Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

Geo-Replicated Storage with Scalable Deferred Update Replication

Daniele Sciascia[1], Fernando Pedone[1]

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

Abstract

Many current online services are deployed over geographically distributed sites (i.e., datacenters). Such distributed services call for geo-replicated storage, that is, storage distributed and replicated among many sites. Geographical distribution and replication can improve locality and availability of a service. Locality is achieved by moving data closer to the users. High availability is attained by replicating data in multiple servers and sites. This paper considers a class of scalable replicated storage systems based on deferred update replication with transactional properties. The paper discusses different ways to deploy scalable deferred update replication in geographically distributed systems, considers the implications of these deployments on user-perceived latency, and proposes solutions. Our results are substantiated by a series of microbenchmarks and a social network application.

Report Info

Published
August 2015

Number
USI-INF-TR-2015-3

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Many current online services are deployed over geographically distributed sites (i.e., datacenters). Such distributed services call for *geo-replicated storage*, that is, storage distributed and replicated among many sites. Geographic distribution and replication can improve locality and availability of a service. Locality is achieved by moving the data closer to the users and is important because it improves user-perceived latency. High availability is attained by deploying the service in multiple replicas; it can be configured to tolerate the crash of a few nodes within a datacenter or the crash of multiple sites, possibly placed in different geographical locations.

In this paper, we consider a class of scalable replicated storage systems based on deferred update replication. Deferred update replication is a well-established approach (e.g., [27],[16], [21], [24]). The idea behind a scalable deferred update replication (S-DUR) protocol is conceptually simple: the database is divided into partitions and each partition is fully replicated by a group of servers [32]. To execute a transaction, a client interacts with (at most) one server per partition and there is no coordination among servers during the execution of the transaction—essentially, the technique relies on optimistic concurrency control [18]. When the client wishes to commit the transaction, he atomically broadcasts the transaction's updates (and some meta data) to each partition involved in the transaction. Atomic broadcast ensures that servers in a partition deliver the updates in the same order and can certify and possibly commit the transaction in the same way. Certification guarantees that the transaction can be serialized with other concurrent transactions within the partition. Transactions are globally serialized using a two-phase commit-like protocol: servers in the involved partitions exchange the outcome of certification (i.e., the partition's vote) and if the transaction passes the certification test successfully at all involved partitions it is committed; otherwise it is aborted.

Scalable deferred update replication offers good performance, which grows proportionally with the number of database partitions in workloads dominated by single-partition transactions [32], however, it is obli-

vious to the geographical location of clients and servers. While the actual location of clients and servers is irrelevant for the correctness of S-DUR, it has important consequences on the latency perceived by the clients. S-DUR distinguishes between local transactions, those that access data in a single partition, and global transactions, those that access data in multiple partitions. Intuitively, a local transaction will experience lower latency than a global transaction since it does not require the two-phase commit-like termination needed by global transactions. Moreover, in a geographically distributed environment, the latency gap between local and global transactions is likely wider since the termination of global transactions may involve servers in remote regions, subject to longer communication delays. This is not the case for local transactions whose partition servers are within the same region. Applications can exploit these tradeoffs by distributing and replicating data to improve locality and maximize the use of local transactions.

Although local transactions are "cheaper" than global transactions when considered individually, in mixed workloads global transactions may hinder the latency advantage of local transactions. This happens because within a partition, the certification and commitment of transactions is serialized to ensure determinism, a property without which replicas' state would diverge. As a consequence, a local transaction delivered after a global transaction will experience a longer delay than if executed in isolation. We have assessed this phenomenon in a geographically distributed environment and found that even a fairly low number of global transactions in the workload is enough to increase the average latency of local transactions by more than 10 times.

We propose three different techniques to avoid the increased latency. The first technique tries to ensure that a global transaction is delivered by all involved partitions at approximately the same time. Doing so reduces the chances that a local transaction is unnecesarily delivered after a global transaction.

The remaining two techniques try to reorder local transactions against global transactions that were previously delivered but are not committed yet. As we will discuss, two transactions may be reordered only if the resulting ordering does not introduce conflicts. Moreover the decision to reorder to given transactions must be deterministic within a partition, every replica has to reach the same decision with respect to whether to reorder two transactions or not.

This paper makes the following contributions. First, it revisits scalable deferred update replication and discusses how it can be deployed in geographically distributed systems. Second, it experimentally assesses the performance of these deployments, using Amazon's elastic compute infrastructure, and quantifies the problems mentioned above. Third, it proposes three extensions to address the limitations of the original protocol and presents a detailed experimental analysis of their effectiveness. Our experimental study considers a series of microbenchmarks and a social network application, prototypical of current online services deployed over geographically distributed sites.

The remainder of the paper is structured as follows. Section 2 presents our system model and some definitions. Section 3 recalls the scalable deferred update replication approach and introduces a novel algorithm that implements certification-less global read-only transactions. Section 4 discusses how to deploy S-DUR in a geographically distributed system, points out performance issues with these deployments, and details solutions to the problems. Section 5 evaluates the performance of the protocol under different conditions. Section 6 reviews related work and Section 7 concludes the paper.

2 System model and definitions

In this section, we define our system model and introduce some definitions used throughout the paper.

2.1 Processes and communication

We consider a distributed system composed of an unbounded set $C = \{c_1, c_2, ...\}$ of *client* processes and a set $S = \{s_1, ..., s_n\}$ of *server* processes. Set S is divided into P disjoint groups, $S_1, ..., S_P$. The system is asynchronous: there is no bound on messages delays and on relative process speeds. We assume the crash failure model according to which a process may fail by halting but does not present arbitrary behavior (e.g., no Byzantine failures). A process, either client or server, that never crashes is *correct*, otherwise it is *faulty*.

Processes communicate using either one-to-one or one-to-many communication. One-to-one communication uses primitives $send(m)$ and $receive(m)$, where m is a message. Links are quasi-reliable: if both the sender and the receiver are correct, then every message sent is eventually received. One-to-many communication relies on atomic broadcast, implemented within each group p. Atomic broadcast is defined by primitives $abcast(p, m)$ and $adeliver(p, m)$ and ensures two properties: (1) if message m is delivered by

a server in p, then every correct server in p eventually delivers m; and (2) no two messages are delivered in different order by their receivers.

While several atomic broadcast algorithms exist [14], we use Paxos to implement atomic broadcast within a group of servers [20]. Paxos requires a majority of correct servers within a group and additional assumptions to ensure liveness, notably a leader-election oracle at each group [20].

