

- *S. Peluso is with the ECE Department, Virginia Tech, Blacksburg, Virginia, USA. E-mail: peluso@vt.edu.*
- *P. Ruivo is with the Red Hat Inc, Dublin, Ireland. E-mail: pruivo@gsd.inesc-id.pt.*
- *P. Romano and L. Rodrigues are with the INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal. E-mail: romano@inesc-id.pt, ler@ist.utl.pt.*
- *F. Quaglia is with the Dipartimento di Ingegneria Informatica, Automatica, e Gestionale, Sapienza University of Rome, Italy. E-mail: quaglia@dis.uniroma1.it.*

Manuscript received 1 Jan. 2015; revised 20 Nov. 2015; accepted 7 Dec. 2015. Date of publication 22 Dec. 2015; date of current version 14 Sept. 2016. Recommended for acceptance by A.R. Butt.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2510998

1045-9219 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

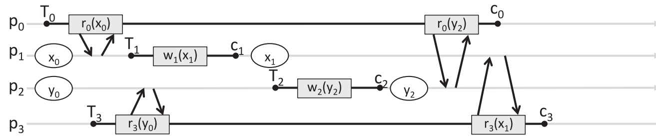


Fig. 1. In this history T_1 and T_2 neither conflict nor are causally dependent. The resulting execution, using GMU, is correct according to EUS but not according to 1CS.

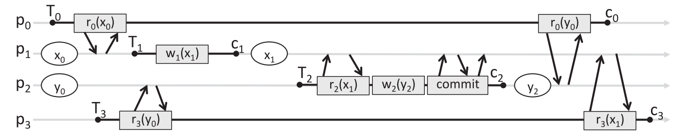


Fig. 2. In this history T_1 and T_2 conflict. The resulting execution, using GMU, is correct according to 1CS.

snapshots to transactions that have to be eventually aborted. Such a property, which is not guaranteed by 1CS, can be important for applications executing in non-sandboxed environments and which may behave erroneously upon observing non-serializable histories [15].

However, unlike 1CS, EUS allows concurrent read-only transactions to observe snapshots associated with different linear extensions of the history of update transactions. In other words, the only anomalies observable by read-only transactions are imputable to the re-ordering of update transactions that neither conflict (directly or transitively) on data, nor are causally dependent. Hence, the only discrepancies perceivable by end-users are associated with the ordering of logically independent concurrent events.

We illustrate the difference between 1CS and EUS using the example in Fig. 1. In this example we have four transactions T_0, T_1, T_2, and T_3 on nodes p_0, p_1, p_2, and p_3 respectively, where node p_1 stores the versions of datum x, and node p_2 stores the versions of datum y. For the sake of simplicity, in the examples we assume that each object is stored on a single node (although GMU is able to handle data replication across multiple nodes). T_1 and T_2 are concurrent and non-conflicting transactions (therefore, they can be serialized in any order with respect to each other) that commit two new versions of x and y respectively. Since the execution is genuine, p_1 (resp. p_2) does not contact any other node for the execution of T_1 (resp. T_2) because the accessed data are stored locally. Further two read-only transactions T_0 and T_3 execute concurrently by issuing two read operations on both x and y, and returning the last available version at the time each read operation is executed.

The reader can notice that such an execution would violate 1CS if both T_0 and T_3 committed, because T_1 would appear as serialized before T_2 from the point of view of T_3, while it would appear as serialized after T_2 from the point of view of T_0. Assume, for instance, that x and y were two prices, and that T_1 and T_2 doubled their values. Then, the clients/entities executing T_0 and T_3 may be confused about the relative order according to which the stock prices went up. This departure from 1CS, however, is perceivable only if clients of read-only transactions communicate with each other directly [16]; otherwise, EUS is as good as 1CS [11].

On the other hand, if T_1 and T_2 were dependent on one another, the values read by T_0 and T_3 would no longer be correct, even in EUS. This is illustrated in Fig. 2, where transaction T_2 reads from T_1. In fact, EUS mandates that the serialization orders witnessed by different read-only transactions correspond to *some* serial execution of the (partially ordered) history of update transactions. This implies that all read-only transactions agree on the order of update transactions that developed any (possibly indirect) data dependency. Therefore, in this case, both T_0 and T_3 must read values that are compatible with the correct serialization of T_1 before T_2. Therefore, T_0 can no longer read the last version of y because this version depends on the most recent version of x that was missed by T_0.

Interestingly, the reader will notice that the communication pattern induced and directly observed by transaction T_0 is the same in Figs. 1 and 2. This happens because both executions are *genuine*, i.e., in both the execution and the validation of any transaction only involves other nodes if they hold copies of the data being accessed. The challenge that is addressed by GMU is to derive a distributed protocol that can distinguish both scenarios while keeping executions genuine. As it will be explained in the paper, GMU differs from typical approaches, e.g., [17], [18], that are non-genuine as they propagate the commit of update transactions to all nodes, independently of the data that they accessed (such that transactions can be aware of other transactions that have committed before they start).

We have integrated GMU into Infinispan [3], which is an open-source transactional in-memory key-value store developed by Red-Hat. Given that the native replication mechanism implemented in Infinispan only supports non-serializable consistency (i.e., Repeatable Read (RR)), this system represents an ideal baseline to evaluate the additional costs incurred by GMU to ensure EUS consistency. Our experimental study of GMU has been based on TPC-C [19], an industry standard benchmark for OLTP systems, and YCSB [20], a recent benchmark for distributed key-value stores. The results of experiments conducted with up to 20 physical nodes and up to 100 virtual machines (VM) deployed on a public cloud infrastructure, show that our protocol achieves almost linear scalability, and provides stronger consistency than Repeatable Read at a marginal additional cost.

This article extends a conference version [21] by providing the correctness proof of GMU, presenting experimental results with a scaled up platform hosted on a public cloud system, and discussing aspects related to fault-tolerance, implementation's optimizations and data freshness.

The remainder of this article is structured as follows. In Section 2 we discuss related work. In Section 3 we present the target data model together with an overview of the EUS consistency criterion. The GMU protocol is presented in Section 4, while its correctness proof is provided in Section 5. The results of the experimental study are provided in Section 6. The Supplemental Material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2015.2510998, section discusses issues such as failure handling and data freshness in GMU, and efficient support for specific GMU operations.

2 Related Work

Most of the literature dealing with replication of transactional systems targets the case of full data replication, where

a copy of each data item is maintained at each site. In this context, we can find solutions addressing protocol specification [22], as well as proposing replication architectures based on middleware level approaches [23] and/or on extensions of the inner logic of transactional systems [22], [24]. Studies oriented at comparing the different approaches [25] have demonstrated how the proposals based on total order primitives, such as [26], exhibit the potential to improve performance. Also, total order based protocols relying on speculative transaction processing schemes, such as [8], [27], [28], have been shown to further reduce the impact of distributed synchronization. Compared to all these proposals, in this paper we take an orthogonal perspective and focus on partial data replication (as opposed to full replication).

As for solutions natively oriented to partially replicated transactional systems, they can be grouped depending on (i) whether they can be considered genuine, and on (ii) the specific consistency guarantees they provide. The proposals in [17], [18] cope with partial replication, but do not provide genuineness since transaction commitment requires interactions with all the sites, not only with those sites keeping copies of the data read/written by the transaction. Compared to these approaches, our proposal, being genuine, exhibits the potential for higher scalability.

The protocol in [10] provides a genuine solution supporting strict consistency, namely 1CS. However, differently from the present proposal, this protocol imposes that read-only transactions undergo a distributed validation phase. Also, these transactions are potentially subject to rollback/retry. Instead, our proposal only entails a local validation scheme, and never aborts read-only transactions. Overall, compared to the work in [10], our proposal provides a different trade-off between consistency (since it ensures EUS semantic) and performance (since it handles read-only transactions more efficiently).

Other solutions providing genuineness in partial replication can be found in [29], [30]. Like GMU, these approaches spare read-only transactions from distributed validations. Also, analogously to GMU, they adopt non-serializable consistency criteria. Particularly, these protocols ensure, respectively, Non-monotonic Snapshot Isolation [29] and Parallel Snapshot Isolation [30]. Unlike EUS, though, these consistency criteria do not guarantee serializability of the committed update transactions. On the contrary GMU guarantees that committed update transactions are fully isolated, admitting non-serializable schedules only for aborted or read-only transactions. In addition, the proposal in [30] serializes transactions at startup time. This design choice may impact freshness of data visible by (long-running) transactions, a problem which is tackled in GMU by attempting to advance the observed snapshot along transactions' execution.

Spanner [31] is related to GMU because it executes update transactions under strong consistency and it is also able to execute read-only transactions as snapshot transactions, namely at system-chosen timestamps and without locks thus not blocking conflicting concurrent writes. However Spanner relies on the TrueTime API to assign timestamps to commits and read operations, which exposes the absolute time and the uncertainty of the time measurement. Since Spanner slows down with higher uncertainty, TrueTime ultimately relies on specialized hardware components like clock references, i.e., GPS and atomic clocks, in order to keep uncertainty small. Consequently, as GMU only relies on logical clocks, it is employable in wider settings.

Our proposal has also relations with approaches for the management of distributed transactions in non-replicated systems, such as [11], [32], given that they support (E)US. The proposal in [32] exploits this semantic just to avoid aborting read-only transactions. GMU exploits this semantic for the same purposes, but targets partially replicated transactional systems, which gives rise to a protocol that addresses orthogonal issues as compared to the one in [32].

Our proposal is also related to Hindsight [33], a multiversion concurrency control mechanism for distributed software transactional memories. Differently from GMU, this protocol does not cope with (partially) replicated data. Also, it guarantees Snapshot Isolation, while we target EUS. Hence, our proposal guarantees serializability for update transactions, which is not guaranteed by the solution in [33].

