Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

P-Store: Genuine Partial Replication in Wide Area Networks

Nicolas Schiper[1], Pierre Sutra[2], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

[2] Université Paris VI and INRIA Rocquencourt, avenue du président Kennedy 104, 75016 Paris, France

Abstract

Partial replication is a way to increase the scalability of replicated systems since updates only need to be applied to a subset of the system's sites, thus allowing replicas to handle independent parts of the workload in parallel. In this paper, we propose P-Store, a partial database replication protocol for wide area networks. In P-Store, each transaction T optimistically executes on one or more sites and is then certified to guarantee serializability of the execution. The certification protocol is *genuine*, it only involves sites that replicate data items read or written by T, and incorporates a mechanism to minimize a convoy effect. P-Store makes a thrifty use of an atomic multicast service to guarantee correctness: no messages need to be multicast during T's execution and a single message is multicast to certify T. This is in contrast to previously proposed solutions that either require multiple atomic multicast messages to execute T, are non-genuine, or do not allow transactions to execute on multiple sites. Experimental evaluations reveal that the convoy effect plays an important role even when one percent of the transactions are *global*, that is, they involve multiple sites. We also compare the scalability of our approach to a fully replicated solution when the proportion of global transactions and the number of sites vary.

Report Info

Published
April 2010

Number
USI-INF-TR-2010-3

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Partial replication is a way to improve the scalability of replicated systems by allowing sites to store a subset of the application data and split the load among replicas, to maximize throughput. As a consequence, data is replicated close to clients, to favor locality, and storage resources are used sparingly.

In this paper, we present P-Store, a scalable distributed key-value store that supports partial replication and transparent transactional access. P-Store assumes a wide area network environment where sites are clustered in *groups* (e.g., data centers) and seeks to minimize costly and slow inter-group communication.

The solution we propose is *flexible* and *scalable* in a precise sense as we explain next. Data items may be replicated anywhere and any number of times provided that sites of a given group replicate the same set of data items. Transaction execution does not require data items to be accessed from the same site, allowing more flexibility when partitioning data. Read requests are executed optimistically with no inter-site synchronization; transactions are then certified to guarantee serializability of the execution. To improve scalability, the certification protocols we present ensure *genuine partial replication*:

- For any submitted transaction T, only database sites that replicate data items read or written by T exchange messages to certify T.

In P-Store, correctness relies on the use of an atomic multicast service to order transactions that operate on the same data items. We make an economical use of this service: to execute and certify each transaction, a single message is atomically multicast.

This is in contrast to previously proposed solutions that either atomically multicast multiple messages to handle each transaction [1, 2], are non-genuine [3, 4], or do not allow transactions to execute on multiple sites, thereby forcing at least one replica to hold the entire database if a single transaction operates on the entire data [3]. To the best of our knowledge, this is the first genuine partial database replication protocol that allows transactions to execute on multiple sites while using a single atomic multicast message per transaction.

The first certification protocol we propose is simple but vulnerable to a convoy effect that slows down transaction certification due to *global transactions*, i.e., transactions that involve multiple groups. To mitigate this undesired phenomenon, we propose a second protocol that doubles the throughput of the first protocol even when only 1% of transactions are global—this advantage grows when the percentage of global transactions increases.

We further study the performance of P-Store and compare its scalability to a fully replicated solution when the percentage of global transactions and the number of groups vary. P-Store provides a linear scale-out up to eight groups and when a fourth of the transactions are global. With this number of groups, P-Store allows to almost double the peak throughput of the fully replicated scheme and can process multiple thousands of update transactions when each data items is replicated three times and inter-group links have a delay of 50 milliseconds and 10 megabits per second of bandwidth. Preliminary experimental results suggest that partial replication is interesting in systems with four or more groups when global transactions access few groups.

The rest of the paper is structured as follows. Section 2 introduces our model and assumptions. Sections 3 and 4 respectively present P-Store and the two certification protocols; the current state-of-the-art is surveyed in Section 5. The implementation of P-Store is sketched in Section 6 and empirical results are reported in Section 7. Section 8 concludes the paper. We prove the correctness of P-Store in the appendix.

2 System Model and Definitions

2.1 Sites, Groups, and Links

We consider a system $\Pi = \{s_1, ..., s_n\}$ of sites, each equipped with a local database. Sites communicate through message passing and do not have access to a shared memory nor a global clock. We assume the benign crash-stop failure model: sites may fail by crashing, but do not behave maliciously. A site that never crashes is *correct*; otherwise it is *faulty*.

The system is asynchronous, i.e., messages may experience arbitrarily large (but finite) delays and there is no bound on relative site speeds. To circumvent the FLP impossibility result [5] and make atomic multicast implementable, we further assume that the system is augmented with unreliable failure detectors [6]. The exact failure detector needed depends on the atomic multicast algorithm. Hereafter, we assume an atomic multicast service, as defined in 2.4.

We define $\Gamma = \{g_1, ..., g_m\}$ as the set of site groups in the system. Groups are disjoint, non-empty, and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. For each site $s \in \Pi$, group(s) identifies the group s belongs to. A group g that contains at least one correct site is correct; otherwise g is faulty.

Communication links are quasi-reliable, i.e., they do not create, corrupt, nor duplicate messages, and for any two *correct* sites s and s', and any message m, if s sends m to s', then s' eventually receives m.

2.2 Database and Replication Model

A database $\mathscr{D}$ is a finite set of tuples (k, v, ts), or *data items*, where k is a key, v its value, and ts is its version number. Each site holds a partial copy of the database. For each site s_i, we denote by $Items(s_i) \subseteq \mathscr{D}$ the set of keys replicated at site s_i. Given a site s_i and a key k replicated at site s_i, $Version(k, s_i)$ returns the version of k stored at s_i. We suppose that, initially, for every key k and every site s_i, $Version(k, s_i) = 1$. We assume that sites in the same group replicate the same set of data items, that is, $\forall g \in \Gamma : \forall s, s' \in g : Items(s) = Items(s')$, and we allow data items to be replicated in more than one group. For every key k in the database, there exists a correct site s_i that replicates the value associated with k, i.e., $k \in Items(s_i)$.

A transaction is a sequence of read and write operations on data items followed by a commit or abort operation. A read on some key k by transaction T, denoted $r_T[k]$, returns the value associated with k as well as its corresponding version. A write performed by T is designated as $w_T[k, v, ts]$, where v is the value written on key k, and ts is its version. For simplicity, we hereafter represent a transaction T as a tuple (id, rs, ws, up) where id is the unique identifier of T, rs is the set of key-version pairs read by T, ws is the set of keys written by T, and up contains the updates of T. More precisely, up is a set of tuples (k, v), where v is the new value T associates to k; the set $T.ws$ equals $\{k : (k, v) \in T.up\}$, where we refer to element e of T's tuple as $T.e$.

For every transaction T, $Items(T)$ is the set of keys read or written by T. Transactions T and T' *read-write conflict* if one transaction reads a key k that the other transaction updates. We do not consider write-write conflicts since, in the certification protocols we propose, updates to each key k are ordered. Every transaction T is associated to a unique site: $Proxy(T)$, which submits T's read and write requests on behalf of a client. We denote $WReplicas(T)$ as the set of sites that replicate at least one data item written by T, and $Replicas(T)$ as the sites that replicate at least one data item read or written by T. Transaction T is *local* iff for any site s in $Replicas(T)$, $Items(T) \subseteq Items(s)$; otherwise, T is *global*.

2.3 Data Consistency Criteria

On each site, the local database ensures *order-preserving serializability*: the local execution of transactions has the same effect as a serial execution of these transactions in which the commit order is preserved [7]. This condition is typically met by relying on two-phase locking.

In this paper, we provide a partial replication protocol that ensures the transaction execution on multiple *partial* copies of the database is *equivalent* to some one-copy serial execution of the same set of transactions. More precisely, the devised protocol ensures *one-copy serializability* (1-SR) [8].