2.2 Databases, transactions and serializability

The database is a set $D = \{x_1, x_2, ...\}$ of data items. Each data item x is a tuple $\langle k, v, ts \rangle$, where k is a key, v its value, and ts its version—we assume a multiversion database. The database is divided into P *partitions* and each partition p is replicated by servers in group S_p. Hereafter, we assume that atomic broadcast can be solved within each partition. For brevity, we say that server s belongs to partition p meaning that $s \in S_p$, and that p performs an action (e.g., sending a message) with the meaning that some server in p performs the action. For each key k, we denote $partition(k)$ the partition to which k belongs.

A transaction is a sequence of read and write operations on data items followed by a commit or an abort operation. We represent a transaction t as a tuple $\langle id, rs, ws \rangle$ where id is a unique identifier for t, rs is the set of data items read by t ($readset(t)$) and ws is the set of data items written by t ($writeset(t)$). The set of items read or written by t is denoted by $Items(t)$. The readset of t contains the keys of the items read by t; the writeset of t contains both the keys and the values of the items updated by t. We assume that transactions do not issue "blind writes", that is, before writing an item x, the transaction reads x. More precisely, for any transaction t, $writeset(t) \subseteq readset(t)$. Transaction t is said to be *local* if there is a partition p such that $\forall (k, -) \in Items(t) : partition(k) = p$. If t is not a local transaction, then we say that t is *global*. The set of partitions that contain items read or written by t is denoted by $partitions(t)$.

The isolation property is *serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [5].

3 Scalable Deferred Update Replication

Scalable deferred update replication (S-DUR) is an extension of deferred update replication that accounts for partitioned data. In this section, we recall how S-DUR works, introduce a novel algorithm that implements certification-less read-only transactions and discuss the performance of S-DUR.

3.1 Transaction execution

In S-DUR, the lifetime of a transaction is divided in two phases: (1) the *execution phase* and (2) the *termination phase*. The execution phase starts when the client issues the first transaction operation and finishes when the client requests to commit or abort the transaction, when the termination phase starts. The termination phase finishes when the transaction is completed (i.e., committed or aborted).

During the execution phase of a transaction t, client c submits each read operation of t to a server s in the partition p that contains the item read. This assumes that clients are aware of the partitioning scheme.[1] When s receives a read command for x from c, it returns the value of x and its corresponding version. The first read determines the *database snapshot* at partition p the client will see upon executing other read operations for t. Therefore, reads within a single partition see a consistent view of the database. Transactions that read from multiple partitions must either be certified at termination to check the consistency of snapshots or request a globally-consistent snapshot upon start; globally-consistent snapshots, however, may observe an outdated database since they are built asynchronously by servers. Write operations are locally buffered by c and only propagated to the servers during transaction termination.

Read-only transactions execute against a globally-consistent snapshot and commit without certification. Update transactions must pass through the termination phase to commit, as we describe next.

3.2 Transaction termination

To request the commit of t, c atomically broadcasts to each partition p accessed by t the subset of t's readset and writeset related to p, denoted $readset(t)_p$ and $writset(t)_p$. Client c uses one broadcast operation per

[1] Alternatively, a client can connect to a single server s and submit all its read requests to s, which will then route them to the appropriate partition.

partition—running a system-wide atomic broadcast would result in a non-scalable architecture. Upon delivering t's $readset(t)_p$ and $writeset(t)_p$, a server s in p certifies t against transactions delivered before t in p—since certification within a partition is deterministic, every server in p will reach the same outcome for t. If t passes certification, it becomes a *pending* transaction in s; otherwise s aborts t. If t is a local transaction, it will be committed after s applies its changes to the database. If t is a global transaction, s will send the outcome of certification, the partition's *vote*, to the servers in $partitions(t)$ and wait for the votes from $partitions(t)$. If each partition votes to commit t, s applies t's updates to the database (i.e., commit); otherwise s aborts t.

The certification of a local transaction checks whether the transaction can be serialized according to its delivery order. If transactions t_i and t_j executed concurrently in partition p and t_i is delivered before t_j, t_j will pass certification with respect to t_i if $readset(t_j)_p \cap writeset(t_i)_p = \emptyset$. Logically, in a serial execution where t_i executed before t_j, t_j would see any of t_i's updates. Since t_i and t_j executed concurrently, certification allows t_j to commit only if t_j did not read any item updated by t_i. This relatively simple certification test is possible thanks of the totally ordered delivery of transactions within a partition, implemented by atomic broadcast.

The certification of global transactions is more complex due to the absence of total order across partitions. If t_i and t_j are concurrent global transactions that read from partitions p_x and p_y, it may happen that t_i is delivered before t_j at p_x and t_j is delivered before t_i at p_y. Simply certifying that t_j does not read any item updated by t_i at p_x and t_i does not read any item updated by t_j at p_y does not ensure serializability.[2] To enforce serializable executions without system-wide total order, S-DUR uses a more strict certification test for global transactions: If global transaction t_i executed concurrently with transaction t_j in partition p, and t_j is delivered before t_i, t_i will pass certification with respect to t_j if $readset(t_j)_p \cap writeset(t_i)_p = \emptyset$ *and* $readset(t_i)_p \cap writeset(t_j)_p = \emptyset$. Intuitively, this means that if t_i and t_j pass certification, they can be serialized in any order—thus, it does not matter if t_i is delivered before t_j at one partition and t_j is delivered before t_i at another partition.

ALGORITHM 1: Scalable Deferred Update Replication, client c

```
1:  function begin(t)
2:      t.rs ← ∅                                          {initialize readset}
3:      t.ws ← ∅                                          {initialize writeset}
4:      t.st[1...P] ← [⊥...⊥]                             {initialize vector of snapshot times}
5:  function read(t, k)
6:      t.rs ← t.rs ∪ {k}                                 {add key to readset}
7:      if (k, ⋆) ∈ t.ws then                             {if key previously written...}
8:          return v s.t. (k, v) ∈ t.ws                   {return written value}
9:      else                                              {else, if key never written...}
10:         p ← partition(k)                              {get the key's partition}
11:         send(read, k, t.st[p]) to s ∈ S_p              {send read request}
12:         wait until receive(k, v, st) from s           {wait response}
13:         if t.st[p]=⊥ then t.st[p]←st                  {if first read, init snapshot}
14:         return v                                      {return value from server}
15: function write(t, k, v)
16:     t.ws ← t.ws ∪ {(k, v)}                            {add key to writeset}
17: function commit(t)
18:     send(commit, t) to a preferred server s near c
19:     wait until receive(outcome) from s
20:     return outcome                                    {outcome is either commit or abort}
```

3.3 The algorithm in detail

Algorithm 1 shows the client for scalable deferred update replication. To execute a read, the client first figures out which partition stores the key to be read, and sends the request to one of the servers in that partition (lines 10–12). Notice that the snapshot of a transaction is an array of snapshots, one for each partition (line 4).