3 Transaction and Consistency Model

A transaction is a totally ordered set of *begin*, *read*, *write*, *commit* and *abort* operations. Therefore, a transaction is sequential by nature and no multiple operations of a same transaction can be executed concurrently. Transactions that do not execute any write operation are called read-only transactions, otherwise they are called update (or equivalently write) transactions. The kth *write* operation of transaction T_i on a datum x, with $k = 1, 2, \dots$, is denoted with $w_i(x_{i.k})$, meaning that T_i creates the new version $x_{i.k}$ of x. Since the concurrency control scheme presented in this paper makes available to other transactions only the version $x_{i.n}$ produced by T_i's last write operation on x (if committed), intermediate versions of a same datum produced by earlier write operations of T_i are not visible. Hence, multiple writes on a same datum x by T_i are simply assumed to behave as a single write operation corresponding to the last executed one. The notation $w_i(x_i)$ is always adopted to denote a write operation of transaction T_i on x where $w_i(x_i) \equiv w_i(x_{i.n})$ such that $n = max\{k : \exists x_{i.k}\}$. Hence, a *read* operation by transaction T_j on x is denoted with $r_j(x_i)$, meaning that T_j has read version x_i created by T_i.

The first operation of a transaction is the *begin*, denoted as b_i to indicate that T_i starts its execution. The last operation of a transaction T_i is either a *commit* operation, denoted as c_i to indicate that T_i is completed successfully, or an *abort* operation, denoted as a_i. Also, there is at most one commit or abort operation per transaction.

A history $\mathcal{H}$ over a set of transactions $\mathcal{T}$ is constituted by: (a) a partial order of the events E that reflects the operations by the transactions belonging to $\mathcal{T}$; (b) a version order $\ll$, that is the total order defined by the creation of the versions for each datum d. Each event in $\mathcal{H}$ corresponds to the execution of an operation of a transaction in $\mathcal{T}$, e.g., $w_i(x_i)$, $r_i(x_j)$, c_i, b_i, a_i for $T_i, T_j \in \mathcal{T}$. The partial order E is such that: (i) it preserves the order of operations within a transaction; (ii) for any datum x and a pair of different transactions T_i, T_j, the event $r_j(x_i)$ is always preceded by the event $w_i(x_i)$ (but this does not mean that $w_i(x_i)$ is the last write operation on x before the execution of $r_j(x_i)$); (iii) for any datum x and

transaction T_j, if the event $r_j(x_i)$ is preceded by the event $w_j(x_{j.k})$ and no other write event on x is in between w_j and r_j then $x_i = x_{j.k}$. This means that E cannot revert the order of operations within a transaction, and it must ensure that a read operation returns a previously created version. Also, transactions are forced to observe the versions created by their own write operations. The version order $\ll$ defines the order of committed versions of any datum x such that $x_i \ll x_j$ iff T_i has committed before T_j.

Our target consistency criterion is Extended Update Serializability, also referred to as EUS, which was originally defined in the weaker form of Update Serializability by Hansdah and Patnaik [13] in terms of view serializability, and later re-formulated by Adya with the isolation level (E) PL-3U [11]. The interested reader can find a precise definition of EUS in Section 5.1. However, to follow our algorithm it is enough to understand that, roughly speaking, under EUS: (A) committed update transactions appear as executed sequentially and the resulting history is serializable; (B) every transaction always observes a consistent state, i.e., generated by some linear extension of the history of committed update transactions; (C) two read-only or executing/aborted transactions may observe two states produced by two different linear extensions of the history of committed transactions, which differ in the serialization order of non-dependent transactions.

As a consequence, the only discrepancies in the serialization orders observable by read-only, executing and aborted transactions are imputable to the re-ordering of update transactions that do not conflict (directly or transitively) on data. In other words, the only perceivable anomalies are associated with the re-ordering of logically independent concurrent events. As noted by Adya [11], such anomalies can ever be perceived by users of an application, only if they can communicate via some external channel. Hence, we argue that EUS represents a viable correctness criterion for a wide range of real world applications.

4 The GMU Protocol

4.1 Target System

The GMU protocol assumes a classic distributed system model composed of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate through message passing and do not have access to either a shared memory or a global clock. Messages may experience arbitrarily long (but finite) delays, and no bound on relative process speeds or clock skews is assumed. We assume a fail-stop failure model, where processes may fail by crashing, but do not behave maliciously. A process that never crashes is correct, otherwise it is faulty.

We assume that the system is eventually synchronous, namely there exists a time t after which processes can communicate with one another in a bounded time interval. In addition, due to the result in [34], at least a majority of processes are assumed to be correct. This is to provide the system with a View Synchronous Group Communication Service (GCS) [35] that integrates two complementary services: *group membership* and *reliable communication*. The role of the membership service is to provide each participant with information about which process is active (hence being reachable) and which one has failed (hence being unreachable). Such information is called a *view* of the group of participants. We assume that the GCS provides a view-synchronous primary-component group membership service [36], which maintains a single agreed view of the group at any given time and provides processes with information on whether they belong to the primary component. Furthermore, we assume that the processes can send and receive point-to-point messages on reliable channels via the primitives $send(m)$ and $receive(m)$ in such a way that for any two correct processes p_i and p_j, if p_i sends message m to p_j then p_j eventually receives m.

Each process p_i stores a partial copy of data, for which we assume a simple key-value representation. Hence, a data item d is stored as a sequence of committed versions $ver = \langle val, vid \rangle$, all associated with a key k, which reflects the order of the committed write operations on d. We denote as $k.last$ the reference to the sequence of committed versions associated with any key k. Particularly, $k.last$ points to the latest committed version within the sequence.

The fields val and vid of a committed version ver are respectively a value of d and a logical identifier associated with the commit of ver. The logical identifier vid is a reference to a logical vector timestamp (or vector clock [12]), i.e., an array of integer non-negative numbers, which identifies the snapshot ver belongs to. For the sake of clarity, in the presentation we assume that the size of any vector clock is equal to $|\Pi|$, and we omit optimizations that can be applied in practice, such as [37], [38].

Given a sequence of versions associated with d and stored by a process p_i, the corresponding vid vector clocks have monotonically decreasing values of their ith entry, noted as $vid[i]$. The binary relation $\leq$ is used to define an ordering relation on vector clocks according to the following rules. The pair $\langle v_1, v_2 \rangle$ of vector clocks is in $\leq$, by also writing $v_1 \leq v_2$, if $\forall i, v_1[i] \leq v_2[i]$. Further, if there exists at least one index j such that $v_1[j] < v_2[j]$, where $<$ is the standard *less* relation defined for natural numbers, then $v_1 < v_2$ holds.

Data are subdivided across m partitions, and each partition is replicated across r processes (in other words, r represents the replication degree for each data item). The set $\Gamma = \{g_1, \ldots, g_j, \ldots, g_m\}$ denotes the set of m groups of processes, where g_j is the group of processes replicating the jth data partition. Each group is composed of exactly r processes, of which a majority is assumed to be correct. Groups are not required to be disjoint (they can have processes in common), and a process may participate to multiple groups, as long as $\bigcup_{j=1\ldots m} g_j = \Pi$. In addition, $replicas(S)$ denotes the set of processes that replicate the data partitions containing all the keys $k \in S$, which are also called owners of S. Note that this model allows for capturing a wide range of data distribution algorithms, such as the ones currently in use by several NoSQL transactional data stores, which rely on consistent hashing [1], [39] to: i) minimize data transfer upon joining/leaving of processes [40]; ii) avoid distributed lookups to retrieve the identities of the group of processes storing the replicas of the requested data items.

4.2 Data Structures

Each process p_i keeps three main data structures, namely *CommitQueue*, *CLog* and *LastPrepVC*.

CommitQueue is an ordered queue whose entries are tuples $\langle T, vc, s \rangle$ such that T is a transaction, vc is its current commit

vector clock, and s is a value in the domain {*pending*, *ready*}. These entries are ordered according to the ith entry of their vector clocks, i.e., $vc[i]$, and possible ties are assumed to be broken using deterministic functions (e.g., hash functions) taking as input the unique transaction identifier T. The semantic associated with the s field is the following: if s is equal to *pending*, then T is currently successfully prepared to commit on process p_i and is waiting for a final commit/abort decision from the transaction coordinator. We will refer the current vc associated with a pending transaction to as its *prepareVC*. The value *ready* for s means instead that (a) the transaction has already received the commit decision from the transaction coordinator and that (b) it has already been assigned a final vector clock. Such a final vc will be also referred to as the *commitVC* of the transaction. As it will be discussed in the following, a *ready* transaction T will be committed as soon as T becomes the top standing transaction in the *CommitQueue*.

CLog is a list that maintains, for each committed transaction, the triple $\langle T$, *commitVC*, *updatedKeys*$\rangle$, such that T is the identifier of a committed transaction, *commitVC* is the commit vector clock associated with T and *updatedKeys* is the set of keys of data locally stored by process p_i which T has updated. Therefore *commitVC* is the vector clock pointed to by the vid of each new version created by committing T, i.e., the last version associated with any key in the set *updatedKeys* upon the commit of T.

The elements in the *CLog* list kept by p_i are totally ordered according to the value *commitVC*$[i]$ and the order follows the sequence of commits of update transactions that updated some data associated with a key stored by p_i. The vector clock associated with the most recent update transaction committed on p_i is denoted as *CLog.mostRecentVC*, and *CLog* is initialized as containing only one element $\langle -, commitVC_{init}, - \rangle$, where $CLog.mostRecentVC = commitVC_{init} = [0, \ldots, 0]$.

LastPrepVC is a vector clock that is used during the prepare phase of a transaction in order to determine its *prepareVC*. GMU maintains the following invariant on *LastPrepVC*: it is always greater than or equal to the vector clock associated with the last committed transaction in *CLog*, i.e., $\forall j$, $LastPrepVC[j] \geq CLog.mostRecentVC[j]$.