2.4 *Atomic Multicast*

We assume that our system is equipped with an atomic multicast service that allows messages to be disseminated to any subset of groups in Γ [1, 9]. For every message m, $m.dst$ denotes the groups to which m is multicast. A message m is multicast by invoking A-MCast(m) and delivered with A-Deliver(m). We define the relation $<$ on the set of messages sites A-Deliver as follows: $m < m'$ iff there exists a site that A-Delivers m before m'.

Atomic multicast satisfies the following properties: (i) *uniform integrity*: for any site s and any message m, s A-Delivers m at most once, and only if $s \in m.dst$ and m was previously A-MCast, (ii) *validity*: if a correct site s A-MCasts a message m, then eventually all correct sites $s' \in m.dst$ A-Deliver m, (iii) *uniform agreement*: if a site s A-Delivers a message m, then eventually all correct sites $s' \in m.dst$ A-Deliver m, (iv) *uniform prefix order*: for any two messages m and m' and any two sites s and s' such that $\{s, s'\} \subseteq m.dst \cap m'.dst$, if s A-Delivers m and s' A-Delivers m', then either s A-Delivers m' before m or s' A-Delivers m before m', (v) *uniform acyclic order*: the relation $<$ is acyclic.

To guarantee *genuine partial replication*, we require atomic multicast protocols to be *genuine* [10]: an algorithm $\mathscr{A}$ solving atomic multicast is genuine iff for any admissible run R of $\mathscr{A}$ and for any site s, in R, if s sends or receives a message, then some message m is A-MCast, and either s is the site that A-MCasts m or $s \in m.dst$.

3 The Lifetime of a Transaction in P-Store

We present the lifetime of transactions in our partially replicated system P-Store. We consider a transaction T and comment on the different states T can be in.

- *Executing:* Each read operation on key k is executed at some site that stores k; k and the data item version ts read are stored as a pair (k, ts) in $T.rs$. Reads are optimistic, that is, no synchronization between sites in $Replicas(T)$ occurs to guarantee the consistency of T's view of the database; later on, when T is in the Submitted state, a *certification protocol* checks that T read the correct data item versions. Every update of key k to some value v is buffered as a pair (k, v) in $T.up$.

  When $Proxy(T)$ requests a commit, T passes to the Committed state if T is read-only and local. Otherwise, if T is global or an update transaction, T is submitted to the certification protocol,

at which point T enters the Submitted state. The goal of the certification protocol is twofold. First, it propagates T's updates to *WReplicas*(T). Second, it ensures that the execution of transactions is one-copy serializable. To submit T, sites use one of the certification protocols presented in Section 4.

- *Submitted:* To ensure one-copy serializability, the certification protocol checks whether T observed an up-to-date view of the database despite optimistic reads and concurrent updates. If T passes the certification test, T enters the Committed state; otherwise, T passes to the Aborted state.

- *Committed/Aborted:* If T commits, all sites in *WReplicas*(T) apply its updates. Whatever T's outcome, *Proxy*(T) is notified.

P-Store, combined with any of the two certification protocols proposed in this paper, ensures *genuine partial replication* and one-copy serializability. Moreover, the following two liveness properties are also ensured:

- *non-trivial certification*: If there is a time after which no two read-write conflicting transactions are submitted, then eventually transactions are not aborted by certification

- *termination*: For every submitted transaction T, if *Proxy*(T) is correct, then all correct sites in *WReplicas*(T) either commit or abort T.

4 Certifying Transactions

In this section, we present two certification protocols. The first one is simple but suffers from a *convoy effect*, that is, transaction certification may be delayed by transactions currently being certified. The second protocol seeks to minimize this undesired phenomenon.

4.1 A Genuine Protocol

The algorithm $\mathscr{A}_{ge}$ we present next relies on atomic multicast to certify transactions. We first present an overview of the algorithm and then present $\mathscr{A}_{ge}$ in detail.

Algorithm Overview When a transaction T is submitted for certification, Algorithm $\mathscr{A}_{ge}$ atomically multicasts T to all groups storing keys read or updated by T. Upon A-Delivering T, each site s_r that replicates data items read by T checks whether the values read are still up-to-date. To do so, s_r compares the version of the data items read by T against the versions currently stored in the database. If they are the same, T passes certification at s_r, otherwise T fails certification at s_r.

In a partially replicated context, s_r may only store a subset of the keys read by T, in which case s_r does not have enough information to decide on T's outcome. Hence, to satisfy *non-trivial certification*, we introduce a voting phase where sites replicating data items read by T send the result of their certification test to each site s_w in *WReplicas*(T).[1] Site s_w can safely decide on T's outcome when s_w received votes from a *voting quorum* for T. Intuitively, a voting quorum VQ for T is a set of sites such that for each data item read by T, there is at least one site in VQ replicating this item. More formally, a quorum of sites is a voting quorum for T if it belongs to $VQ(T)$, defined as follows:

$$VQ(T) = \{VQ | VQ \subseteq \Pi \wedge T.rs \subseteq \bigcup_{s_r \in VQ} Items(s_r)\} \quad (1)$$

Transaction T can safely commit when every site in a voting quorum for T voted *yes*. If a site in the quorum votes *no*, it means that T read an old value and should be aborted to ensure serializability of the execution.

Figure 1 illustrates the execution of a global transaction T that reads data items from groups g_1 and g_2. After all read requests have been executed, T is submitted to $\mathscr{A}_{ge}$ for certification.

Algorithm in Detail Algorithm $\mathscr{A}_{ge}$ (see next page) is composed of three concurrent tasks. Each line of the algorithm is executed atomically. The algorithm uses a global variable named *Votes* that stores the votes received, i.e., the results of the certification tests.

When a transaction T is submitted, *Proxy*(T) atomically multicasts T to *Replicas*(T) (line 10). Upon A-delivering T, each site s that stores data items read by T certifies T (line 8). If T is local, s knows T's outcome at this point and, in case T commits, s applies T's updates to the database (lines 16-17). Otherwise, T aborts (line 18). If T is global, the result of the certification test is stored locally at s and sent to every site s_w in *WReplicas*(T), except to members of s's group (lines 20-22). Each site s_w waits until it receives votes from a voting quorum for T at which point s_w can safely decide on T's outcome (lines 23-24), and T is handled similarly as local transactions (lines 25-28). The outcome of T is then sent to *Proxy*(T) (line 29).

4.2 Minimizing the Convoy Effect

The convoy effect occurs when the certification of a transaction T_1 is delayed by another global transaction T_2 although T_1 is ready to be certified. In the certification protocol $\mathscr{A}_{ge}$, this phenomenon may happen

[1] A similar voting phase appears in [11]. In contrast to $\mathscr{A}_{ge}$, the protocol in [11] is non-genuine and relies on a total order to certify transactions.

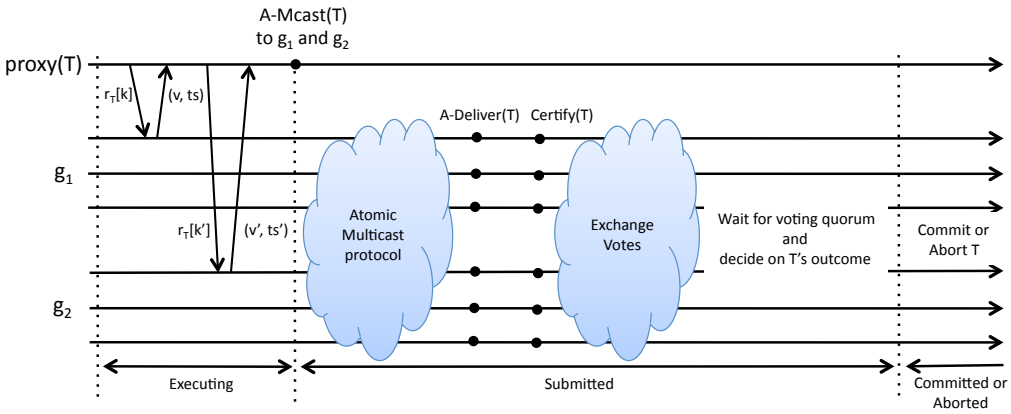


Figure 1: The execution and certification of a global transaction T involving groups g_1 and g_2 with Algorithm $\mathcal{A}_{ge}$.