[2] To see why, let t_i read x and write y and t_j read y and write x, where x and y are items in p_x and p_y, respectively. If t_i is delivered before t_j at p_x and t_j is delivered before t_i at p_y, both pass certification at p_x and p_y, but they cannot be serialized.

Upon receiving the first response from the server, the client initializes its snapshot time for the corresponding partition (line 13). Subsequent requests to the same partition will include the snapshot count so that reads to the same partition observe a consistent view. At commit time (lines 18–20), the client submits a transaction to one server (line 18), which will be broadcast for certification to all partitions concerned by the transaction. The client then waits for a server that replies with the transaction's outcome (lines 19–20).

Algorithm 2 shows the server side. When local transaction t is delivered, it is certified (line 15) and appended to the queue of pending transactions (line 39), if local certification for t results in a commit. The order of the transactions in the pending queue *PL* follows the delivery order, and it is the same on every server within the same partition. The pending queue is consumed in order; when t is at the head of *PL* it can be completed (lines 20–23). To complete t, the protocol proceeds as follows: if t passes certification, it is applied to the local database, thus generating a new snapshot, and the outcome of t is sent to the client (lines 24–29).

The delivery of a global transaction t requires additional steps: it is certified (line 15), and the outcome of the local certification test is exchanged between servers in different partitions (lines 18–19). Servers keep track of the received votes in a set called *VOTES* (lines 12–13). Global transaction t can be completed when it is at the head of the pending queue and if enough votes have been received. Function *readyToCommit*() ensures that, for a given global transaction t, a vote from the partitions concerned by t (lines 20–23) has been received. For the algorithm to be correct, it is sufficient to wait for only one vote from every partition involved, as every server in the partition produces the same vote for a given transaction. The final outcome for global transaction t is commit if every partition voted for committing t (line 22); otherwise it is abort.

3.4 Handling partially terminated transactions

With scalable deferred update replication, a client may fail while executing the various atomic broadcasts involved in the termination of a global transaction t. It may be the case that some partitions deliver t's termination request while others do not. This is possible because the atomic broadcast only guarantees that within a partition all servers deliver the same sequence of transactions. However, there is no guarantee that if a partition delivers t then every partition involved by t also delivers t.

A partition p_k that delivers the request will certify t, send its vote to the other partitions involved in t, and wait for votes from the other partitions to decide on t's outcome. Obviously, a partition p_l that did not deliver t's termination request will never send its vote to the other partitions and t will remain partially terminated. Local transactions do not suffer from the same problem since atomic broadcast ensures that within a partition either a local transaction is delivered by all servers or by no server. To handle partially terminated transactions, a server s in p_k that does not receive p_l's vote after a certain time suspects that servers in p_l did not deliver t's termination request.

In this case, s broadcasts a termination request for t on behalf of c. Since s does not have the readset and writeset of t for p_l, it broadcasts a request to abort t at p_l. Notice that s may unjustifiably suspect that servers in p_l did not deliver t's termination request. However, atomic broadcast ensures that c's message requesting t's termination and s's message requesting t's abort are delivered by all servers in p_l in the same order.

Servers in p_l process the first message they deliver for t: c's message will lead to the certification of t at p_k; s's message will result in an abort vote. Whatever message is delivered first, no transaction will remain partially terminated.

3.5 Certification—less read-only transactions

In the protocol described so far, both read-only and update transactions must be certified to ensure that they can be serialized. Certifying read-only transactions is a disadvantage with respect to the baseline deferred update replication protocol, where read-only transactions commit without certification.

In the following we describe a protocol that can be used for creating read-only snapshots. A snapshot essentially consists of a set of snapshot counters, one per partition. A read-only transaction that reads from such a snapshot is guaranteed to be serializable with respect to update transactions. However, such read-only transactions are not necessarily exposed to the *freshest* snapshot. This algorithm can thus be used in cases when the client knows, a priori, that the transaction that is going to be executed is read-only, and can tolerate the fact that it may not see the latest results. Another use is to create consistent read-only snapshots of the dataset. This could be useful to create backups, or to create immutable snapshots of the state (i.e., using copy-on-write) to be accessed using read-only transactions.

To allow certification-less read-only transactions, we must consider both local and global read-only transactions. We illustrate these cases with examples.

ALGORITHM 2: Scalable Deferred Update Replication, server s in partition p

```
1:  Initialization
2:      DB ← [...]                                   {list of applied transactions}
3:      PL ← [...]                                   {list of pending transactions}
4:      SC ← 0                                       {snapshot counter}
5:      VOTES ← ∅                                    {votes for global transactions}
6:  when receive(read, k, st) from c
7:      if st = ⊥ then st ← SC                       {if first read, init snapshot}
8:      retrieve(k, v, st) from DB                   {most recent version ≤ st}
9:      send(k, v, st) to c                          {return result to client}
10: when receive(commit, t)
11:     for all q ∈ partitions(t) : abcast(t) to partition q
12: when receive(tid, v) from partition q
13:     VOTES ← VOTES ∪ (tid, q, v)                  {one more vote for tid}
14: when adeliver(t)
15:     vote ← certify(t)                            {see line 34}
16:     if vote = abort then                         {certification resulted in abort?}
17:         complete(t, abort)                       {see line 24}
18:     if t is global then
19:         send(t.id, vote) to all servers in partitions(t)   {send votes}
20: when readyToCommit(head(PL))
21:     t ← head(PL)                                 {get head without removing entry}
22:     outcome ← (t.id, ⋆, abort) ∈ VOTES           {one abort vote and t will be aborted}
23:     complete(t, outcome)                         {see line 24}
24: function complete(t, outcome)                    {used in lines 17, 23}
25:     PL ← PL ⊖ t                                  {remove t from PL}
26:     if outcome = commit then                     {if t commits...}
27:         DB[SC + 1] ← t                           {create next snapshot and...}
28:         SC ← SC + 1                              {...expose snapshot to clients}
29:     send(outcome) to client of t
30: function readyToCommit(t)
31:     return t is local ∨ ∀q ∈ partitions(t) : (t.id, q, ⋆) ∈ VOTES
32: function ctest(t, t′)
33:     return (t.rs ∩ t′.ws = ∅) ∧ (t is local ∨ (t.ws ∩ t′.rs = ∅))
34: function certify(t)                              {used in line 15}
35:     if ∃t′ ∈ DB[t.st[p] ... SC] : ctest(t, t′) = false then
36:         return abort                             {t aborts if conflicts with committed t′}
37:     if ∃t′ ∈ PL : ctest(t, t′) = false then
38:         return abort                             {t aborts if conflicts with pending t′}
39:     PL ← PL ⊕ t                                  {append t to pending list if no conflicts}
40:     return commit
```

Example 1. The problem with local read-only transactions is that although they may individually see a consistent database snapshot, together they may lead to an inconsistent execution.