4.3 Transaction Execution Phase

GMU stores the following information in the transactional context of any transaction T: (1) The transaction vector clock *VC*, namely an array of scalar (integer) logical timestamps, having cardinality equal to the number of processes in the system, which keeps track of the (data and causal) dependencies developed by the transaction during its execution. When T starts its execution at process p_i, its vector clock, denoted as *T.VC*, is initialized as equal to the *CLog.mostRecentVC* vector clock at process p_i. (2) The transaction read-set, denoted as *T.rs*, which stores the set of identifiers of the keys associated with data read by T. (3) The transaction write-set, denoted as *T.ws*, which stores, as a set of pairs $\langle$*key*, *value*$\rangle$, the identifiers of the keys associated with data updated by the transaction, and the corresponding written values. (4) An array of boolean values, denoted as *T.hasRead*, which has an entry for each process in the system, and whose jth entry stores the flag $\top$, i.e., true, if T has executed a read operation on some key, which has been served by retrieving data stored by process p_j; otherwise the entry stores the flag $\bot$, i.e., false. As we will discuss, setting *T.hasRead*$[j]$ (possibly $j = i$, where p_i is the process where T has been started) to the value $\top$ is done in order to indicate that T will not be allowed to read data versions produced by transactions committed at process p_j after the first read operation by T on a datum kept by p_j has already occurred. In other words, having *T.hasRead*$[j]$ set to $\top$ indicates that the snapshot of data kept by p_j, which is visible to T, has been determined.

Algorithm 1. Write and Read Operations (process p_i)

```
1:  void WriteTransaction T, Key k, Value val
2:      T.ws ← T.ws \ {⟨k, −⟩} ∪ {⟨k, val⟩}
3:
4:  Value ReadTransaction T, Key k
5:      if ⟨k, val⟩ ∈ T.ws then
6:          return val
7:      if p_i ∈ replicas({k}) then                ▷ k is local
8:          target ← {p_i}
9:      else                                        ▷ k is remote
10:         target ← replicas({k})
11:     send ReadRequest([k, T.VC, T.hasRead]) to all p_j ∈ target
12:     wait receive ReadReturn([val, VC*, last]) from any p_h ∈
        target
13:     T.hasRead[h] ← ⊤
14:     if last = ⊥ ∧ T.ws ≠ ∅ then
15:         throw ABORT
16:     T.VC ← max(T.VC, VC*)
17:     T.rs ← T.rs ∪{k}
18:     return val
```

The pseudocode describing transaction execution in GMU is reported in Algorithm 1. Write operations (see lines 1-2 of Algorithm 1) are simply handled by storing the key and the new value in the transaction write-set, or by overwriting the corresponding value in case that key was already updated by the transaction (hence already being present in its write-set). If a transaction T issues a read operation on key k at a process p_i, it is first checked whether T has already written k. In this case, the value stored in T's write-set is returned (see lines 5-6 of Algorithm 1). Otherwise, p_i determines whether the key to be read by transaction T is local or not (see lines 7-11 of Algorithm 1). If the data is local, the read is processed locally (in practice, in this case the read request is executed as a function call), otherwise the read request is sent to all replicas holding the key. In any case, a single response (local or remote) is enough to make progress. When the response is received, *T.hasRead*$[h]$ is set to $\top$ just to indicate that at least one read operation by transaction T has been served by process p_h (see lines 12-13 of Algorithm 1).

The read operation returns the value of the selected version, along with the vector clock VC^*, and a boolean flag that specifies whether the returned version of k is the most recently committed one. The case in which the flag is set to $\bot$ is perfectly acceptable for read-only transactions, which can be serialized in the past. However, it is not acceptable for update transactions, that are required to abort (see lines 14-15 of Algorithm 1).

If transaction T is not aborted, then its vector clock is updated via an entry-wise maximum operation involving the current $T.VC$ and the vector clock VC^* that has been returned by the read operation. This update of $T.VC$ reflects any update in the snapshot that shall be visible to T for future read operations, by capturing the happened before relationship [14] between the commit event of the transaction that wrote the version of data observed by T's read, and the corresponding read event of T (see line 16 of Algorithm 1). Finally, before returning the value of the requested key k to the application, k is added to T's read-set (see lines 17-18 of Algorithm 1).

4.4 Reading a Data Item

As seen before, during the execution of a transaction, data items are read by making a read request (that can be processed locally or remotely, depending on whether a local copy of the item exists or not). Upon receiving this request, any replica executes the steps depicted in Algorithm 2.

Algorithm 2. Version Visibility Logic (process p_i)

1: **upon receive** READREQUEST *[Key k, VC xactVC, bool[] has Read]* **from** p_j
2: **if** $\neg hasRead[i]$ **then**
3: **wait until** $CLog.mostRecentVC[i] \geq xactVC[i]$
4: $VisibleSet \leftarrow \{vc : \langle -, vc, - \rangle \in CLog \wedge \forall w(hasRead[w] = \top \Rightarrow vc[w] \leq xactVC[w])\}$
5: $MaxVC \leftarrow vc: \forall w, vc[w] = \max\{v[w]: v \in VisibleSet\}$
6: **else**
7: $MaxVC \leftarrow xactVC$
8: $ver \leftarrow k.last$
9: $last \leftarrow \top$
10: **while** $\exists w : (hasRead[w] = \top \wedge ver.vid[w] > MaxVC[w])$ **do**
11: $ver \leftarrow ver.prev$
12: $last \leftarrow \bot$
13: **send** READRETURN($[ver.val, MaxVC, last]$) **to** p_j

Essentially, this algorithm shows two distinct behaviors depending on whether process p_i, on which the execution is being carried out, already served a read request by transaction T, or not. This information is encoded in $T.hasRead$ (particularly in its ith entry), namely the last parameter passed in input to the read request. In case this flag is not raised, it means that we still need to determine which is the actual snapshot of data observable by T on process p_i (see line 2 in Algorithm 2). The determination of this snapshot takes place by initially waiting that all the commit operations involving data kept by p_i, which T depends on, are first completed by process p_i. This occurs when the value of the ith entry of the vector clocks within p_i's *CLog* becomes at least equal to the value of the ith entry of the vector clock currently associated with T upon the execution of the read operation on process p_i.

To ensure this, T is temporarily blocked waiting that $CLog.mostRecentVC[i] \geq xactVC[i]$ gets satisfied (see line 3 in Algorithm 2). Once the wait phase ends, we need to compute a new vector clock to be associated with T in order for it to correctly express that T read some data committed by process p_i, which in their turn may depend on specific commit operations that took place at other processes in the system. This information is encoded in the vector clocks of *CLog* at process p_i, so that for any entry $xactVC[w]$ (namely the wth entry of the $T.VC$ passed in input to the read algorithm), whose corresponding entry $hasRead[w]$ is set to $\top$, we initially determine the set *VisibleSet* of the vector clocks vc in p_i's *CLog* such that $vc[w] \leq xactVC[w]$ (see line 4 in Algorithm 2). This set corresponds to the set of snapshots of data that are currently visible to T, which may entail more than one snapshot. Once the set of visible snapshots *VisibleSet* is determined,[1] the read algorithm performs an entry-wise maximum operation to determine the unique snapshot to be associated with transaction T on the basis of the current read operation performed. This snapshot is encoded into the *MaxVC* vector clock (see line 5 in Algorithm 2).

On the other hand, if T already read data from p_i (hence being $hasRead[i]$ already set to the value $\top$ upon the execution of the read algorithm by process p_i), the operation of determining *MaxVC* on the basis of the commit-log information kept by p_i does not need to be carried out anymore. Hence, the transaction vector clock passed in input to the read algorithm is directly used to fill the value of *MaxVC* (see lines 6-7 in Algorithm 2).

According to this computation logic of *MaxVC*, the most recent local snapshot that is visible by T on process p_i is determined by discarding all the transactions $\bar{T}$ (hence the data versions they generated) such that $\bar{T}$'s commit event depends (either directly or transitively) on the set of events occurring after the first read operation issued by T on any node. Therefore, the remaining transactions T^* are the ones having their commit vector *commitVC* not exceeding the reading transaction's vector clock $T.VC$ on the entries fixed by $T.hasRead$, namely the entries of $T.VC$ whose corresponding entries of $T.hasRead$ are found to be already set to $\top$. The finally computed *MaxVC* is then used (see lines 8-13 of Algorithm 2) to determine the version of k that is visible to T, namely the most recent version ver of k that is committed by one among the transactions T^*.

Roughly speaking, the first time T issues a read on a node p_i, GMU serializes T after transactions T^*, establishing an upper bound *MaxVC* on the *freshness* of the snapshots that T can observe during subsequent reads. Specifically, *MaxVC* is computed by merging via a per-entry maximum operation the *commitVC* vector clocks (stored in *VisibleSet* in the pseudocode) associated with transactions T^*. Hence, GMU prevents T from observing versions committed, on any node, by transactions that have to be serialized after T.

4.5 Transaction Commit Phase

As already mentioned, GMU can commit read-only transactions without any kind of local or remote validation phase.

The scheme used to commit *update* transactions in GMU is specified by the pseudocode shown in Algorithms 3 and 4. GMU uses a Two Phase Commit protocol, involving *all and only* the processes in the set rep, i.e., the processes that replicate keys read or written by a committing transaction T, as well as the process that originated T. Exploiting 2PC,

1. An optimized implementation of the logic used to compute *VisibleSet* can be found in Section 1.2 of the Supplemental Material, available online.

GMU can use standard techniques to ensure transaction atomicity and to verify its compatibility with the history of committed (update) transactions. The latter goal is achieved by acquiring read/write locks on all keys read/written by the transaction, at all processes where these keys are stored, and then performing a validation of the transaction's read-set (see lines 2-4 of Algorithm 4). Note that, to cope with deadlock scenarios, a maximum latency is admitted for lock acquisition, which is achieved by bounding the lock-wait phase by a timeout such that when the timeout expires the acquisition terminates with a negative outcome.