Algorithm $\mathcal{A}_{ge}$
A Genuine Certification Protocol - Code of site s

```
1:  Initialization
2:    Votes ← ∅

3:  function ApplyUpdates(T)
4:    foreach ∀(k, v) ∈ T.up : k ∈ Items(s) do
5:      let ts be Version(k, s)
6:      w_T[k, v, ts+1]                          {write to the database}

7:  function Certify(T)
8:    return ∀(k, ts) ∈ T.rs s.t. k ∈ Items(s) : ts = Version(k, s)

9:  To submit transaction T                      {Task 1}
10:   A-MCast(T) to Replicas(T)                  {Executing → Submitted}

11: When receive(Vote, T.id, vote) from s'       {Task 2}
12:   Votes ← Votes ∪ (T.id, s', vote)

13: When A-Deliver(T)                            {Task 3}
14:   if T is local then
15:     if Certify(T) then
16:       ApplyUpdates(T)
17:       commit T                               {Submitted → Committed}
18:     else abort T                             {Submitted → Aborted}
19:   else
20:     if ∃(k, -) ∈ T.rs : k ∈ Items(s) then
21:       Votes ← Votes ∪ (T.id, s, Certify(T))
22:       send(Vote, T.id, Certify(T)) to all s' in WReplicas(T) s.t.
                                                 s' ∉ group(s)
23:     if s ∈ WReplicas(T) then
24:       wait until ∃VQ ∈ VQ(T) :
                  ∀s' ∈ VQ : (T.id, s', -) ∈ Votes
25:       if ∀s' ∈ VQ : (T.id, s', yes) ∈ Votes then
26:         ApplyUpdates(T)
27:         commit T                             {Submitted → Committed}
28:       else abort T                           {Submitted → Aborted}
29:   if s ∈ WReplicas(T) then send T's outcome to Proxy(T)
```

as follows: T_1 was A-Delivered but it must wait until T_2's votes are received to be certified. As the frequency of submitted global transactions increases, this phenomenon deteriorates the performance of $\mathcal{A}_{ge}$: an ever growing chain of transactions waiting to be certified is formed since only one global transaction can be certified per inter-group delay. We address this problem in Algorithm $\mathcal{A}^*_{ge}$. We first give an overview of the algorithm and then present $\mathcal{A}^*_{ge}$ in detail.

Algorithm Overview To reduce the convoy effect, we seek to certify transactions in parallel as much as possible. In the scenario described above, this allows T_1 to be certified while T_2's votes are exchanged.

Obviously, not all transactions can be certified concurrently. Consider two read-write conflicting transactions T_0 and T_1 such that T_i reads key k_i and writes key k_{1-i} with $i \in [0,1]$. Further, suppose that keys k_0 and k_1 are replicated in different groups and thus a vote phase is needed to certify these transactions. If T_0 and T_1 were certified in parallel, a site s_1 could certify T_1 followed by T_2 while another site s_2 could certify T_2 before T_1. In this scenario, s_1 would vote "commit" for T_0 but "abort" for T_1, the inverse of what s_2 would do.

We observe that when transactions do not read-write conflict, their certification order does not matter since they do not *affect* each other. These transactions can thus be certified in parallel.

Nevertheless, the updates of such transactions must be applied in the order defined by atomic multicast. In short, this is because local read-only transactions are not certified. To illustrate this, consider the following execution that violates 1-SR. Suppose that two transactions T_1 and T_2 update keys k_1 and k_2, respectively, and on site s_1, T_1 commits before T_2, while on site s_2, T_2 commits before T_1. Consider in addition that on s_1 a local read-only transaction T_3 reads keys k_1 and k_2 before T_2 commits but after T_1 does, and that s_2 executes a local read-only transaction T_4 that reads the same data items as T_3 before T_1 commits but after T_2 does. This execution is not 1-SR: in a one-copy serial execution, T_3 must be placed before T_2 but after T_1. However, T_4 should be placed before T_1 but after T_2, which is impossible.

Algorithm in Detail Algorithm $\mathscr{A}^*_{ge}$ is composed of four concurrent tasks. Each line of the algorithm is executed atomically. The algorithm uses two global variables: *Votes* stores the results of the certification tests, as in $\mathscr{A}_{ge}$, and *CertifyQ* is a FIFO queue of transactions that are being certified.

To submit a transaction T, *Proxy*(T) atomically multicasts T to *Replicas*(T) (line 11). When T is A-Delivered at a site s and T does not read-write conflict with any transaction currently being certified, s stores and sends the result of T's certification test if s replicates data items read by T and T is global (lines 14-17). If s is concerned by T's outcome, s adds T to the tail of *CertifyQ* (line 18).

Site s then waits until there exists a transaction T in *CertifyQ* whose outcome is known, i.e., Outcome(T) $\neq \perp$. Recall that T's outcome is known after s certifies T if T is local (lines 4-5); if T is global, its outcome is determined by votes from a voting quorum for T (lines 6-7). If T can commit, s waits until T is at the head of the certification queue before applying T's updates and committing T (lines 19-22). This ensures that transaction updates are applied in the order defined by atomic multicast. If T failed the certification test, T can be aborted regardless of T's position in *CertifyQ* (lines 19). Transaction T is then removed from *CertifyQ* and its outcome sent to *Proxy*(T) (lines 25-26).

4.3 Why Does it Work?

We briefly argue why P-Store ensures one-copy serializability, and refer the reader to the appendix for a complete proof. We consider only certification protocol $\mathscr{A}^*_{ge}$ since $\mathscr{A}_{ge}$ is a special case of $\mathscr{A}^*_{ge}$.

Let H be a replicated data history consisting of committed transactions only. History H is 1-SR iff H is *view-equivalent* to some one-copy serial history $1H$, where H and $1H$ are view-equivalent iff the following holds [8]:

1. H and $1H$ are defined over the same set of transactions,
2. H and $1H$ have the same *reads-x-from* relationships on data items: $\forall T_i, T_j \in H$ (and hence, $T_i, T_j \in 1H$): T_j *read-x-from* T_i in H iff T_j *reads-x-from* T_i in $1H$, and,
3. For each final write $w_T[k, v, ts]$ in $1H$, $w_T[k_a, v, ts]$ is also a final write in H for some copy k_a of key k.

We show how to construct a one-copy serial history $1H$ that is view-equivalent to H. History $1H$ consists of the same committed transactions as H, write operations follow the order defined by atomic multicast, and operations from different transactions do not interleave. In certification protocol $\mathscr{A}^*_{ge}$, a transaction T passes the certification test iff the data items read by T are still up-to-date. Hence, if T commits then no transactions updated data items read by T in the meantime. Consequently, T reads a key k written by some transaction T' in H iff T reads key k from T' in $1H$. The fact that $\mathscr{A}^*_{ge}$ certifies non-conflicting transactions in parallel does not matter since certifying these transactions sequentially would produce the same result. Finally, since writes to every key k are ordered by atomic multicast, it follows directly that if $w_T[k, v, ts]$ is a final write to k in $1H$, then $w_T[k_a, v, ts]$ is also a final write to k in H.