Assume global transaction t_i updates both x_1 and y_1, and global transaction t_j updates x_2 and y_2. Transactions t_a and t_b are local and readonly: t_a reads x_1 and x_2; t_b reads y_1 and y_2 (see Figure 1). At p_x, t_i is delivered first and passes certification; then t_a is executed and reads the value of x_1 written by t_i and the initial value of x_2. Finally, t_j is delivered and since its readset does not intersect the writesets of t_i, it passes certification. At p_y, t_j is delivered first, followed by t_i. Transaction t_b reads the value of y_2 written by t_j, and the initial value of y_1. Similarly to what happens at p_x, all transactions commit, although the execution is not serializable. Notice that in this case global transactions t_i and t_j do not read or write any data items in common.

Example 2. Consider now a global read-only transaction that reads from the latest snapshot counters available at different partitions. For instance, assume that read-only transaction t reads from snapshots SC_x and SC_y at partitions p_x and p_y respectively. Transaction t may see an inconsistent view of the database, in

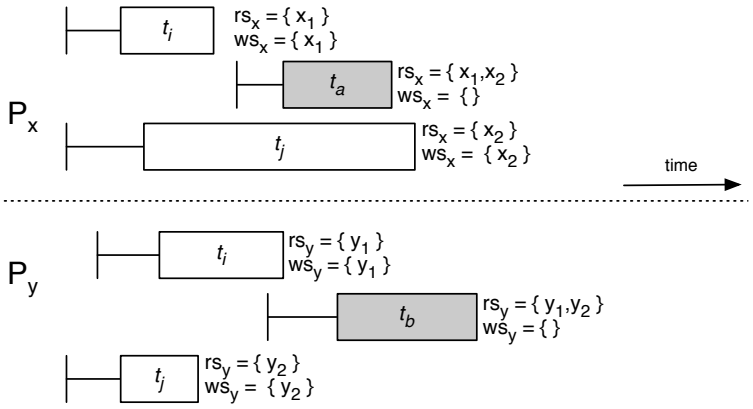

Figure 1: Illustration of Example 2. At partitions p_x, read-only transaction t_a observes the effects t_i but not the effects of t_j. At partition p_y, read-only transaction t_b observes the effects of t_j but not the effects of t_i. Thus violating serializability.

that SC_x may contain a global transaction t_g that is not included in SC_y. Potentially, t may see some of the updates of t_g at partition p_x, and miss some updates at partition p_y, thus violating serializability.

We define snapshot S as an n-tuple $S = \langle SC_1, SC_2, \ldots, SC_n \rangle$, where n is the number of partitions, and SC_i is the count of transactions that were committed at partition i. Intuitively, we must find a snapshot that can be serialized with the history of transactions. That is, for all transactions T that precede S, S's snapshot counter at partition i must be greater than or equal to the snapshot counter created by T at partition i. Similarly, for all transactions T that succeed S, S's snapshot counter at partition i must be smaller than the snapshot counter created by T at partition i.

Algorithm 3 shows how to find a serializable snapshot. For simplicity, we consider that each partition is composed of a single server. The complete protocol uses the broadcast primitive to ensure markers are received and processed in the same order within the same partition. The algorithm is inspired by the Chandy-Lamport snapshot protocol [7], in that it uses markers and relies on FIFO-channels. Function *snapshot*() (lines 1–3) is invoked by server s to initiate the snapshot algorithm. To take a snapshot a server sends the marker to itself. When a marker is received (lines 4–7) for the first time by server s, s will suspend the delivery of global transactions, and relay the marker to all other servers (in FIFO order). When server s receives a marker from all partitions, then a snapshot has been found and each server records its SC, the current snapshot counter. Collectively, the SC of all partitions is a snapshot that does not include partial transactions, thus avoiding the problem we illustrated in example 2. Notice that to ensure serializable snapshots, only one instance of the snapshotting algorithm can be active at any time. Otherwise, two concurrent instances of the algorithm could create snapshots that allow the execution of example 1. To ensure this, only one partition initiates the algorithm by calling function *snapshot*().

ALGORITHM 3: Snapshot Algorithm, server s in partition p

```
1:  function snapshot()                                  {Used by server to create a new snapshot}
2:      let marker be a unique integer
3:      send(marker) to s                                {server s sends marker to itself}
4:  when received(marker) from server q
5:      if received marker for the first time then
6:          suspend the delivery of global transactions  {disable line 14 in Algorithm 2}
7:          for all servers r : send(marker) to r        {send maker to all partitions}
8:  when received marker from all partitions
9:      snapshot contains everything up to SC
10:     resume delivery of global transactions           {enable line 14 in Algorithm 2}
```

3.6 Discussion

Local and global read-only transactions and local update transactions scale linearly in S-DUR with the number of servers. The performance of global transactions depends on how many partitions transactions access. In general, when running global transactions only, we can expect the system to be outperformed by the traditional deferred update replication protocol. Therefore, overall performance will depend on a partitioning of the database that reduces the number of global transactions and the number of partitions accessed by global transactions.

With respect to deferred update replication, our certification condition introduces additional aborts in the termination of global update transactions. Transaction termination in DUR relies on total order: any two conflicting transactions t_i and t_j are delivered and certified in the same order in every server. Thus, it is sufficient to abort one transaction to solve the conflict. Since in S-DUR t_i and t_j can be certified in any order, to avoid inconsistencies, we must conservatively abort both transactions. This is similar to deferred update replication algorithms that rely on atomic commit to terminate transactions [26] [1].