Algorithm 3. Commit Phase (process p_i)

1: *bool CommitTransaction T*
2:   **if** $T.ws = \emptyset$ **then**
3:     **return** $\top$
4:   *commitVC* $\leftarrow$ *T.VC*
5:   *outcome* $\leftarrow \top$
6:   **send** PREPARE($[T]$) **to all** $p_j \in$ *replicas*$(T.rs \cup T.ws) \cup \{p_i\}$
7:   **for all** $p_j \in$ *replicas*$(T.rs \cup T.ws) \cup \{p_i\}$ **do**
8:     **wait receive** VOTE($[T.id, VC_j, res]$) **from** p_j **or timeout**
9:     **if** $res = \bot \vee$ **timeout then**
10:       *outcome* $\leftarrow \bot$
11:       **break**
12:     **else**
13:       *commitVC* $\leftarrow$ *max*(*commitVC*,VC_j)
14:   *xactVN* $\leftarrow$ *max*$\{$*commitVC*$[w] : p_w \in \Pi\}$
15:   **for all** $p_j \in$ *replicas*$(T.ws)$ **do**
16:     *commitVC*$[j] \leftarrow$ *xactVN*
17:   **send** DECIDE($[T$, *commitVC*, *outcome*$]$) **to all** $p_j \in$ *replicas*$(T.rs \cup T.ws) \cup \{p_i\}$
18:   **wait until** *T.completed* $= \top$
19:   **return** *T.outcome*
20:
21:   *bool validate*(*Set rs*, *VC xactVC*)
22:   **for all** $k \in rs$ **do**
23:     **if** *k.last.vid*$[i] >$ *xactVC*$[i]$ **then**
24:       **return** $\bot$
25:   **return** $\top$

The key innovative feature of GMU's commit algorithm, however, consists in the scheme employed to establish an agreement among the processes involved in the execution of transaction T, on the commit vector clock to be assigned to T. To this end, GMU blends into the 2PC messaging pattern a distributed consensus scheme that resembles the one used by Skeen's total order multicast algorithm [9].

When process p_i receives a prepare message for transaction T (and after its successful validation), it sends back with the VOTE message the proposal of a new vector clock for T. This proposal, i.e., *prepareVC*, is built starting from *LastPrepVC* to ensure that the commit of T on p_i does not precede any other transaction commit already occurred at p_i. If T is not going to update any key stored by p_i, assigning *LastPrepVC* to *prepareVC* is enough to maintain the aforementioned invariant (see line 9 of Algorithm 4); otherwise, if p_i stores a key that is contained in T's write-set, the i-th entry of *LastPrepVC* is incremented by 1 before performing the assignment (see lines 11-12 of Algorithm 4). In addition, in the latter case, the prepare phase is concluded by storing the triple $\langle T$, *prepareVC*, *pending* $\rangle$ into p_i's *CommitQueue* (see line 13 of Algorithm 4).

Algorithm 4. Commit Phase (process p_i)

1: **upon receive** PREPARE $[Transaction\ T]$ **from** p_j
2:   *outcome* $\leftarrow$ *getExclLocks*$(T.id, T.ws)$
3:   *outcome* $\leftarrow$ *outcome* $\wedge$ *getSharLocks*$(T.id, T.rs)$
4:   *outcome* $\leftarrow$ *outcome* $\wedge$ *validate*$(T.rs, T.VC)$
5:   **if** *outcome*$= \bot$ **then**
6:     *releaseLocks*$(T.id, T.ws, T.rs)$
7:     **send** VOTE($[T.id, T.VC$,*outcome*$]$) **to** p_j
8:   **else**
9:     *prepareVC* $\leftarrow$ *LastPrepVC*
10:     **if** $p_i \in$ *replicas*$(T.ws)$ **then**
11:       *LastPrepVC*$[i]++$
12:       *prepareVC* $\leftarrow$ *LastPrepVC*
13:       *CommitQueue.put*$(\langle T$, *prepareVC*, *pending* $\rangle)$
14:     **send** VOTE($[T.id$, *prepareVC*, *outcome*$]$) **to** p_j
15:
16: **upon receive** DECIDE *[Transaction T, VC commitVC, bool outcome]* **from** p_j
17:   **if** *outcome* $= \top$ **then**
18:     *LastPrepVC* $\leftarrow$ *max(LastPrepVC, commitVC)*
19:     **if** $p_i \in$ *replicas(T.ws)* **then**
20:       *CommitQueue.update*$(\langle T$, *commitVC*, *ready* $\rangle)$
21:     **else**
22:       *releaseSharedLocks*$(T.id, T.rs)$
23:       *T.outcome* $\leftarrow \top$
24:       *T.completed* $\leftarrow \top$
25:   **else**
26:     *CommitQueue.remove*(T)
27:     *releaseLocks*$(T.id, T.ws, T.rs)$
28:     *T.outcome* $\leftarrow \bot$
29:     *T.completed* $\leftarrow \top$
30:
31: **upon** $\exists \langle T, vc, s\rangle : \langle T, vc, s\rangle =$ *CommitQueue.head* $\wedge s = ready$
32:   **for all** $\langle k, val\rangle \in T.ws : p_i \in$ *replicas*$(\{k\})$ **do**
33:     *apply*(k, val, vc)
34:   *CLog.add*$(\langle T, vc, T.ws\rangle)$
35:   *CommitQueue.remove*(T)
36:   *releaseLocks*$(T.id, T.ws, T.rs)$
37:   *T.outcome* $\leftarrow \top$
38:   *T.completed* $\leftarrow \top$

The 2PC coordinator gathers all the proposed *prepareVC*, and performs two operations in order to derive the *commitVC* for the transaction (see lines 13-16 of Algorithm 3). First, it merges the *prepareVC*'s values with the current values of the *commitVC* to be associated with the transaction using the max operator, which outputs a vector clock having, for each of its entries j, the maximum of the jth entry of the vector clocks passed as input. This allows *commitVC* to keep track of the causal dependencies developed by T during its execution as well as by the most recently committed transactions at all the processes contacted by T. Next, the coordinator determines the common value to attribute to the entries of the *commitVC* related to the processes keeping keys that have been updated by the transaction. This is achieved by picking the maximum value among all the entries in the *prepareVC* vector clocks delivered to the coordinator by all the processes involved in the commit of the transaction.

At this point the coordinator sends back a decision to all the involved processes. Upon the receive of a decision, these processes update the entry associated with transaction T in *CommitQueue*, if any, whose vector clock is replaced with

commitVC and whose status is set to *ready* (see lines 19-20 of Algorithm 4). We note that only those processes that maintain keys updated by the transaction, namely the processes in *replicas(T.ws)*, logged the transaction record within their *CommitQueue* after the receipt of the prepare request by the coordinator, and after the successful acquisition of locks and validation (see lines 10-13 of Algorithm 4). In addition, in order to ensure that *LastPrepVC* on p_i is never less than the vector clock of the last committed transaction on p_i, *LastPrepVC* is updated accordingly by using the received *commitVC* (see line 18 of Algorithm 4). In the pseudocode we opted for having the coordinator always sending the decision for T to itself as well, so that it can reply to the transaction commit request for T as soon as the commit operation is finalized locally.

In order to finalize the commit of T locally, any process p_i waits until the entry associated with T (if any) has become the top standing one in its *CommitQueue* (recall that the *CommitQueue* at process p_i is ordered on the basis of the ith entry of the vector clocks it contains). This scheme ensures that if a process p_i commits a transaction with a local scalar timestamp (i.e., having the ith entry in its *commitVC*) equal to v, then the local scalar timestamps of the transactions that subsequently commit at p_i will be greater than v. Also, all the replicas in *replicas*$(T.ws)$ commit T by using the same *commitVC* and in the same total order with respect to all the other committed transactions.

We note that the merging of the causal histories encoded by the transaction's vector clocks and by all the gathered *prepareVC* guarantees that the total order of the commit events is propagated across chains of possibly transitively dependent transactions. This represents one of the key mechanisms leveraged by GMU in order to ensure 1CS of the history of update transactions.

If the transaction's coordinator receives at least one negative vote, i.e., with res equal to $\perp$, it sends to all the participants (including itself) a negative decision, i.e., with *outcome* equal to $\perp$, which triggers the transaction abort.

Note that, for simplicity, we presented GMU without including standard techniques to ensure the high availability of the coordinator state. We discuss how such techniques can be integrated with GMU in the Supplemental Material, available online.

4.6 Garbage Collection

The GMU logic can be extended with a distributed garbage collection protocol relying on background dissemination (e.g., via gossip [41] or piggybacking). Particularly, each process p_i can (periodically) compute its local *lower bound vector clock*, denoted as $lbVC_i$, by applying the minimum operator to the set of vector clocks associated with the active transactions on p_i, and can communicate it to the other processes. Then each process can determine a conservative estimate of the global lower bound vector clock, say *glbVC*, that delimits the set of snapshots that may be still visible by any active transaction in the system. It is possible to garbage collect any version with *vid* < *glbVC* without risking to remove versions that may be later requested.

Once a process p_i is aware of the current value of $lbVC_j$ at each process $p_j \in \Pi$, it computes *glbVC*, where $glbVC[w]$ is the minimum value among the ones stored in $lbVC_j[w]$. It can then safely detach and delete from *CLog* every entry $\langle T,$ *commitVC*, *updatedKeys*$\rangle$ such that *commitVC* < *glbVC*. Then for each key k in *updatedKeys* it can remove every version ver such that ver.*vid* < *glbVC* and ver is not the last committed version of k.

5 Correctness Proof

In this section we prove that GMU only produces Extended Update Serializable histories.

5.1 EUS Formalization

Extended Update Serializability has been formulated by Adya, with the isolation level (E)PL-3U, in terms of topological properties of a graph, called the Direct Serialization Graph (DSG) [11].