Algorithm $\mathscr{A}^*_{ge}$
Minimizing the Convoy Effect - Code of site s

```
1:  Initialization
2:    Votes ← ∅, CertifyQ ← ϵ
      { Functions ApplyUpdates and Certify are as in A_ge }
3:  Function Outcome(T)
4:    if T is local then
5:      return Certify(T)
6:    else if ∃VQ ∈ VQ(T) : ∀s' ∈ VQ : ∃(T.id, s', −) ∈ Votes then
7:      return ∀s' ∈ VQ : (T.id, s', yes) ∈ Votes
8:    else
9:      return ⊥
10: To submit transaction T                                   {Task 1}
11:   A-MCast(T) to Replicas(T)                  {Executing → Submitted}
12: When receive(Vote, T.id, vote) from s'                    {Task 2}
13:   Votes ← Votes ∪ (T.id, s', vote)
14: When A-Deliver(T) and ∄T' ∈ CertifyQ :
          T' read-write conflicts with T                      {Task 3}
15:   if T is global and ∃(k, -) ∈ T.rs : k ∈ Items(s) then
16:     Votes ← Votes ∪ (T.id, s, Certify(T))
17:     send(Vote, T.id, Certify(T)) to all s' in WReplicas(T) s.t.
                                                  s' ∉ group(s)
18:   if s ∈ WReplicas(T) then add T to the tail of CertifyQ
19: When ∃T ∈ CertifyQ : Outcome(T) ≠ ⊥ and
        (T = head(CertifyQ) or Outcome(T) = no)               {Task 4}
20:   if Outcome(T) = yes then
21:     ApplyUpdates(T)
22:     commit(T)                                {Submitted → Committed}
23:   else if Outcome(T) = no then
24:     abort(T)                                  {Submitted → Aborted}
25:   CertifyQ ← CertifyQ \ {T}
26:   send T's outcome to Proxy(T)
```

5 Related Work

Numerous protocols for full database replication have been proposed [13, 14, 15, 16], some of which have been evaluated in wide area networks [17]. Fewer protocols for partial replication exist. These protocols can be grouped in two categories: those that are optimized for local area networks [18, 19, 11], and those that are topology-oblivious. In the following, we review pro-

Algorithm	Genuine?	Execution on multiple sites?	Execution latency	Certification latency	messages (execution + certification)	Consistency criterion
[3]	no	no	-	2Δ	$O(n^2)$	GSI
[4]	no	yes	$(r_r + w_r + 1) \times 2\Delta$	2Δ	$O(n^2)$	GSI
[12]	yes	yes	$r_r \times 2\Delta \mid r_r \times 3\Delta$	4Δ	$O(k^2d^2) \mid O((r_r + k^2) \times d^2)$	1-SR
[2]	yes	yes	$r_r \times 2\Delta$	$2\Delta \mid 4\Delta$	$O(k^2d^2) \mid O((r_r + w_r + k^2) \times d^2)$	1-SR
$\mathscr{A}_{ge}$ & $\mathscr{A}^*_{ge}$	yes	yes	$r_r \times 2\Delta$	3Δ	$O(k^2d^2)$	1-SR

Table 1: Comparison of the database replication protocols (r_r and w_r are the number of remote reads and writes respectively, n is the number of sites in the system, d is the number of sites per group, and k is the number of groups addressed by T).

tocols from the second category that either provide a generalized form of snapshot isolation (GSI) [3, 4] or one-copy serializability (1-SR) [12, 2].

With GSI, transactions read data from a possibly old committed snapshot of the database and execute without interfering with each other. A transaction T can only successfully commit if no other transaction T' updated the same data items and committed after T started (*first-committer-wins* rule). This consistency criterion never blocks nor aborts read-only transactions and update transactions are never blocked nor aborted due to read-only transactions.

To the best of our knowledge, none of the protocols that ensure GSI guarantee *genuine partial replication*. In fact, to certify a transaction T, the protocols in [3, 4] require to atomically multicast T to all sites in the system. Moreover, in [3], each transaction operation must be executed at the same site. Thus, if a single transaction operates on the entire data, at least one replica must store the whole database. The protocol in [4] does not have this drawback, however, for each transaction T executed, an additional *snapshot* message must be atomically multicast to guarantee that T observes a consistent view of the database.

In [12] the authors propose a database replication protocol based on atomic multicast that ensures 1-SR. Every read operation on data item x is multicast to the group replicating x; writes are multicast along with the commit request. The delivered operations are executed on the replicas using strict two-phase locking and results are sent back to the client. A final atomic commit protocol ensures transaction atomicity. In the atomic commit protocol, every group replicating a data item read or written by a transaction T sends its vote to a *coordinator* group, which collects the votes and sends the result back to all participating groups.

In [2], a protocol that allows transactions to be executed on multiple sites is presented. To certify a transaction T, T is reliably multicast to *Replicas*(T)[2], and each operation of T on some data item x is atomically multicast to the replicas of x. Sites then build the graph G of transactions that *precede* T in the execution by exchanging their partial view of G. One-copy serializability is ensured by checking that G is acyclic. These last two operations, namely building G and checking that G is acyclic, can be expensive.

Two algorithms based on atomic multicast that ensure *genuine partial replication* are presented in this paper. They require a single atomic multicast per transaction and allow transactions to execute at multiple sites. This allows to effectively partition the database even if transactions access the database in its entirety. Algorithm $\mathscr{A}_{ge}$ may suffer from the convoy effect since it certifies transactions sequentially. The second algorithm $\mathscr{A}^*_{ge}$ alleviates this undesired phenomenon by allowing non-conflicting transactions to be certified in parallel.

Table 1 compares the properties and cost of the reviewed protocols. Columns two and three respectively indicate whether the protocols ensure *genuine partial replication* and whether transactions can be executed at multiple sites. The subsequent three columns present the cost of the protocols, namely the inter-group latency to execute and certify a transaction T, and the total number of inter-group messages exchanged to execute and certify T. To compute these costs, we consider that T is global and is executed from within some group g. Transaction T issues r_r *remote* reads and w_r *remote* writes. These operations access data items stored outside of g and thus require inter-group communication. Further, we assume that groups are correct, neither failures nor failure suspicions occur, inter-group messages have a delay of Δ, and intra-group message delays are assumed to be negligible. In Table 1, costs are computed by using the atomic multicast algorithm in [20]. This protocol has a latency of Δ and 2Δ for messages addressed to one and multiple groups respectively, and sends $O(x^2)$ inter-group messages to deliver multicast messages, where x is the number of sites to which the transaction is multicast. In columns four, five, and six, we report the costs of the algorithms when data items are replicated in one and two groups respectively. When these costs are identical, we report a single value.

6 The Implementation of P-Store

We implemented P-Store in Java on top of BerkeleyDB (BDB). To execute a transaction T, a client sends read

[2]Reliable multicast ensures all properties of atomic multicast except uniform prefix and acyclic order.

and write requests to *Proxy*(T), one of the sites in *Replicas*(T). Write requests are buffered by *Proxy*(T) and each read of key k is executed at some site that stores k inside a BDB transaction. When T is ready to commit, *Proxy*(T) submits T along with T's set of key-version pairs read and T's updates using one of the certification protocols presented in this paper. If T passes the certification test, a BDB transaction applies T's updates to the database in the order defined by atomic multicast. Otherwise, P-Store re-executes T and resubmits T to the certification protocol.

Certification protocols $\mathscr{A}_{ge}$ and $\mathscr{A}^*_{ge}$ are implemented on top of an atomic multicast library optimized for wide area networks [21]. When a transaction T is A-Delivered at some site s, s assigns a *certifier thread* to T. This thread executes the certification test for T and applies T's update to the database as soon as T does not read-write conflict with transactions currently being certified. Certifiers are part of a thread pool whose size is configurable. When P-Store uses $\mathscr{A}_{ge}$ to certify transactions, the certifier pool contains a single thread.

7 Performance Evaluation

In this section, we present an experimental evaluation of the performance of P-Store with the certification protocols $\mathscr{A}_{ge}$ and $\mathscr{A}^*_{ge}$. We start by presenting the system settings and the benchmark used to assess the protocols. We then evaluate the impact of the convoy effect on $\mathscr{A}_{ge}$ and asses the scalability of P-Store when partial and full replication are assumed.

7.1 Experimental Settings

The system The experiments were conducted in a cluster of 24 nodes connected with a gigabit switch. Each node is equipped with two dual-core AMD Opteron 2 Ghz, 4GB of RAM, and runs Linux 2.6. In all experiments, each group consisted of 3 nodes, and the number of groups varied from 2 to 8. We assumed that groups were correct and used an atomic multicast service optimized for this assumption. To provide higher degrees of resilience, it would have sufficed to replace the atomic multicast service with an implementation that tolerates group crashes [9]. The bandwidth and message delay of our local network, measured using netperf and ping, were about 940 Mbps and 0.05 ms respectively. To emulate inter-group delays with higher latency and lower bandwidth, we used the Linux traffic shaping tools. We considered a network in which the message delay between any two groups follows a normal distribution with a mean of 50 ms and a standard deviation of 5 ms, and each group is connected to the other groups via a 1.25MBps (10 Mbps) full-duplex link.