4 Scalable Deferred Update Replication in geo-replicated environments

S-DUR is oblivious to the geographical location of clients and servers. In this section, we revisit our system model considering a geographically distributed environment, discuss possible deployments of S-DUR in these settings, point out performance issues with these deployments, and propose solutions to overcome the problems.

4.1 A system model for geo-replication

We assume client and server processes grouped within *datacenters* (i.e., *sites*) geographically distributed over different *regions*. Processes within the same datacenter and within different datacenters in the same region experience low-latency communication; hereafter denoted as δ (from a fraction of a millisecond to a few milliseconds of roundtrip time). Messages exchanged between processes located in different regions are subject to larger latencies; hereafter denoted as Δ (i.e., roundtrip time of tens to hundreds of milliseconds). A partition replicated entirely in a datacenter can tolerate the crash of some of its replicas. If replicas are located in multiple datacenters within the same region, then the partition can tolerate the crash of a whole site. Finally, catastrophic failures (i.e., the failure of all datacenters within a region) can be addressed with inter-region replication.

Replication across regions is mostly used for locality, since storing data close to the clients avoids large delays due to inter-region communication. We account for client-data proximity by assuming that each database partition p has a preferred server, denoted by $pserver(p)$, among the servers that contain replicas of p. Partition p can be accessed by clients running at any region, but applications can reduce transaction latency by carefully placing the preferred server of a partition in the same region as the partition's main clients.

4.2 Geographically distributed deployments

We now consider two deployments of S-DUR in a geographically distributed system.

The first deployment ("WAN 1" in Figure 2) places a majority of the servers that replicate a partition in the same region, possibly in different datacenters; other servers replicating p are distributed across regions. Having the majority of servers in the same region allows the partition to order messages quickly and therefore terminate local transactions without long delays; global transactions are subject to inter-region delays.

A local transaction executed against the preferred server of partition p_1 (s_1 in the figure) will terminate in 4δ, where δ is the maximum communication delay among servers in the same region. A global transaction that accesses partitions p_1 and p_2, executed against server s_1, will be subject to $4\delta + 2\Delta$, where Δ is the maximum inter-region delay.

The second deployment ("WAN 2") distributes the servers of a partition across regions. This deployment can tolerate catastrophic failures, as we discuss next. The termination of a local transaction will experience $2\delta + 2\Delta$ since Paxos will no longer run among servers in the same region. Global transactions are more expensive than local transactions, requiring $3\delta + 3\Delta$ to terminate. Note that we do not place server s_4 in Region 1 because this would result in Region 2 having no preferred server. Therefore, if a region r contains a server of

p, read operations issued by transactions in execution at r on items in p will not experience long delays. Although this deployment tolerates the failure of an entire region (i.e., a catastrophic failure), it imposes longer delays in the termination of both local and global transactions.

In both deployments, a global transaction that executes at p_1 (respectively, p_2) will read items from p_2 (p_1) within 2δ. Servers in remote regions speed up the execution of global transactions that execute in these regions and read data items in p.

Deployments WAN 1 and WAN 2 tolerate the failure of servers in a partition as long as a majority of the servers is available in the partition. The first deployment, however, does not tolerate the failure of all servers in a region, since such an event would prevent atomic broadcast from terminating in some partitions. For example, in Figure 2, first deployment, if all servers in Region 1 fail, then transactions local to Partition 1 and global transactions that modify Partition 1 will block.

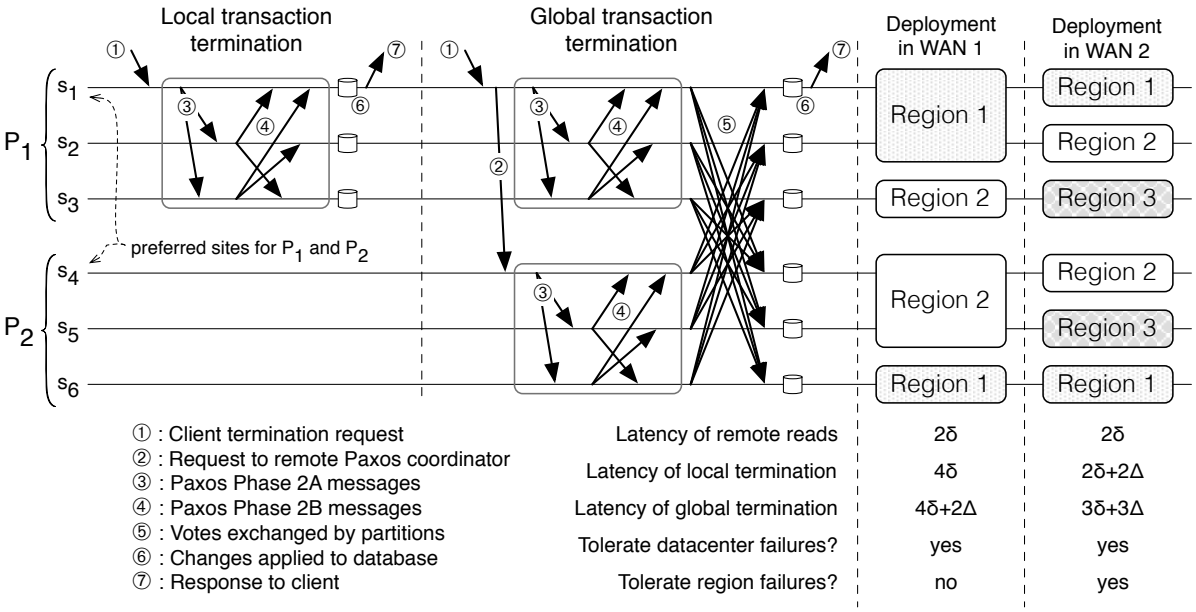

Figure 2: Scalable Deferred Update Replication deployments in a geographically distributed environment, where δ is the maximum communication delay between servers in the same region and Δ is the maximum communication delay across regions; typically $\Delta \gg \delta$. The database contains two partitions, p_1 and p_2, and clients are deployed in the same datacenter as server s_1.

4.3 Performance considerations

S-DUR provides in theory good latency in the geo-replicated settings described above. In practice, we identified the following problem that prevents S-DUR from achieving good performance in wide-area networks. A partition decides the order of a global transaction at delivery time. As a consequence global transactions potentially delay local transaction by up to two inter-region delays. For example, if t_i is delivered before t_j in partition p, t_i will be certified before t_j. If t_i and t_j pass certification (in all concerned partitions), t_i's updates will be applied to the database before t_j's. While this mechanism guarantees deterministic transaction termination, it has the undesirable effect that t_j may have its termination delayed by t_i. This is particularly problematic in S-DUR if t_i is a global transaction and t_j is a local transaction since global transactions may take much longer to terminate than local transactions.