A Direct Serialization Graph $DSG(\mathcal{H})$ on a history $\mathcal{H}$ is a graph with a vertex V_{T_i} for each transaction T_i in $\mathcal{H}$ and an edge $V_{T_i} \rightarrow V_{T_j}$ for each pair of conflicting transactions T_i, T_j in $\mathcal{H}$. Two transactions are conflicting if they both access a common datum and at least one of these accesses is a write operation. The $DSG(\mathcal{H})$ contains three types of edges depending on the three types of conflicts, also called dependencies, that two transactions T_i, T_j can have in $\mathcal{H}$:

- T_j *directly read-depends on* T_i. $DSG(\mathcal{H})$ contains the read-dependence edge $V_{T_i} \xrightarrow{wr} V_{T_j}$ because there exists a datum x such that both $w_i(x_i)$ and $r_j(x_i)$ are in $\mathcal{H}$.
- T_j *directly write-depends on* T_i. $DSG(\mathcal{H})$ contains the write-dependence edge $V_{T_i} \xrightarrow{ww} V_{T_j}$ because there exists a datum x such that both $w_i(x_i)$ and $w_j(x_j)$ are in $\mathcal{H}$, and x_i precedes x_j according to the order defined by $\ll$ in $\mathcal{H}$.
- T_j *directly anti-depends on* T_i. $DSG(\mathcal{H})$ contains the anti-dependence edge $V_{T_i} \overset{rw}{\dashrightarrow} V_{T_j}$ because there exists a datum x such that both $r_i(x_k)$ and $w_j(x_j)$ are in $\mathcal{H}$, and x_k precedes x_j according to the order defined by $\ll$ in $\mathcal{H}$.

Unlike in its original definition [11], $DSG(\mathcal{H})$ is not restricted to committed transactions in $\mathcal{H}$, but also considers aborted and executing ones. On the contrary, to specify that the direct serialization graph has vertexes for only committed transactions in $\mathcal{H}$, we use the notation $DSG(\mathcal{H}^c)$.

EUS dictates different guarantees for update and read-only transactions. Committed update transactions are forced to appear as executed serially, i.e., their history is equivalent to a sequential history. On the other hand, read-only transactions, as well as executing and aborted ones, are only required to observe a state that could be generated by *any* (i.e., possibly different) linear extension of the history of committed update transactions. Specifically, following the definition in [11], a history $\mathcal{H}$ guarantees EUS if $\mathcal{H}$ proscribes the following anomalies:

- *G1a*. $\mathcal{H}$ contains the operations $w_i(x_i)$, $r_j(x_i)$ and a_i. This means that transactions T_j has read a version written by an aborted transaction T_i.
- *G1b*. $\mathcal{H}$ contains the operations $w_i(x_{i.k})$, $r_j(x_{i.k})$, and $w_i(x_{i.k})$ is not the last write $w_i(x_i)$ of T_i on x. This means that T_j has read an intermediate value of x.

- *G1c.* The $DSG(\mathcal{H}^c)$ graph built on the history $\mathcal{H}^c$ derived from $\mathcal{H}$ by removing aborted and executing transactions contains an oriented cycle of all dependency edges.
- *Extended G-update.* The $DSG(\mathcal{H}^{upc}_{T_i})$ graph built on the committed update transactions in $\mathcal{H}$ plus transaction T_i in $\mathcal{H}$ contains an oriented cycle with one or more anti-dependency edges.

Proscribing *G1a* and *G1b* means that every transaction can only observe a committed version. In addition proscribing $G1c$ and *Extended G-update* means that for any transactions T_i, T_j there is an *unidirectional flow of information* from T_i and T_j, i.e., if T_i depends on T_j it does not miss the effects of T_j and of all committed update transactions that T_j depends or anti-depends on [11].

The graph considered in the *Extended G-update* anomaly only includes committed update transactions and at most one additional transaction T_i belonging to one among the following categories: aborted, executing or read-only transactions. Therefore, EUS does not exclude that the DSG built on all the executed (or even only committed) transactions contains an oriented cycle with one or more anti-dependency edges. As a consequence, EUS allows scenarios such that for a pair of read-only (or also executing and aborted) transactions T_1 and T_2, there are two committed update transactions T_3 and T_4 such that: (i) T_1 depends on T_3 but misses the effects of T_4, and (ii) T_2 depends on T_4 but misses the effects of T_3. Notice that can happen only if T_3 and T_4 are not conflicting because EUS proscribes the anomaly *G1c*.

In our proof, we show that the protocol avoids the anomalies *G1a*, *G1b*, *G1c* and *Extended G-update*.

5.2 Structure of the Proof

To simplify the structure of the proof, but with no loss of generality, we treat both an executing and an aborted transaction at time t as a read-only transaction formed by its prefix up to time t, which contains all its read operations performed until time t, except the read operation which triggered the abort (if any). This is an admissible reduction since write operations are buffered during the execution of a transaction and are externalized (i.e., the associated updates are applied) only upon a successfully completed commit phase. Moreover, the read operation that triggers the abort of a transaction T can be safely discarded because it does not return any value to the application layer, and hence it does not generate any dependency in the $DSG(\mathcal{H})$ graph.

Since in GMU write operations are externalized only upon successfully committing the transactions, then the *G1a* anomaly is trivially avoided by the fact that a transaction T_j can never read a value written by an aborted transaction T_i; on the other hand the *G1b* anomaly is avoided because only the final modification produced by a transaction T on any key k is made visible after transaction T commits.

The rest of the proof is organized in two parts: the former proves that the *G1c* anomaly is avoided, namely that the *unidirectional flow of information* is guaranteed; the latter proves that the *Extended G-update* anomaly is avoided, namely that the *no-update-conflict-misses* property extended to executing and aborted transactions is guaranteed.

5.3 Unidirectional Flow of Information

As already described, for each history $\mathcal{H}$ containing committed, aborted and executing transactions, and a pair of transactions T_i, T_j in $\mathcal{H}$, there is an unidirectional flow of information from T_i to T_j if $DSG(\mathcal{H})$ does not contain any directed cycle consisting entirely of dependence edges from T_i to T_j. To prove that the last statement is true with GMU, we will show that the sub-graph $DSG(\mathcal{H}^{upc})$ does not contain any directed cycle consisting entirely of dependency edges, where $\mathcal{H}^{upc}$ is obtained by considering only the committed update transactions in $\mathcal{H}$. This simplification is admissible given that: (1) by excluding anti-dependence edges, read-only transactions are necessarily sink nodes of $DSG(\mathcal{H})$, i.e., they do not have any outgoing edge, as they can only develop incoming read dependence edges in this analysis; (2) each executing or aborted transaction can be treated as a read-only transaction. Subsequently we prove that for each edge $V_{T_i} \xrightarrow{wr} V_{T_j} \in DSG(\mathcal{H}^{upc})$ and $V_{T_i} \xrightarrow{ww} V_{T_j} \in DSG(\mathcal{H}^{upc})$, the relation $T_i.commitVC < T_j.commitVC$ holds, where $T_i.commitVC$ (respectively $T_j.commitVC$) is the vector clock used to commit transaction T_i (respectively T_j). Hence, $DSG(\mathcal{H}^{upc})$ cannot contain any oriented cycle with (write and read) dependence edges. Therefore, if an edge $V_{T_i} \xrightarrow{E} V_{T_j}$ is in $DSG(\mathcal{H}^{upc})$, we distinguish in the proof two cases, depending on whether T_j directly read-depends on T_i, i.e., $E = wr$, or T_j directly write-depends on T_i, i.e., $E = ww$.

5.3.1 T_j Directly Read-Depends on T_i

In this case T_j reads a value val associated with a version ver of some key k that has been committed by T_i on a process p_n. Let $T_j.VC$ be the vector clock associated with T_j after T_j has read from T_i, we prove that both $T_i.commitVC \leq T_j.VC$ and $T_j.VC < T_j.commitVC$ hold.

Lemma 5.1. *If transaction T_j directly read-depends on transaction T_i then T_i.commitVC $\leq T_j.VC$, where $T_j.VC$ is the vector clock associated with T_j right after T_j has read from T_i.*

Proof. Right after T_j has read from T_i, $T_j.VC$ is equal to the vector clock obtained by maximizing each entry of $T_j.VC$ with its counterpart in the VC^* vector clock (see line 16 of Algorithm 1), implying that $VC^* \leq T_j.VC$.

We have to distinguish two cases, depending on whether this is the first T_j's read operation served by process p_n, or not. In the former case, VC^* is equal to the vector clock $MaxVC$, computed by p_n relying on its $CLog$, such that

$$VC^* \leftarrow MaxVC \leftarrow vc : \forall w, vc[w] = max\{v[w] : v \in VisibleSet\} \quad (1)$$

(see line 5 of Algorithm 2) and where

$$VisibleSet \leftarrow \{vc \ : \ \langle -, vc, - \rangle \ \in CLog \wedge \forall w(hasRead[w] = \top \Rightarrow vc[w] \leq xactVC[w])\} \quad (2)$$

(see line 4 of Algorithm 2), where $xactVC$ corresponds to the value held by $T_j.VC$ right before the read operation. In addition, since T_j has read version ver of key k from T_i, by line 10 of Algorithm 2, the following condition is verified

$$\forall w(hasRead[w] = \top \Rightarrow T_i.commitVC[w] \leq MaxVC[w]) \quad (3)$$

and therefore

$$MaxVC \geq T_i.commitVC \tag{4}$$

because $T_i.commitVC \in VisibleSet$ (by Equations (1), (2) and (3)). On the other hand, since $T_i.commitVC \leq VC^*$ by Equations (1) and (4), and since $VC^* \leq T_j.VC$, it follows that $T_i.commitVC \leq T_j.VC$.

Consider now the case in which p_n has already processed some read of T_j. In this case, $T_i.commitVC \leq T_j.VC$ still holds because $T_i.commitVC$ was in the *VisibleSet* computed by T_j upon the first read on p_n. If this was not the case, it would mean that at the time of that first read there existed an index h such that $T_j.hasRead[h] = \top$ and $T_i.commitVC[h] > T_j.VC$. However, by the visibility rule at line 10 of Algorithm 2, $T_j.VC$ is always updated by means of a max operator and once $T_j.hasRead[h]$ is set to $\top$ then $T_j.VC[h]$ cannot change anymore. □

Lemma 5.2. *For each committed update transaction T_j, $T_j.VC_{MAX} < T_j$.commitVC, where $T_j.VC_{MAX}$ is the vector clock associated with T_j before T_j enters the commit phase.*

Proof. Let I be the set of the identifiers of the set Π of processes in the system. $T_j.VC_{MAX} < T_j.commitVC$ because:

(A) $T_j.VC_{MAX}[h] \leq T_j.commitVC[h]$, $\forall h \in I$, given that $T_j.commitVC$ is initialized with the values of $T_j.VC_{MAX}$ (see line 4 of Algorithm 3) and it is later modified by means of a *max* operator (see line 13 of Algorithm 3).