BerkeleyDB was configured with asynchronous disk writes and logging in memory. Moreover, on each site s, the cache was big enough to hold the entire portion of the database replicated at s. This corresponds to a setting were data durability is ensured through replication.

The benchmark We measured the performance of the protocols using a modified version of the industry standard TPC-B benchmark [22]. TPC-B consists of update transactions only, and defines one transaction type that deposits (or withdraws) money from an account. In our implementation of TPC-B, each transaction reads and updates three data items: the account, the teller in charge of the transaction, and the branch in which the account resides. Accounts and tellers are associated with a unique branch.

We horizontally partitioned the branch table such that each group is responsible for an equal share of the data. Accounts and tellers of a branch b were replicated in the same group as b, and members of a given group replicated the same set of data items. Before partitioning, the database consisted of 3'600 branches, 36'000 tellers, and 360'000 accounts.

In TPC-B, about 15% of transactions involve two groups, that is, the teller in charge of the transaction is replicated in a different group than the group in which the branch and account are stored. The remaining 85% of transactions access data in the same group. To assess the scalability of our protocols, we parameterized the benchmark to control the proportion p of global transactions. In the experiments, we report measurements when p varies from 0% to 50%.

Each node of the system contained an equal number of clients that executed the benchmark in a closed loop: each client executed a transaction T and waited until it was notified of T's outcome before executing the next one. Each client c was placed in the same group as the teller in charge of handling c's transactions, and inside each group g the load generated by g's clients was shared among g's replicas. For all the experiments, we report the average transaction execution time (in milliseconds) as a function of the throughput, i.e., the number of transactions committed per second. We computed 95% confidence intervals for the transaction execution times but we do not report them here as they were always smaller than 5% of the average execution time. The throughput was increased by adding an equal number of clients to each node of the system and, on average, four hundred thousand transactions were executed per experiment. Unless we explicitly state otherwise, the executions were cpu-bound at peak loads.

7.2 Assessing the Convoy Effect

We explore the influence of the number of certifier threads on the performance of P-Store. We consider a system with four groups, one percent of global transactions, and vary the number of certifier threads from one to one hundred fifty. With any pool size bigger than one, the certification protocol used is $\mathscr{A}^*_{ge}$, otherwise it is $\mathscr{A}_{ge}$.

In Figure 2, we observe that the convoy effect affects both latency and throughput despite a low percentage of global transactions. With one certifier thread, as soon as sites certify a global transaction, no other transactions can be certified for the duration of the vote exchange, that is, one inter-group delay. This creates long chains of transactions waiting to be certified and limits throughput. Adding extra certifiers quickly improves the performance of P-Store which reaches its peak bandwidth with one hundred fifty threads. With this pool size, the peak throughput reached by P-Store more than doubled compared to when P-Store uses a single certifier. We observed that the difference between these two certification protocols grows when the percentage of global transactions increases.

In the following experiments, we configured $\mathscr{A}^*_{ge}$ to use one hundred certifiers since the performance gained by adding an extra fifty certifiers is small.

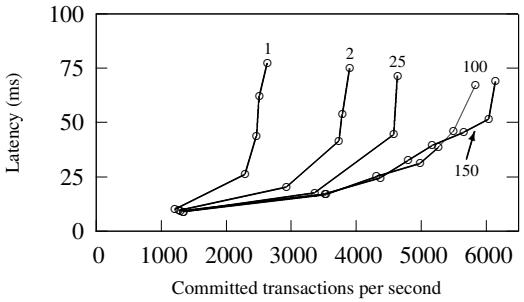


Figure 2: The influence of the number of certifier threads in a system with four groups and 1% of global transactions.

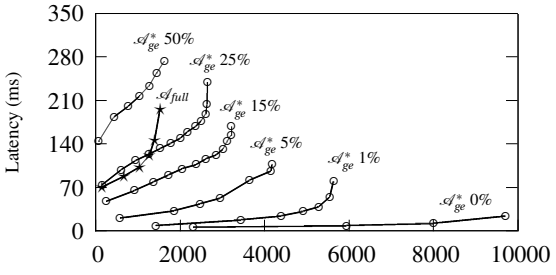
7.3 Full versus Partial Replication

In the following, we assess the scalability of P-Store and compare certification protocol $\mathscr{A}^*_{ge}$ against a certification protocol denoted as $\mathscr{A}_{full}$ that assumes full replication, i.e., all sites store the entire database. To do so, we evaluate the performance of $\mathscr{A}^*_{ge}$ and $\mathscr{A}_{full}$ when the number of groups and percentage of global transactions vary.

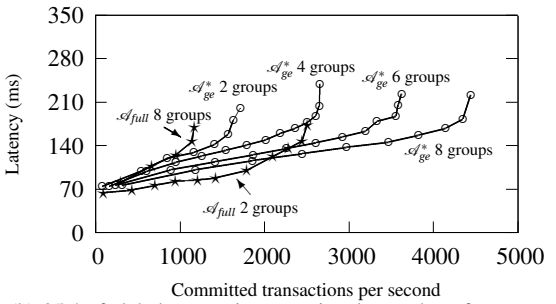
Protocol $\mathscr{A}_{full}$ is based on $\mathscr{A}^*_{ge}$: non-conflicting transactions are certified in parallel and updates are applied in the order defined by atomic multicast. When an update transaction T is submitted to $\mathscr{A}_{full}$, T is atomically multicast to all sites to be certified and to propagate its updates. Since full replication is assumed, all transactions are local and do not require a vote phase. The certifier pool of $\mathscr{A}_{full}$ contains one hundred threads.

Varying the percentage of global transactions In Figure 3(a), we report the latency of P-Store as a function of throughput when using certification protocols $\mathscr{A}^*_{ge}$ and $\mathscr{A}_{full}$. We consider from 0% to 50% of global transactions and a system with four groups. $\mathscr{A}^*_{ge}$ presents a similar latency as $\mathscr{A}_{full}$ with 25% of global transactions but supports higher loads. Any percentage of global transactions higher than 25% makes full replication more attractive than partial replication. This is explained by the extra cost paid by $\mathscr{A}^*_{ge}$ to fetch remote data items and execute vote phases to handle global transactions.

With lower percentages of global transactions, $\mathscr{A}^*_{ge}$ provides lower latencies and improves the peak throughput of $\mathscr{A}_{full}$ by a factor of 2.1, 2.7, 3.7, and 6.3 when the percentage of global transactions is respectively 15%, 5%, 1%, and 0%. When no transactions are global, each group acts as a completely independent replicated system, the corresponding curve in Figure 3(a) thus only serves as an illustrative purpose.



(a) 4 groups, varying the percentage of global transactions



(b) 25% of global transactions, varying the number of groups

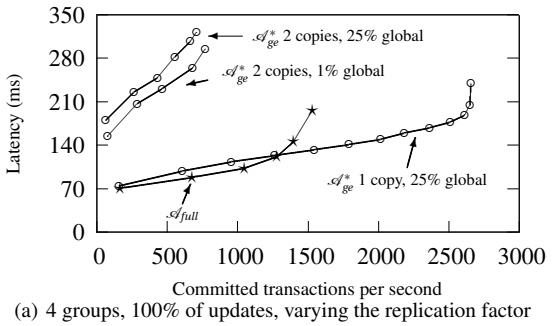
Figure 3: The scalability of $\mathscr{A}_{full}$ and $\mathscr{A}^*_{ge}$ when the percentage of global transactions and the number of groups vary.

Varying the number of groups In Figure 3(b), we study the scalability of $\mathscr{A}^*_{ge}$ and $\mathscr{A}_{full}$ when the number of groups varies and consider 25% of global transactions.