The consequences of global transactions on the latency of local transactions depend on the difference between the expected latency of local and global transactions. For example, in WAN 1 local transactions are expected to terminate much more quickly than global transactions, which is not the case in WAN 2. Thus, global transactions can have a more negative impact on local transactions in WAN 1 than in WAN 2. We have assessed this phenomenon experimentally (details in Section 5) and found that in WAN 1, global transactions can increase the latency of local transactions by up to 18 times. We next discuss three techniques that reduce the effects of global transactions on the latency of local transactions.

4.4 Delaying transactions

In our example in the previous section, if t_j is a local transaction delivered after a global transaction t_i at server s, t_j will only terminate after s has received votes from all partitions in $partitions(t_i)$ and completed t_i.

We can reduce t_i's effects on t_j as follows. When s receives t_i's termination request (message 1 in Figure 2), s forwards t_i to the other partitions (message 2) but delays the broadcast of t_i at p by Δ time units. Delaying the broadcast of t_i in p increases the chances that t_j is delivered before t_i but does not guarantee that t_j will not be delivered after t_i.

Note that if Δ is approximately the time needed to reach a remote partition (message 2 in Figure 2), then delaying the broadcast of t_i at p by Δ will not increase t_i's overall latency.

Algorithm 4 shows the delaying of transactions for a server s in partition p. To delay a transaction, the only part that is affected is when server s receives a request to commit transactions t. In which case s broadcasts t to each one of the partitions involved by t, possibly delaying the broadcast at partition p. In Algorithm 4, function $delay(x, p)$ of line 5 returns the estimated latency between partitions x and p.

ALGORITHM 4: Transaction delaying, server s in partition p

```
1: when receive(commit, t)
2:     let P be partitions(t) \ {p}                    {broadcast t to each...}
3:     for all x ∈ P : abcast(x, t)                    {...remote partition}
4:     Δ ← max({delay(x, p) | x ∈ P})                  {determine maximum delay}
5:     abcast(p, t) after Δ time units                 {delay local broadcast}
```

4.5 Reordering with fixed threshold

The idea behind reordering is to allow a local transaction t_j to be certified and committed before a global transaction t_i even if t_i is delivered before t_j. This is challenging for two reasons: First, when t_j is delivered by some server s in partition p, s may have already sent t_i's vote to other partitions. Thus, reordering t_j before t_i must not invalidate s's vote for t_i. For example, assume t_i reads items x and y and writes item y and s voted to commit t_i. If t_j updates the value of x, then s cannot reorder t_j before t_i since that would change s's vote for t_i from commit to abort. Second, the decision to reorder transactions must be deterministic, that is, if s decides to reorder t_j, then every server in p must reach the same decision.

We ensure that at partition p local transaction t_j can be reordered with previously delivered pending transactions $t_{i_0}, ..., t_{i_M}$ using a reordering condition similar to the one presented in [27], originally devised to reduce the abort rate of concurrent transactions. In our context, we define that t_j can be serialized at position l if the following holds:

(a) $\forall k, 0 \leq k < l$: $writeset(t_{i_k}) \cap readset(t_j) = \emptyset$ and

(b) $\forall k, l \leq k \leq M$: $writeset(t_j) \cap readset(t_{i_k}) = \emptyset$.

If there is a position l that satisfies the constraints above, t_j passes certification and is inserted at position l, which essentially means that it will become the l-th transaction to be applied to the database, after transactions $t_{i_0}, ..., t_{i_{l-1}}$ have completed. If more than one position meets the criteria, servers choose the leftmost position that satisfies the conditions above since that will minimize t_j's delay.

Consider now an execution where t_i is pending at server s when t_j is delivered and let t_i read and write item x and t_j read and write item y. Thus, s can reorder t_j before t_i in order to speed up t_j's termination. At server s', before t_j is delivered s' receives all votes for t_i and commits t_i. The result is that when s' delivers t_j, it will not reorder t_j before t_i since t_i is no longer a pending transaction at s'. Although t_i and t_j modify different data items, servers must commit them in the same order to avoid non-serializable executions. For example, a transaction that reads x and y at s could observe that t_j commits before t_i and another transaction that reads x and y at s' could observe that t_i commits before t_j.

To guarantee deterministic reordering of transactions, we introduce a *reordering threshold* of size k per pending global transaction t_i. Transaction t_i's reordering threshold determines that (a) only local transactions among the next k transactions delivered after t_i can be reordered before t_i; and (b) s can complete t_i only after s receives all votes for t_i and s has delivered k transactions after t_i. In the previous example, if we

set $k = 1$, then server s' would not complete t_i after receiving t_i's votes from other partitions, but would wait for the delivery of t_j and, similarly to server s, s' would reorder t_j and t_i.

Note that we try to reorder local transactions with respect to global transactions only. We found experimentally that reordering local transactions among themselves and global transactions among themselves did not bring any significant benefits. The reordering threshold must be carefully chosen: a value that is too high with respect to the number of local transactions in the workload might introduce unnecessary delays for global transactions. Replicas can change the reordering threshold by broadcasting a new value of k.

Algorithm 5 shows the mechanism in detail. The algorithm overrides functions *readyToCommit*() and *certify*(). In addition, the algorithm keeps *DC*, the number of delivered transactions, which is updated whenever a transactions is certified (line 6)).

Function *certify*() certifies t against transactions that committed after t started (lines 7–8), using the usual certification test performed by function *ctest*(). This check distinguishes between local and global transactions: while a local transaction has its readset compared against the writeset of committed transactions, a global transaction has both its readset and writeset compared against committed transactions (see Section 3 for a description of why this is needed). If some conflict is found, t must abort (line 8); otherwise the check continues.

A local transaction t is reordered among pending transactions (lines 15–24). The idea is to find a position for t in the pending list as close to the beginning of the list as possible, since that would allow t to leap over the maximum number of global transactions (line 15), that satisfies the following constraints: (a) transactions placed in the pending list that will consequently commit before t must not update any items that t reads (line 16); (b) we do not wish to reorder t with other local transactions, and thus, all transactions placed after t in the pending list must be global (line 17); (c) we do not allow a local transaction to leap over a global transaction that has reached its reorder threshold, in order to ensure a deterministic reordering check (line 18); and finally (d) the reordering of t must not invalidate the votes of any previously certified transactions (lines 19–20). If no position satisfies the conditions above, t must abort (line 21); otherwise, s inserts t in the appropriate position in the list of pending transactions (lines 22–23) and t is declared committed (line 24).