(B) $\forall s \in I$ such that T_j writes on a key maintained by process p_s, then the relation $T_j.VC_{MAX}[s] < T_j.commitVC[s]$ holds, given that $T_j.commitVC[s]$ is set to *xactVN*, which is a new scalar version number derived from the increment of the *LastPrepVC[s]* scalar clock on s (see lines 14-16 of Algorithm 3 and line 11 of Algorithm 4) and $T_j.VC_{MAX}[s] \leq LastPrepVC[s]$ before the increment. □

Lemma 5.3. *If a committed transaction T_j directly read-depends on a transaction T_i then T_i.commitVC $< T_j$.commitVC.*

Proof. Indicating again with $T_j.VC_{MAX}$ the vector clock associated with T_j before T_j enters the commit phase, we have that $T_j.VC \leq T_j.VC_{MAX}$ by construction. Also, given that $T_i.commitVC \leq T_j.VC$ by Lemma 5.1 and $T_j.VC_{MAX} < T_j.commitVC$ by Lemma 5.2, we have that $T_i.commitVC < T_j.commitVC$. □

5.3.2 T_j Directly Write-Depends on T_i

In this case T_j overwrites a key k already written by T_i.

Lemma 5.4. *If a committed transaction T_j directly write-depends on a transaction T_i then T_i.commitVC $< T_j$.commitVC.*

Proof. Since (i) a write is actually executed when a transaction commits, (ii) an exclusive lock for each key to be written is acquired during the prepare phase and (iii) all the locks are released at the end of the commit phase, then T_j commits after T_i has already committed and both T_i and T_j commit on at least a common node p_n (the node that stores the key overwritten by T_j). Also, by line 13 of Algorithm 3:

$$prepareVC_{j,n} \leq T_j.commitVC, \tag{5}$$

where $prepareVC_{j,n}$ is the prepare vector clock used to prepare T_j on node p_n. Moreover, since T_j gets prepared after T_i has already inserted $T_i.commitVC$ in p_n's *CLog* and $prepareVC_{j,n}$ is greater than any vector clock in that *CLog* (see lines 11-12 of Algorithm 4), it follows that

$$T_i.commitVC < prepareVC_{j,n}. \tag{6}$$

Hence, $T_i.commitVC < T_j.commitVC$ by (5) and (6). □

Theorem 5.5. *For each history $\mathcal{H}$ containing committed, aborted and executing transactions, $DSG(\mathcal{H})$ does not contain any oriented cycle with (write and read) dependence edges.*

Proof. As already pointed out (see Section 5.3) proving that the $DSG(\mathcal{H}^{upc})$ does not contain any directed cycle with dependency edges is a sufficient condition to prove that $DSG(\mathcal{H})$ does not contain such cycles. The history $\mathcal{H}^{upc}$ is obtained by removing all the aborted, executing and read-only transactions from $\mathcal{H}$. Therefore $DSG(\mathcal{H}^{upc})$ cannot contain any directed cycle with dependency edges because if such a cycle C existed, then, by Lemmas 5.3 and 5.4, there would be the absurd such that for each transaction T_i in C, $T_i.commitVC < T_i.commitVC$. □

5.4 No-Update-Conflict-Misses

The no-update-conflict-misses property is guaranteed if, for each history $\mathcal{H}$ and aborted/executing/committed transaction $T_i \in \mathcal{H}$, the $DSG(\mathcal{H}^{upc}_{T_i})$ containing all committed update transactions of $\mathcal{H}$ and transaction T_i does not contain any oriented cycle with anti-dependence edges. Since every aborted/executing transaction can be reduced to a read-only transaction (see the introductory discussion in Section 5), the following proof considers T_i as a generic (i.e., either read-only or update) committed transaction.

Lemma 5.6. *If a committed transaction T_j directly anti-depends on a transaction T_i then T_i.commitVC $< T_j$.commitVC.*

Proof. T_j directly anti-depends on T_i means that T_i reads a version of a key k older than the one created by T_j on k, and then it successfully commits. Since the protocol ensures that an update transaction T is not aborted if and only if (i) it successfully acquires the locks on all the keys in its write-set and read-set during the prepare phase and (ii) it passes the validation step, namely other transactions have not concurrently altered the version chains associated with the keys in T's read-set (see lines 21-25 of Algorithm 3), then T_i commits before T_j commits and the two sets of processes involved in the commit operations of these two transactions are not disjoint. This is the same scenario that has been dealt with in Lemma 5.4, where we have considered the case in which T_j directly write-depends on T_i. Therefore, by using identical arguments as the ones we used while proving Lemma 5.4, one can easily show that $T_i.commitVC < T_j.commitVC$. □

Lemma 5.7. *For each history $\mathcal{H}$ and read-only transaction $T^{RO} \in \mathcal{H}$, the $DSG(\mathcal{H}^{upc}_{T^{RO}})$ containing all committed update transactions of $\mathcal{H}$ and transaction T^{RO} does not contain any oriented cycle involving T^{RO}.*

Proof. As T^{RO} is a read-only transaction, it follows that any incoming edge from an update transaction T_i to T^{RO} must be a read-dependence edge, namely $V_{T_i} \xrightarrow{wr} V_{T^{RO}}$. Further, the only outgoing edges from T^{RO} to update transactions T_j must be anti-dependence edges, namely $V_{T^{Ro}} \overset{rw}{\dashrightarrow} V_{T_j}$.

If $DSG(\mathcal{H}^{upc}_{T^{RO}})$ contains an edge $V_{T_i} \xrightarrow{wr} V_{T^{RO}}$, then by Lemma 5.1 we have that $T_i.commitVC \leq T^{RO}.VC$, where $T_i.commitVC$ is the commit vector clock of T_i and $T^{RO}.VC$ is the vector clock associated with T^{RO} right after T^{RO} has developed the dependence on T_i (by reading a version that has been committed by T_i). On the other hand, if $DSG(\mathcal{H}^{upc}_{T^{RO}})$ contains an edge $V_{T^{Ro}} \overset{rw}{\dashrightarrow} V_{T_j}$, then by the definition of anti-dependence edge it follows that $\exists k \neq j$ and exists a key x such that T^{RO} has read a version x_k on a node p_n, T_j has committed a version x_j on p_n and $x_k \ll x_j$ (x_k is committed before x_j). In this case there always exists an index h such that $T^{RO}.VC_{MAX}[h] < T_j.$ *commitVC[h]*.

In particular, if at the time T^{RO} reads version x_k transaction T_j has already committed version x_j, then, according to the condition at line 10 of Algorithm 2, $\exists\ p_h$ from which T^{RO} has already read such that $T^{RO}.VC[h] < T_j.$ *commitVC[h]*. Hence, since $T^{RO}.VC[h]$ cannot change anymore after T^{RO} has read on p_h, $T^{RO}.VC_{MAX}[h] < T_j.$ *commitVC[h]* holds.

Otherwise, if T^{RO} misses T_j updates because its read operation on key x is delivered on p_n before T_j's commit, then $T^{RO}.VC[h] < T_j.commitVC[h]$ holds, where $h = n$, because the commit log *CLog* of process p_n is always totally ordered according to the commit vector clocks' values with index n. Also in this case $T^{RO}.VC_{MAX}[h] < T_j.$ *commitVC[h]* holds because $T^{RO}.VC_{MAX}[h]$ cannot change anymore after T^{RO} has read data from p_n.

No other cases are possible since the reading rule defined at line 3 of Algorithm 2 prevents the transaction T^{RO} from missing the updates of T_j on p_n when $T^{RO}.VC[n] \geq T_j.commitVC[n]$ holds, and $T^{RO}.VC_{MAX} \geq T^{RO}.VC$ holds by construction of $T^{RO}.VC_{MAX}$.

At this point, let us assume by contradiction that $DSG(\mathcal{H}^{upc}_{T^{RO}})$ has an oriented cycle involving T^{RO}. Without loss of generality such a cycle C can be assumed as follows: $V_{T_1} \to \cdots \to V_{T_i} \to V_{T^{RO}} \to V_{T_j} \to \cdots \to V_{T_1}$. Then by Lemmas 5.3, 5.4 and 5.6 it follows that $T_1.commitVC < T_i.commitVC$ and $T_j.commitVC < T_1.commitVC$. In addition $T_i.commitVC \leq T^{RO}.VC$, after T^{RO} has read a version committed by T_i, and there always exists an index h such that $T^{RO}.VC_{MAX}[h] < T_j.$ *commitVC[h]*. Since $T^{RO}.VC \leq T^{RO}.VC_{MAX}$ by construction, we would get the following absurd: $T_1[h] \leq T^{RO}[h] < T_j[h] \leq T_1[h]$.

Hence, the assumption that $DSG(\mathcal{H}^{upc}_{T^{RO}})$ contains cycles involving T^{RO} is contradicted and the claim follows. □

Finally, the no-update-conflict-misses property is guaranteed by the following Theorem.

Theorem 5.8. *For each history $\mathcal{H}$ and transaction $T_i \in \mathcal{H}$, the $DSG(\mathcal{H}^{upc}_{T_i})$ containing all committed update transactions of $\mathcal{H}$ and an arbitrary (read or update, committed, executing or aborted) transaction T_i does not contain any oriented cycle with at least an anti-dependence edge.*

Proof. If T_i is a committed update transaction, an oriented cycle with at least an anti-dependence edge cannot exist due to Lemmas 5.3, 5.4 and 5.6. If T_i is a read-only transaction, $DSG(\mathcal{H}^{upc}_{T_i})$ does not contain any cycle by Lemma 5.7. No other cases have to be considered because any executing or aborted transaction can be treated as a read-only transaction (see the introductory discussion in Section 5). □

6 Experimental Evaluation

We integrated GMU in Infinispan,[2] an open source in-memory distributed key-value store by Red Hat. Infinispan supports partial replication and uses a lightweight data placement scheme based on consistent hashing [39]. The strongest consistency level natively ensured by Infinispan is Repeatable Read [5], which is significantly weaker than (E)US as it allows the commit of (both read-only and update) transactions that observe non-serializable schedules [11]. Repeatability of read operations is instead guaranteed by storing the data items observed by read operations, and returning them upon subsequent reads, while an encounter based two phase locking scheme is applied to write operations. Also, classic 2PC is used for managing replicated data [6].