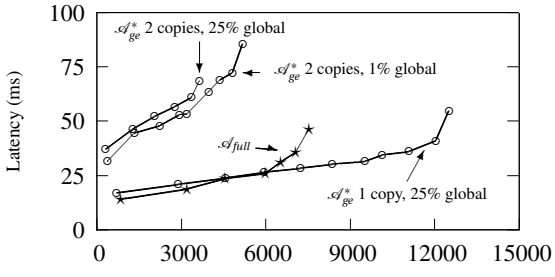
$\mathscr{A}_{full}$ does not scale when the number of groups increases. In fact, $\mathscr{A}_{full}$ performs best with two groups; with eight groups the execution is network-bound. In contrast, $\mathscr{A}^*_{ge}$ presents a scale-out of roughly 0.7 up to eight groups, that is, multiplying the number of groups by k increases the peak throughput of $\mathscr{A}^*_{ge}$ by a factor of $0.7k$. Moreover, $\mathscr{A}^*_{ge}$ with eight groups can support a load that is 1.7 times higher than the best peak throughput of $\mathscr{A}_{full}$ among all considered system sizes. This shows that $\mathscr{A}^*_{ge}$ offers good scalability even when 25% of the workload involves multiple groups.

We note that at peak loads, from 7% to 15% of transactions were aborted by $\mathscr{A}^*_{ge}$—$\mathscr{A}_{full}$ never aborted more than 5% of the transactions. In $\mathscr{A}^*_{ge}$, aborts were primarily caused by the optimistic reads of global transactions. We are currently working on techniques to reduce this phenomenon.

7.4 Replicating Data across Groups



(a) 4 groups, 100% of updates, varying the replication factor



(b) 4 groups, 20% of updates, varying the replication factor

Figure 4: The scalability of $\mathscr{A}_{full}$ and $\mathscr{A}^*_{ge}$ when the replication factor and the percentage of update transactions vary.

To reduce the amount of data items fetched from remote groups and increase the locality of the execution, it may be interesting to replicate data items in multiple groups. To evaluate this idea, the database is split into as many partitions as there are groups, and each group replicates x partitions. We denote x as the replication factor or number of copies. With x equal to one, the data is perfectly partitioned such that no two groups replicate the same data item—this is the setting $\mathscr{A}^*_{ge}$ used up to now. With x equal to the number of groups, we fall back to full replication, i.e., $\mathscr{A}_{full}$. In Figures 4(a) and 4(b), we report results in a system with four groups and respectively 100% and 20% percent of updates—read-only transactions read a single account. Recall that update transactions also perform reads and may benefit from replicating data items in multiple groups to allow local reads.

Maintaining copies of each data item in two groups harms the latency and scalability of $\mathscr{A}^*_{ge}$ dramatically with 25% of global transactions (see Figure 4(a)). Transactions are now atomically multicast to twice as many groups to be certified. In particular, global transactions must be multicast to all four groups of the system. Interestingly, reducing the percentage of global transactions to 1% does not affect this result significantly. This is because with two copies, the updates of local transactions must be applied to two groups, compared to only one group otherwise. This situation does not change when the workload is composed of only 20% of updates (see Figure 4(b)). In Figures 4(a) and 4(b), $\mathscr{A}_{full}$ provides better performance than $\mathscr{A}^*_{ge}$ with two copies and regardless of the proportion of global transactions. This suggests that performing remote reads is less costly than replicating data across groups.

7.5 Summary

Based on the results above, we provide a tentative decision diagram to determine whether to deploy full or partial replication given the workload. Figure 5 advocates partial replication when the number of groups is important and global transactions access few groups. We suspect that the percentage of read-only transactions does not affect the decision procedure, except in special cases (e.g., when the majority of read-only transactions read data items from multiple groups). Further refining this decision procedure is future work.

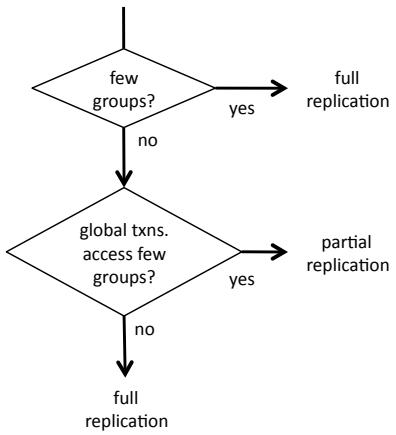


Figure 5: Deciding between full and partial replication.

8 Conclusion

P-Store is a partial database replication protocol for wide area networks that allows transactions to optimistically execute at multiple sites and certifies transactions in a genuine manner: to certify a transaction T only sites that replicate data items read or written by T exchange messages. The certification protocol $\mathscr{A}^*_{ge}$ proposed in this paper allows to reduce the convoy effect by certifying non-conflicting transactions in parallel. To guarantee serializability of the transaction execution, P-Store makes a thrifty use of an atomic multicast service: a single message is atomically multicast during T's certification. This is in contrast to previously proposed solutions that either do not allow transactions to execute at multiple sites, are non-genuine, or invoke the atomic multicast service multiple times to handle each transaction.

Our experimental results show that the certification protocol $\mathscr{A}^*_{ge}$ effectively reduces the convoy effect and doubles the peak throughput of P-Store even when only one percent of transactions are global. We also observed that P-Store scales better than a fully replicated solution when the percentage of global transactions is no more than twenty-five percent and provides an almost linear scale-out up to at least eight groups.

As future work, we plan to further investigate the parameters that influence the scalability of partial replication and refine our decision procedure to determine when partial replication is a better choice than full replication.

References

[1] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal, "Fault-tolerant total order multicast to asynchronous groups," in *Proceedings of SRDS'98*. IEEE Computer Society, pp. 578–585.

[2] P. Sutra and M. Shapiro, "Fault-tolerant partial replication in large-scale database systems," in *Proceedings of Euro-Par'08*, pp. 404–413.

[3] D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation," in *Proceedings of PRDC '07*. Washington, DC, USA: IEEE Computer Society, pp. 290–297.

[4] J. E. Armendáriz, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz, "Sipre: a partial database replication protocol with si replicas," in *Proceedings of SAC '08*. New York, NY, USA: ACM, pp. 2181–2185.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[6] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[7] C. Beeri, P. A. Bernstein, and N. Goodman, "A model for concurrency in nested transactions systems," *J. ACM*, vol. 36, no. 2, pp. 230–269, 1989.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] N. Schiper and F. Pedone, "Solving atomic multicast when groups crash," in *Proceedings of OPODIS'08*, pp. 481–495.

[10] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 297–316, 2001.

[11] N. Schiper, R. Schmidt, and F. Pedone, "Optimistic algorithms for partial database replication," in *In Proceedings of OPODIS'06*. Springer, pp. 81–93.

[12] U. Fritzke and P. Ingels, "Transactions on partially replicated data based on reliable and atomic multicasts." in *Proceedings of ICDCS'01*. IEEE Computer Society, pp. 284–291.

[13] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *The VLDB Journal*, 2000, pp. 134–143.

[14] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 375–423, 2005.

[15] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of SIGMOD '05*. New York, NY, USA: ACM, pp. 419–430.

[16] F. Pedone and S. Frølund, "Pronto: High availability for standard off-the-shelf databases," *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 150–164, 2008.

[17] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Consistent data replication: Is it feasible in wans?" in *Proceedings of Euro-Par'05*, pp. 633–643.

[18] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-jdbc: Flexible database clustering middleware," in *USENIX Annual Technical Conference, FREENIX Track*, 2004, pp. 9–18.

[19] C. Coulon, E. Pacitti, and P. Valduriez, "Consistency management for partial replication in a high performance database cluster," in *Proceedings of ICPADS'05*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, pp. 809–815.

[20] N. Schiper and F. Pedone, "On the inherent cost of atomic broadcast and multicast in wide area networks," in *Proceedings of ICDCN'08*. Springer, pp. 147–157.

[21] L. Camargos, P. Sutra, and N. Schiper, "http://sourceforge.net/projects/daisylib/."

[22] "Transaction processing performance council (tpc) - benchmark b," http://www.tpc.org/tpcb/.

We only give the proof of P-Store when used with certification protocol $\mathscr{A}^*_{ge}$ as $\mathscr{A}_{ge}$ is a special case of $\mathscr{A}^*_{ge}$. We show P-Store ensures one-copy serializability, *termination*, *non-trivial certification*, and *genuine partial replication* when used with $\mathscr{A}^*_{ge}$ and the genuine atomic multicast algorithms in [20, 1, 9].