A global transaction t is first tagged with a *reordering threshold* (line 10). Transaction t is further checked against all pending transactions (lines 11–13), to avoid non-serializable executions that can happen when transactions are delivered in different orders at different partitions. In the absence of conflicts, t becomes a pending transaction (line 13) and is locally declared as committed (line 24).

A global transaction t that reaches the head of the pending list can only be completed at server s if (a) s received votes from all partitions involved in t and (b) t has reached its reordering threshold (line 4). If these conditions hold, s checks whether all partitions voted to commit t and completes t accordingly. When a global transaction t reaches the head of the pending list, conditions (a) and (b) above will eventually hold provided that all votes for t are received and transactions are constantly delivered, increasing the value of the *DC* counter (line 6). If a server fails while executing the submit procedure for transaction t, then it may happen that some partition p delivers t while some other partition p' will never do so. As a result, servers in p will not complete t since p''s vote for t will be missing. To solve this problem, if a server s in p suspects that t was not broadcast to p', because t's sender failed, s atomically broadcasts a message to p' requesting t to be aborted. Atomic broadcast ensures that all servers in p' deliver first either s''s request to abort or transaction t. Servers in p' will act according to the first message delivered (Section 3.4).

4.6 Reordering with broadcasting of votes

In the reordering mechanism described earlier, each partition relies on an agreed upon reordering threshold to ensure a deterministic decision within a partition. We next describe an alternative mechanism to perform reordering. The idea is to use the atomic broadcast primitive to agree on a deterministic ordering of global transactions within each partition. Upon certification, local transactions either commit or abort right away. A local transaction t commits right away if it passes the usual checks: t does not conflict with concurrent transactions that are already committed, and t does not invalidate votes of pending global transactions; otherwise t aborts. The mechanism performs the same certification checks on global transactions. In addition, the protocol atomically broadcasts the final outcome of global transactions within partitions. Upon delivery, global transactions are committed or aborted according to the final outcome. This ensures a deterministic ordering of global transactions.

Differently from the previous reordering protocol, this mechanism aims at always reordering local trans-

ALGORITHM 5: Certification with fixed threshold, server s in partition p

```
1:  Initialization
2:    DC ← 0                                                    {delivered transaction counter}
3:  function readyToCommit(t)
4:    return t is local ∨ (t.rt = DC ∧ ∀q ∈ partitions(t) : (t.id, q, ⋆) ∈ VOTES)
5:  function certify(t)
6:    DC ← DC + 1                                               {one more transaction delivered}
7:    if ∃t' ∈ DB[t.st[p] ... SC] : ctest(t, t') = false then
8:      return abort                                            {t aborts if conflicts with committed t'}
9:    if t is global then
10:     t.rt ← DC + ReorderThreshold                            {set t's Reorder Threshold}
11:     if ∃t' ∈ PL : ctest(t, t') = false then
12:       return abort                                          {t aborts if conflicts with pending t'}
13:     PL ← PL ⊕ t                                             {append t to pending list if no conflicts}
14:   else
15:     let i be the smallest integer, if any, such that
16:     ∀k < i : PL[k].ws ∩ t.rs = ∅ and                        {t's reads are not stale}
17:     ∀k ≥ i : (PL[k] is global and                           {no leaping local transactions}
18:           PL[k].rt < DC and                                 {no leaping globals after threshold}
19:           t.ws ∩ PL[k].rs = ∅ and                           {previous votes still valid}
20:           t.rs ∩ PL[k].ws = ∅)                              {ditto!}
21:     if no i satisfies the conditions above then return abort
22:     for k from size(PL) downto i do PL[k + 1] ← PL[k]
23:     PL[i] ← t                                               {after making room (above), insert t}
24:   return commit                                             {t is a completed transaction!}
```

actions at the beginning of the list of pending transactions, so that local transactions never wait for a global transaction to finish. The downside of this mechanism is that global transactions become more expensive, in that the termination protocol requires two invocations of the atomic broadcast primitive. With respect to the deployments described in Section 4.2, the termination of global transactions based on broadcasting of votes requires $6\delta + 2\Delta$ in the case of deployment WAN 1, and $3\delta + 5\Delta$ in the case of deployment WAN 2.

Algorithm 6 shows the termination protocol in detail. With respect to Algorithm 2, we override function *certify*() and the handler for receiving votes from other partitions (lines 1–6).

Function *certify*() (lines 10–19) performs similarly as in Algorithm 2, the only significant change is that a committing local transaction is committed immediately, and is never included in the pending list.

We next describe how the protocol orders global transactions. Whenever the server receives a vote from partition p, the vote is included in the set of votes (line 2). The server proceeds by checking whether enough votes for the transaction have been received by invoking function *readyToCommit*() (line 4, unmodified from Algorithm 2). If so, the server broadcasts the transaction identifier and its final outcome within the partition (lines 5–6). When the final outcome is delivered, the server commits or aborts the transaction according to the delivered final outcome (lines 7–9).

5 Performance Evaluation

In the following, we assess the performance of transaction delaying and reordering in two geographically distributed environments. We compare throughput and latency of the system with and without the proposed techniques.

5.1 Setup and benchmarks

We ran the experiments using Amazon's EC2 infrastructure. We used r3.large instances equipped with two virtual cores and 15 GB of RAM. We deployed servers in three different regions: Ireland (EU), N. Virginia (US-EAST), and Oregon (US-WEST). Using `ping`, we observed the following inter-region round-trip latencies: (a) ≈100 milliseconds between US-EAST and US-WEST, (b) ≈90 milliseconds between US-EAST and EU, and (c) ≈170 milliseconds between US-WEST and EU.