Designed to achieve high scalability and to support weak consistency models, Infinispan represents an ideal baseline to evaluate the costs incurred by GMU to provide stronger consistency guarantees. For comparison purposes, we also implemented a Non-Genuine Multiversion-based (NGM) replication scheme that, analogously to the one in [42], relies on a fully replicated, logically centralized, global scalar clock, to totally order committing update transactions.

In this study we used two well-known benchmarks, namely TPC-C [19] and YCSB [20]. The workload generated by TPC-C is representative of OLTP environments and is characterized by complex and heterogeneous transactions, with very skewed access patterns and high conflict probability. YCSB (Yahoo! Cloud Serving Benchmark) [20] is a framework specifically aimed at benchmarking NoSQL key-value data grids and cloud stores. The transactional profile of this benchmark is quite different from the one of TPC-C, with simpler, shorter transactions that rarely conflict.

We run experiments using two different platforms. One, FutureGrid (www.futuregrid.org), is a public distributed test-bed for cloud computing. This platform allows for evaluating GMU in environments representative of public cloud infrastructures, which are typically characterized by more competitive resource sharing, ample usage of virtualization technology, and relatively less powerful nodes. On FutureGrid, we used up to 100 virtual machines. Each VM was equipped with 7 GB RAM, two 2.93 GHz cores Intel Xeon CPU X5570, running CentOS 5.5 x86_64. All the VMs were deployed in the same physical data-center and interconnected via Gigabit Ethernet. In all the experiments performed on FutureGrid, a single thread per node was

2. The GMU prototype is publicly available at the URL `http://www.cloudtm.eu`.

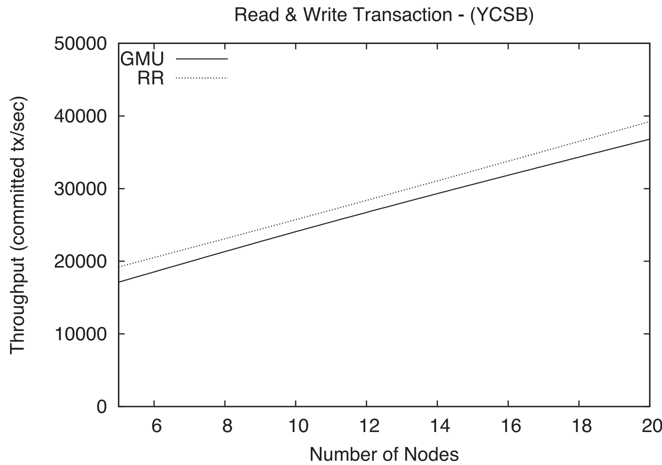
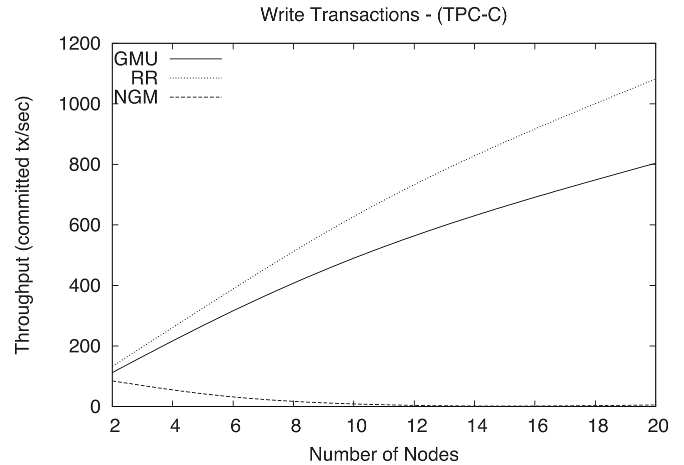


Fig. 3. YCSB benchmark (Cloud-TM).



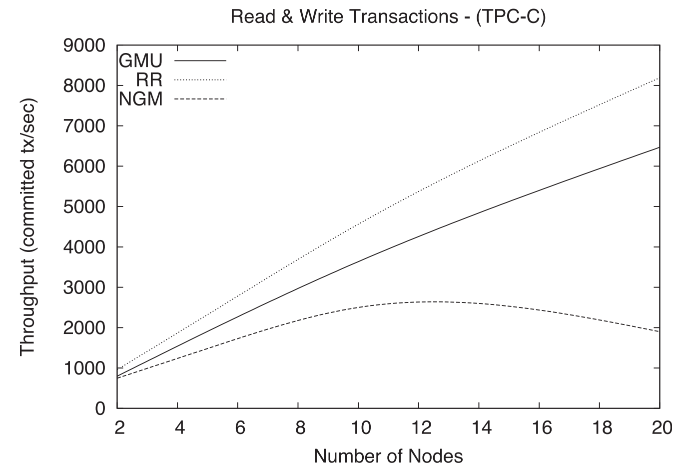


Fig. 4. TPC-C benchmark (Cloud-TM).

adopted to inject transactions (in closed loop), which guaranteed a high utilization of the machine's resources without overloading it. The second experimental platform, referred to as Cloud-TM, is a dedicated cluster of 20 homogeneous nodes, where each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 16 GB of RAM, running *Linux 2.6.32-33-server* and interconnected via a private Gigabit Ethernet. This platform is representative of small/medium private clouds or data-centers environments, with dedicated servers and a fairly large amount of available (computational and memory) resources per node. In order to maintain a similar ratio between threads and available cores with respect to the experiments on FutureGrid, in the experiments on Cloud-TM, four threads per node were used to inject transactions (in closed loop). In all the experiments we set the data replication degree to the value 2, which is a classical configuration enabling failure resiliency at reduced cost in terms of employed storage.

We initially show the results achieved on Cloud-TM by running Workload A [20] of YCSB, which is an update intensive workload (comprising 50 percent of update transactions) simulating a session store that records recent client actions. Fig. 3 reports the throughput (committed transactions per second) achievable by GMU and by the native RR partial replication protocol supported by Infinispan. The plot shows that the average reduction in throughput for GMU oscillates around 8 percent, and that GMU's throughput scales linearly at the same rate as RR, providing an evidence of the efficiency and scalability of the proposed solution.

Fig. 4 reports the results achieved on Cloud-TM by running the TPC-C benchmark configured with a read-dominated profile, composed at the 90 percent by read-only (Order-Status profiles) transactions and, for the remaining 10 percent, by update transactions (Payment and New-Order profiles) in equal parts. Similar read-dominated workloads are commonly found in realistic applications [43], [44] and benchmarks [45], [46]. In this case we also plot the performance of NGM. The results clearly show the detrimental effect on system scalability due to the high logical contention on the fully replicated global clock, which leads to the drastic decay of the throughput (in particular of update transactions, see plot on top of Fig. 4). As for GMU, it shows an almost linear scalability trend, although it suffers of a higher abort rate than RR: for 20 nodes, the abort rate is on the order of 15 percent for GMU, whereas it is around 8 percent for RR. Overall, in high contention scenarios, strong consistency semantics do pay a performance toll, which, in this specific configuration, corresponds to a throughput reduction ranging from 10 percent (at 4 nodes) up to 20 percent (at 20 nodes). On the other hand, one may argue that this is an unavoidable cost to pay in application contexts where correctness can be endangered by adopting non-serializable isolation levels. Note that TPC-C exactly belongs to this class of applications since its transactional profiles might generate data corruptions in presence of concurrency anomalies such as those that are possible using RR.

Fig. 5 reports the results achieved when running TPC-C on FutureGrid, which show that the maximum loss in performance of GMU versus RR (as observed at the larger scale of the platform, namely with 100 VMs) is bounded by 25 percent, and is in most of the cases (i.e., up to 80 VMs) less than 15 percent. In order to identify the causes underlying this performance gap, Fig. 6 reports the abort rate and the communication latency of remote read operations for GMU. The corresponding data for RR are not reported as they exhibit minimal variations.

As expected, also in this deployment, enforcing stronger consistency demands aborting a non-negligible fraction of transactions. RR, conversely, incurs an abort rate (for write transactions) of just 2 percent at 100 nodes, but does permit

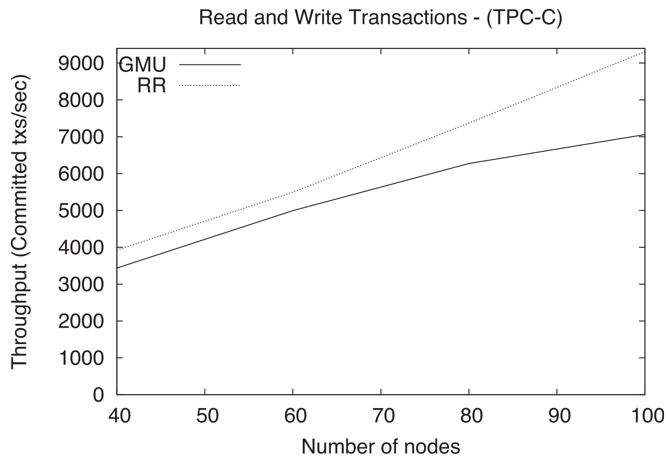


Fig. 5. TPC-C benchmark (FutureGrid).