.1 Proof of Correctness

Definition .1. *We define the binary relation $<_T$ on transactions as follows: $T_1 <_T T_2$ iff $\exists s_i \in \Pi : s_i$ A-Delivers T_1 before T_2. Moreover, let $\mathscr{G}_{<(T)} = (V, E)$ be a finite DAG constructed as follows:*

1. *add vertex T to V*
2. *while $\exists T_1 \in V : \exists T_2 \notin V : T_2 <_T T_1$ do:*
   add T_2 to V and add directed edge $T_2 \rightarrow T_1$ to E

For any transaction T' in $\mathscr{G}_{<(T)}$, we say that T' is at distance k of T iff the longest path from T' to T is of length k. We let $\mathscr{T}_k$ be the subset of transactions in $\mathscr{G}_{<(T)}$ that are at distance k of T.

Lemma .1. *For any submitted transaction T, $\mathscr{G}_{<(T)}$ is acyclic.*

Proof: Follows directly from the uniform acyclic order property of atomic multicast. □

Definition .2. *We define $Certify(T)_i$ as the returned value of the function $Certify(T)$ called on s_i. If there exists no invocation of function $Certify(T)$ on s_i we say that $Certify(T)$ is undefined, and we write $Certify(T)_i = \perp$.*

Definition .3. *We define $vote(VQ, T)$, voting quorum VQ's vote for transaction T ($VQ \in VQ(T)$), considering the VOTE messages of all $q \in VQ$ as follows:*

- $vote(VQ, T) = yes$ *iff* $\forall s_i \in VQ : Certify(T)_i = yes$
- $vote(VQ, T) = no$ *iff* $\forall s_i \in VQ : Certify(T)_i \neq \perp \wedge \exists s_j \in VQ : Certify(T)_j = no$
- $vote(VQ, T) = \perp$ *iff* $\exists s_i \in VQ : Certify(T)_i = \perp$

Definition .4. *For any submitted transaction T and any key k read by T, we define $Version(k, s_i)^T$ as the value of $Version(k, s_i)$ on site s_i after s_i executed line 8 for transaction T. If s_i never executes line 8 for transaction T or s_i does not replicate k, then $Version(k, s_i)^T = \perp$.*

Lemma .2. *For any submitted transaction T:*

1. *There does not exist $VQ_1, VQ_2 \in VQ(T)$ such that $vote(VQ_1, T) = yes$ and $vote(VQ_2, T) = no$.*
2. *For any key $k \in T.ws$, for any two sites s_i, s_j such that $k \in Items(s_i)$ and $k \in Items(s_j)$, if $Version(k, s_i)^T \neq \perp$ and $Version(k, s_j)^T \neq \perp$, then $Version(k, s_i)^T = Version(k, s_j)^T$.*

Proof: Let $\mathscr{T}_k$ be the subset of the transactions in $\mathscr{G}_{<(T)}$ that are at distance k of T. We show that for any k and any $T' \in \mathscr{T}_k$, 1) and 2) hold. Since $T \in \mathscr{T}_0$, this shows the two claims. Let k_{max} be the largest k such that $\mathscr{T}_k \neq \emptyset$. We proceed by simultaneous induction on 1) and 2), starting from k_{max}.

- Base step ($k = k_{max}$):
  1. From the definition of $\mathscr{T}_{k_{max}}$, there exists no transaction $T' \in \mathscr{T}_{k_{max}}$ such that a site s_i A-Delivers a transaction T'' before T'. Hence, on all sites s_i such that $Certify(T)_i \neq \perp$, $Certify(T)_i = yes$. Hence, for all $VQ \in VQ(T')$, $vote(VQ, T') = yes$.
  2. From the definition of $\mathscr{T}_{k_{max}}$ and the algorithm, for any transaction $T' \in \mathscr{T}_{k_{max}}$ and any key $k \in T'.ws$, on all sites s_i such that $k \in Items(s_i)$ and $Version(k, s_i)^{T'}$ is defined, $Version(k, s_i)^{T'} = 1$.
- Induction step: Suppose that the two claims hold for any k such that $0 < k \leq k_{max}$, we show that they also hold for $k-1$. Let T' be any transaction in $\mathscr{T}_{k-1}$.
  1. Suppose, by way of contradiction, that $vote(VQ_1, T') = yes$ and $vote(VQ_2, T') = no$. Hence, there exists a site $s_i \in VQ_2$ such that $Certify(T')_i = no$. Consequently, there exists a tuple $(k, ts) \in T.rs$ such that $ts \neq Version(k, s_i)^{T'}$. From the algorithm, there exists a transaction T'' such that s_i A-Delivers T'' just before T', T'' commits and updates k, and $Version(k, s_i)^{T''} = ts$ (just after T'' commits, $Version(k, s_i)$ is incremented and becomes greater than ts). From the definition of a voting quorum, there exists a site $s_j \in VQ_1$ such that $k \in Items(s_j)$, and thus, T'' is also atomically multicast to s_j. From the uniform prefix order property of atomic multicast, either (i) s_i A-Delivers T' before T'' or (ii) s_j A-Delivers T'' before T'. Case (i) is impossible as s_i would A-Deliver T' twice, a contradiction to the uniform integrity property of atomic multicast. Therefore, s_j A-Delivers T'' before T'. From the induction hypothesis of 1), all voting quorums for T'' vote similarly, and thus, since s_i commits T'', s_j commits T'' as well. From the induction hypothesis of 2), $Version(k, s_i)^{T''} = Version(k, s_j)^{T''}$.

Hence, since s_j A-Delivers T'' before T' and T'' read-write conflicts with T', s_j increments $Version(k, s_j)$ to a greater value than ts before certifying T', and thus Certify$(T')_j$ = no, a contradiction to the fact that $s_j \in VQ_1$ and $vote(VQ_1, T')$ = yes.

2. Let T'' be the last transaction that commits on s_i before T' such that T'' updates k. Using a similar argument as in the induction step of 1), we can show that s_j A-Delivers T'' before T'. From the induction hypothesis of 1), T'' commits on s_j.

   We now show that (*), on s_j, T'' is also the last transaction that updates k and commits before T'. Suppose, by way of contradiction, that there exists a transaction T''' that commits after T'' and before T' on s_j such that $k \in T'''.ws$. From the uniform prefix order of atomic multicast, either (i) s_i A-Delivers T''' before T' or (ii) s_j A-Delivers T' before T'''. Case (ii) is impossible as s_j would A-Deliver T' twice, a contradiction to the uniform integrity property of atomic multicast. Hence, s_i A-Delivers T''' before T'. Now, either (iii) s_i A-Delivers T'' before T''' or (iv) s_j A-Delivers T''' before T''. Case (iv) is impossible for the same reason as case (ii). Therefore, s_i A-Delivers, in order, T'', T''', and T'. From the induction hypothesis of 1), s_i also commits T'''. Consequently, the last transaction that commits before T' on s_i and updates k is T''', a contradiction to the definition of T''.

   By the induction hypothesis of 2), $Version(k, s_i)^{T''} = Version(k, s_j)^{T''}$. Therefore, from (*) and the algorithm, $Version(k, s_i)^{T'} = Version(k, s_j)^{T'}$. □

Proposition .1. (Safety) *There exists a serial one-copy history 1H that is view-equivalent to H.*

Proof: We first explain how to construct *1H* and then show that *1H* is view-equivalent to H. History *1H* is constructed in the following way:

1. History *1H* is composed of the same transactions as H.

2. A read operation $r_T[k]$ of transaction T in H is mapped to the same operation $r_T[k]$ of T in *1H*. Write operations $w_T[k_A, v, ts]$, $w_T[k_B, v, ts]$, ..., $w_T[k_N, v, ts]$ of transaction T in H is mapped to a single write operation $w_T[k, v, ts]$ of transaction T in *1H*.