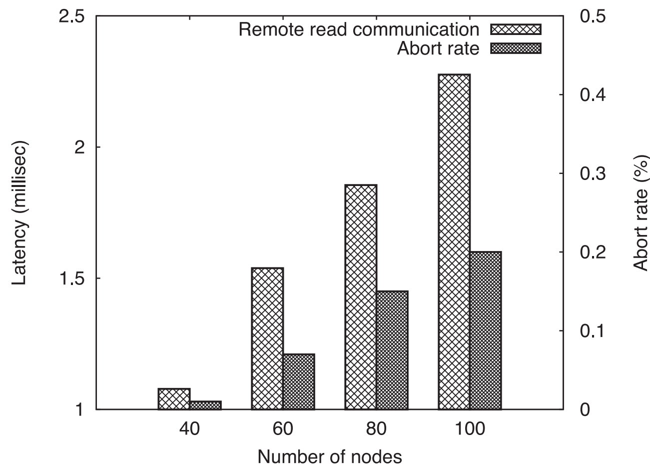


Fig. 6. Read operations latency and abort rate of write transactions (FutureGrid).

anomalies that could compromise application's correctness. Further, by analyzing the latency of remote read operations, we note that the use of vector clocks imposes overheads that grow almost linearly with the system's scale (whereas for RR the communication latency remains almost constant). This is due to the fact that the objects retrieved by read operations in TPC-C are typically very small (a few tens of bytes), which amplifies the relative overhead associated with the piggybacking of vector clocks. Also, the vector clock's implementation in the current GMU prototype is based on the JGroups messaging system (integrated with Infinispan), which imposes a serialization cost that grows linearly with the vector clock's size. We argue that this issue could be tackled by relying on, e.g., optimized vector clock representation schemes (see [37], [38]) and associated implementations, which are fully orthogonal to the actual GMU logic.

7 Conclusions

In this article we presented GMU (Genuine Multiversion Update-Serializable protocol), an innovative partial replication protocol for transactional systems. The core of GMU is a distributed multiversion concurrency control scheme, which relies on a novel vector clock based synchronization algorithm to track, in a totally decentralized (and consequently scalable) way, both data and causal dependency relations. In order to maximize scalability, GMU adopts a genuine partial replication mechanism, which ensures that transactions only contact replicas that maintain data that they accessed. Further, GMU never aborts read-only transactions and spares them from expensive distributed validation schemes. This makes GMU particularly efficient in presence of read-intensive workloads, as typical of a wide range of real-world applications. We evaluated GMU by integrating it into Inifinispan, a mainstream in-memory key-value store, and performing an experimental study based on heterogeneous experimental platforms and industry standard benchmarks (namely TPC-C and YCSB). Our results show that GMU achieves almost linear scalability and that it introduces reduced overheads with respect to solutions ensuring non-serializable semantics in a wide range of workloads. Supported by these experimental results, we argue that GMU hits a sweet spot in the trade-off between consistency and performance. In fact, the consistency semantics guaranteed by GMU, namely Extended Update Serializability, is sufficiently strong to ensure the correctness of complex OLTP workloads (such as TPC-C), but also weak enough to allow for efficient and scalable implementations.

Acknowledgments

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under project UID/CEC/50021/2013. He started working on this article while he was a researcher at INESC-ID.

References

[1] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[2] M. K. Aguilera, et al., "Sinfonia: A new paradigm for building scalable distributed systems," *ACM Trans. Comput. Syst.*, vol. 27, no. 3, pp. 5:1–5:48, Nov. 2009.
[3] F. Marchioni and M. Surtani, *Infinispan Data Grid Platform*. Birmingham, U.K.: PACKT Publishing, 2012.
[4] G. DeCandia, et al., "Dynamo: Amazon's highly available key-value store," in *Proc. Symp. Oper. Syst. Principles*, 2007, pp. 205–220.
[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1995, pp. 1–10.
[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1996, pp. 173–182.
[7] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1018–1032, Jul./Aug. 2003.
[8] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, "On speculative replication of transactional systems," *J. Comput. Syst. Sci.*, vol. 80, no. 1, pp. 257–276, 2014.
[9] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surveys*, vol. 36, no. 4, pp. 372–421, Dec. 2004.
[10] N. Schiper, P. Sutra, and F. Pedone, "P-Store: Genuine partial replication in wide area networks," in *Proc. IEEE Symp. Rel. Distrib. Syst.*, 2010, pp. 214–224.
[11] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," PhD Thesis, Massachusetts Instit. of Technol., Cambridge, MA, USA, Tech. Rep. AAI0800775, 1999.

[12] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Parallel Distrib. Algorithms Conf.*, 1988, pp. 215–226.
[13] R. C. Hansdah and L. M. Patnaik, "Update serializability in locking," in *Proc. Int. Conf. Database Theory*, 1986, pp. 171–185.
[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[15] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha, "Unlocking concurrency," *ACM Queue*, vol. 4, no. 10, pp. 24–33, 2007.
[16] H. Garcia-Molina and G. Wiederhold, "Read-only transactions in a distributed database," *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 209–234, 1982.
[17] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation," in *Proc. Pacific Rim Symp. Dependable Comput.*, 2007, pp. 290–297.
[18] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí, "SIPRe: A partial database replication protocol with SI replicas," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 2181–2185.
[19] TPC Council, "TPC-C Benchmark, Standard Specification, Revision 5.11," Feb. 2010, http://www.tpc.org
[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[21] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *Proc. IEEE 32nd Int. Conf. Distrib. Comput. Syst.*, 2012, pp. 455–465.
[22] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols," *ACM Trans. Database Syst.*, vol. 25, no. 3, pp. 333–379, 2000.
[23] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 419–430.
[24] S. Peluso, R. Palmieri, F. Quaglia, and B. Ravindran, "On the viability of speculative transactional replication in database systems: A case study with PostgreSQL," in *Proc. 12th IEEE Int. Symp. Netow. Comput. Appl.*, 2013, pp. 143–148.
[25] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 4, pp. 551–566, Apr. 2005.
[26] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.
[27] R. Palmieri, F. Quaglia, and P. Romano, "OSARE: Opportunistic speculation in actively REplicated transactional systems," in *Proc. Symp. Rel. Distrib. Syst.*, 2011, pp. 59–64.
[28] N. Carvalho, P. Romano, and L. Rodrigues, "SCert: Speculative certification in replicated software transactional memories," in *Proc. Int. Conf. Syst. Storage*, 2011, pp. 10:1–10:13.
[29] M. S. Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: Scalable and strong consistency for Geo-replicated transactional systems," in *Proc. Int. Symp. Rel. Distrib. Syst.*, 2013, pp. 163–172.
[30] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. Symp. Oper. Syst. Principles*, 2011, pp. 385–400.
[31] J. C. Corbett, et al., "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 1–22, 2013.
[32] A. Chan and R. Gray, "Implementing distributed read-only transactions," *IEEE Trans. Softw. Eng.*, vol. 11, no. 2, pp. 205–212, Feb. 1985.
[33] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized STM algorithm," in *Proc. Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.
[34] B. Charron-Bost and A. Schiper, "Uniform consensus is harder than consensus," *J. Algorithms*, vol. 51, no. 1, pp. 15–37, Apr. 2004.
[35] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Comput. Surveys*, vol. 33, no. 4, pp. 427–469, Dec. 2001.
[36] A. Bartoli and O. Babaoglu, "Selecting a "Primary partition" in partitionable asynchronous distributed systems," in *Proc. 16th Symp. Rel. Distrib. Syst.*, 1997, pp. 138–145.
[37] T. Landes, "Dynamic vector clocks for consistent ordering of events in dynamic distributed applications," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2006, pp. 31–37.
[38] X. Wang, J. Mayo, W. Gao, and J. Slusser, "An efficient implementation of vector clocks in dynamic systems," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2006, pp. 593–599.
[39] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[40] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *Proc. VLDB Endow.*, vol. 4, no. 8, pp. 494–505, 2011.
[41] R. van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 164–206, 2003.
[42] B. Kemme and G. Alonso, "Don'T be lazy, be consistent: Postgres-R, A new way to implement database replication," in *Proc. Int. Conf. Very Large Data Bases*, 2000, pp. 134–143.
[43] J. Cachopo, "Development of rich domain models with atomic actions," Ph.D. dissertation, Technical Univ. of Lisbon, Lisboa, Portugal, 2007.
[44] S. M. Fernandes and J. A. Cachopo, "Strict serializability is harmless: A new architecture for enterprise applications," in *Proc. Int. Conf. Object Oriented Programm. Syst. Lang. Appl. (Companion)*, 2011, pp. 257–276.
[45] Transaction Processing Performance Council, *TPC Benchmark™ W, Standard Specification, Version 1.8*, 2002.
[46] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A benchmark for software transactional memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 315–324, 2007.

Sebastiano Peluso received the MS degree in computer engineering in 2010 from the Sapienza University of Rome and the PhD degree in computer engineering from the Sapienza University of Rome and the Instituto Superior Técnico, Universidade de Lisboa in 2014. He is a postdoctoral research associate at the Virginia Polytechnic Institute and State University. His research interests lie in the area of distributed systems and parallel programming, with focus on scalability and fault tolerance of transactional systems.

Pedro Ruivo received the MS degree in information systems and computer engineering from the Instituto Superior Técnico, Universidade de Lisboa, majoring in distributed systems. He is a remote contractor for Red Hat on Infinispan project. His main interests are distributed key-value store and transactional memory.

Paolo Romano is a senior researcher at INESC-ID since 2008 and an associate professor at the Instituto Superior Técnico, Universidade de Lisboa. His research interests include distributed computing, dependability, autonomic systems, performance modelling, and cloud computing. He has published more than 100 papers and serves regularly as a program committee member and reviewer for international conferences and journals in the above areas.

Francesco Quaglia received the MS degree in electronic engineering in 1995 and the PhD degree in computer engineering in 1999, both from the Sapienza University of Rome, where he currently works as an associate professor. His research interests include parallel and distributed computing systems and applications, high-performance computing, and fault tolerance. In these areas, he has authored (or coauthored) more than 150 technical articles.

Luís Rodrigues is a professor at the Instituto Superior Técnico, Universidade de Lisboa, and a researcher at INESC-ID where he now serves in the board of directors. His research interests lie in the area of reliable distributed systems. He is a coauthor of more than 150 papers and three textbooks on these topics.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.