3. The order of transactions in *1H* is defined by means of a total order relation $<_{1H}$ on transactions. Recall that for any two transactions $T, T' \in H$, $T <_T T'$ iff there exists a site that A-Delivers T before T'. Relation $<_{1H}$ is defined as follows, $\forall T, T' \in 1H: T <_{1H} T'$ iff one of the four following conditions holds:

   (a) T and T' are update transactions and $T <_T T'$

   (b) T and T' are update transactions, neither $T <_T T'$ nor $T' <_T T$, and $T.id < T'.id$

   (c) Either T is read-only and T' is an update transaction or vice-versa and
   - $\exists s_i \in Sites(T) \cap Sites(T')$ and T commits before T' on s_i or
   - $Sites(T) \cap Sites(T') = \emptyset$ and $T.id < T'.id$

   (d) T and T' are both read-only, and $T.id < T'.id$.

4. In *1H*, for any two transactions T and T', their respective operations do not interleave.

We now show that *1H* is view-equivalent to H. For *1H* to be view-equivalent to H, three conditions have to be fulfilled [8]:

1. H and *1H* are defined over the same set of transactions,

2. H and *1H* have the same *reads-x-from* relationships on data items: $\forall T, T' : T'$ *reads-x-from* T in $H \iff T'$ *reads-x-from* T in *1H*.

3. For each final write $w_T[k, v, ts]$ in *1H*, $w_T[k_a, v, ts]$ is also a final write in H for some copy k_a of key k.

H is view-equivalent to *1H*:

1. Clear from construction step 1 of *1H*.

2. ($\Rightarrow$) Let T, T' be two transactions such that T' *reads-x-from* T in H. We prove that T' *reads-x-from* T in *1H*. Obviously, T is an update transaction, and either (a) T' is a local and read-only transaction or (b) T' is a global or update transaction.

   In case (a), let s_i be the site on which T' executes. Since databases ensure order-preserving serializability, and T' *reads-x-from* T in H, on s_i T' commits after T does but before any transaction T'' that updates key x. From construction step 3 of *1H*, $T <_{1H} T'$, and for any transaction T'' that updates x, either $T'' <_{1H} T$ or $T' <_{1H} T''$. Therefore, from construction step 4, T' *reads-x-from* T in *1H*.

   In case (b), since T' commits, T' passed the certification test. Consequently, T' read up-to-date data items and there exists no transaction T'' that updates key x such that $T <_T T'' <_T T'$. Therefore, by construction step 3 of *1H*, there

exists no such transaction T'' in *1H* such that $T <_{1H} T'' <_{1H} T'$, and hence, by construction step 4, T' *reads-x-from* T in *1H*.

($\Leftarrow$) Let T, T' be two transactions such that T' *reads-x-from* T in *1H*. We prove that T' *reads-x-from* T in H. If T' *reads-x-from* T in *1H*, from construct step 4 of *1H* (*) there exists no transaction T'' that commits between T and T' and updates key x in *1H*. There are two cases to consider, either (a) T' is a local and read-only transaction or (b) T' is a global or update transaction.

In case (a), from construction step 3 of *1H* and (*), the site on which T' executes commits T' just after T but before any transaction that updates key x. Since databases guarantee order-preserving serializability, we conclude that T' *reads-x-from* T in H.

In case (b), from construction step 3 of *1H* and (*), there exists no transaction T'' that updates key x such that $T <_T T'' <_T T'$. Since T' commits, T' passed the certification test and, consequently, T' read up-to-date data items. Because databases ensure order-preserving serializability, T' *reads-x-from* T in H.

3. Clear from construction step 1 and the definition of relation $<_{1H}$ (construction step 3). □

Lemma .3. *For any submitted transaction T and correct site s_i such that s_i A-Delivers T:*

1. *line 14 on s_i eventually evaluates to true for T, and,*
2. *line 19 on s_i eventually evaluates to true for T.*

Proof: Let $\mathscr{T}_k$ be the subset of the transactions in $\mathscr{G}_{<(T)}$ that are at distance k of T. We show that for any k and any $T' \in \mathscr{T}_k$, 1) and 2) hold. Since $T \in \mathscr{T}_0$, this shows the two claims. Let k_{max} be the largest k such that $\mathscr{T}_k \neq \emptyset$. We proceed by simultaneous induction on 1) and 2), starting from k_{max}.

- Base step ($k = k_{max}$):
  1. From the definition of $\mathscr{T}_{k_{max}}$, there exists no transaction $T' \in \mathscr{T}_{k_{max}}$ such that a site A-Delivers a transaction T'' before T'. Hence, on a correct site s_i that A-Delivers T', line 14 evaluates to true as soon as s_i A-Delivers T'.
  2. From the definition of $\mathscr{T}_{k_{max}}$, on a correct site s_i that A-Delivers T', T' is the first transaction that s_i A-Delivers. From the base step of 1), s_i eventually inserts T' into *CertifyQ* and T' is the first transaction to be inserted in this queue. Now either (a) T' is local or (b) T' is global.
     - In case (a), line 19 evaluates to true as soon as s_i inserts T' into *CertifyQ*.
     - In case (b), line 19 evaluates to true as soon as s_i inserts T' into *CertifyQ* if $s_i \notin$ *WReplicas*(T'). Otherwise, if $s_i \in$ *WReplicas*(T'), since there exists a correct site for all data items in $\mathscr{D}$, from the base step of 1) a voting quorum of correct sites eventually certify T' and send their votes to *WReplicas*(T'). Because links are quasi-reliable, s_i eventually receives these votes and line 19 evaluates to true.
- Induction step: Suppose that the two claims hold for any k such that $0 < k \leq k_{max}$, we show that they also hold for $k-1$. Let T' be any transaction in $\mathscr{T}_{k-1}$.
  1. From the induction hypothesis, for a correct site s_i that A-Delivers T', for all transactions s_i A-Delivers before T' line 19 eventually evaluates to true. From the algorithm, these transactions are thus removed from *CertifyQ* and line 14 eventually evaluates to true for T' on s_i.
  2. From the induction hypothesis, all transactions that are A-Delivered before T' on a correct site s_i are eventually removed from *CertifyQ*. Consequently, there exists a time at which T' is at the head of *CertifyQ* on s_i. There are two cases to consider, either (a) T' is local or (b) T' is global.
     - In case (a), line 19 evaluates to true on s_i as soon as T' is at the head of *CertifyQ*.
     - In case (b), if $s_i \notin$ *WReplicas*(T'), line 19 evaluates to true on s_i as soon as T' is at the head of *CertifyQ*. Otherwise, if $s_i \in$ *WReplicas*(T'), since there exists a correct site for all data items in $\mathscr{D}$, from the induction step of 1), a voting quorum of correct sites eventually certify T' and send their votes to *WReplicas*(T'). Because links are quasi-reliable, s_i eventually receives these votes and line 19 evaluates to true. □

Proposition .2. ***(termination)*** *For every submitted transaction T, if Proxy(T) is correct, then all correct sites in WReplicas(T) either commit or abort T.*

Proof: Since *Proxy*(T) is correct, *Proxy*(T) A-MCasts(T) and by the validity property of atomic multicast, all correct sites in *Replicas*(T) eventually A-Deliver T. By Lemma .3-2 and .2, all correct sites in *WReplicas*(T) either commit or abort T. □

Proposition .3. **(*non-trivial certification*)** *If there is a time after which no two read-write conflicting transactions are submitted, then eventually transactions are not aborted by certification.*

Proof: Let t_1 be the time after which no two read-write conflicting transactions are submitted. Let $t_2 > t_1$ be the time after which the last transaction T submitted before t_1 commits. We claim that no transaction submitted after t_2 is aborted by certification. Indeed, for any transaction T submitted after t_2, the call of function Certify return *yes* and the condition at line 20 always evaluates to true. This is because on any site s_i that certifies T, and for any tuple $(k, ts) \in T.rs$ such that s_i replicates k, $ts = Version(k, s_i)^T$. □

Proposition .4. **(*genuine partial replication*)** *Using the genuine atomic multicast algorithm in [20, 9, 1], for any submitted transaction T, only database sites that replicate data items read or written by T exchange messages to certify T.*

Proof: Since to certify T, T is atomically multicast to sites that replicate data items read or written by T and votes are only sent to $WReplicas(T)$, the claim holds. □