ALGORITHM 6: Certification with broadcasting of votes, server s in partition p

```
1:  when receive(tid, v) from partition q
2:      VOTES ← VOTES ∪ (tid, q, v)                          {one more vote for tid}
3:      let t be the transaction in PL such that t.id = tid
4:      if readyToCommit(t) then
5:          outcome ← (t.id, ⋆, abort) ∈ VOTES               {one abort vote and t will be aborted}
6:          abcast(t.id, outcome) to partition p
7:  when adeliver(tid, outcome)
8:      let t be transaction in PL such that t.id = tid
9:      complete(t, outcome)
10: function certify(t)
11:     if ∃t' ∈ DB[t.st[p] ... SC] : ctest(t, t') = false then
12:         return abort                                     {t aborts if it conflicts with committed t'}
13:     if ∃t' ∈ PL : (t.rs ∩ t'.ws ≠ ∅) ∨ (t.ws ∩ t'.rs ≠ ∅) then
14:         return abort                                     {t aborts if it conflicts with pending t'}
15:     if t is global then
16:         PL ← PL ⊕ t                                      {append t to pending list if no conflicts}
17:     else
18:         complete(t, commit)                              {commit local transactions right away}
19:     return commit
```

In the experiments we used two partitions, each composed of three servers. For WAN 1, we deployed the partitions as follows: the first partition has a majority of nodes in EU, while the second partition has a majority of nodes in US-EAST. For WAN 2, we deployed the partitions such that each one has one server in EU, one in US-EAST, and one in US-WEST; to form a majority, partitions are forced to communicate across regions. In any case, servers deployed in the same region run in different availability zones.

We present results for two different workloads: a microbenchmark and a Twitter-like social network application. In the microbenchmark, clients perform transactions that update two different objects (two read and two write operations). In the experiments, we vary the percentage of global transactions in which case a transaction updates one local object and one remote object. We use one million data items per partition, where each data item is a 4-byte integer.

The Twitter-like benchmark implements the operations of a social network application in which users can: (1) follow another user; (2) post a new message; and (3) retrieve their timeline containing the messages of users they follow. We implemented this benchmark as follows. Users have a unique id. For each user u we keep track of: (1) a list of "consumers" containing user ids that follow u; and (2) a list of "producers" containing user ids that u follows; and (3) u's list of posts. In the experiments, we partitioned the data by users (i.e., a user, its posts, its producers and consumers lists are stored in the same partition).

Post transactions append a new message to the list of posts. Given the above partitioning, post transactions are all local transactions. Follow transactions update two lists, a consumer list and a producer list of two different users. Follow transactions can be either local or global, depending on the partitions in which the two users are stored. Timeline transactions build a timeline of user u by merging together the posts of the users u follows. Timeline is a global read-only transaction.

In the experiments, we populate two partitions, each storing 100 thousand users. We report results for a mix of 85% timeline, 7.5% post and 7.5% follow transactions. Follow transactions are global with 50% probability.

We report throughput and latency corresponding to 70% of the maximum performance, for both benchmarks. In all experiments, we generate one new snapshot every second, using the mechanism described in Section 3.5, unless stated otherwise. Each experiment lasts 80 seconds; we discard 10 seconds from the beginning and the end of the execution, leaving 60 seconds worth of traces per experiment.

5.2 Baseline - Throughput

In the following experiment we report the baseline throughput of S-DUR and compare it against results we obtained from Berkeley DB (BDB), a widely used open-source embedded database. The experiment was performed on a local cluster. We setup BDB to use the in-memory B-Tree access method with transactions en-

abled. We spawn several client threads issuing RPC requests to the server that embeds BDB. We used Apache Thrift to implement the multithreaded RPC server and clients. For comparison, we setup S-DUR to work with one partition only (3 servers in total). To make the comparison fair, client threads issue RPC requests to one S-DUR server only. Figure 3 shows that the throughput of a single partition S-DUR and BDB are comparable. The write throughput of S-DUR is lower than that of BDB due to S-DUR's termination protocol, which requires an invocation of Paxos. In this experiment, we found the bottleneck to be in our Paxos implementation. Read throughput of S-DUR is also lower than that of BDB, even though read-only in S-DUR transactions do not invoke Paxos. We attribute this to the fact that S-DUR's implementation is single threaded, whereas BDB exploits multiple threads to run transactions simultaneously.

System	Read throughput	Write throughput
S-DUR	58 Ktps	32 Ktps
Berkeley DB	89 Ktps	42 Ktps

Figure 3: Baseline read and write transaction throughput

5.3 Baseline - Latency

We deployed S-DUR in a geographically distributed environment following the two alternatives (WAN 1 and WAN 2) discussed in Section 4.2. Figure 4 shows the throughput and latency for both WAN 1 and WAN 2 deployments with workload mixes containing 0%, 1%, 10% and 50% of global transactions. Latency values correspond to their 99-th percentile and average. Figure 4 also shows the cumulative distribution function (CDF) of latency for 0% and 1% of global transactions.

Global transactions have a clear impact on the system's throughput; as expected the phenomenon is more pronounced in WAN 1 than in WAN 2 (see Section 4.3). In the absence of global transactions, local transactions can execute within 10 milliseconds in WAN 1. The latency of locals increases to 160 milliseconds with 1% of global transactions, a 16x increase. We observed that in workloads with 10% and 50% of global transactions, latency of locals increase to 173 milliseconds (17x increase to the 0% configuration) and 184 milliseconds (18x increase), respectively. This shows that the convoy phenomena deteriorates as we increase the fraction of global transactions.

In WAN 2, local transactions alone experienced a latency of 141 milliseconds, while in workload mixes of 1%, 10% and 50% of global transactions latency increased to 151 milliseconds (1.07x), 181 milliseconds (1.34x) and 179 milliseconds (1.27x), respectively.

The CDFs of Figure 4 show that in workloads with global transactions, the distribution of latency of local transactions follows a similar shape as the latency distribution of global transactions, showing the effect of global on local transactions (see Section 4.3).

5.4 Delaying transactions

We now assess the transaction delaying technique in the WAN 1 deployment. In these experiments, we tested various delay values while controlling the load to keep the throughput of local transactions among the various configurations approximately constant. Figure 5 shows that the technique has a positive effect in all percentages of global transactions we considered. Delaying globals by 40 milliseconds resulted in a reduction in the 99-th percentile latency of 30, 30.8, and 36 milliseconds for 1%, 10%, and 50% of global transactions respectively. Moreover, we noticed no impact on the overall latency of global transactions.

The 40-millisecond delay parameter was chosen such that it approximately matches the latency for sending one message from one partition to the other. As expected, increasing the delay further (e.g., 80 milliseconds) yields no significant improvement on local transactions, while we noticed a significant impact on the latency of global transactions.

5.5 Reordering transactions

Figure 6 show the effects of reordering in the latency of local and global transactions under various workloads in WAN 1. We assess different reordering thresholds in configurations subject to a similar throughput. We compare the various reordering thresholds with reordering based on the broadcasting of votes.

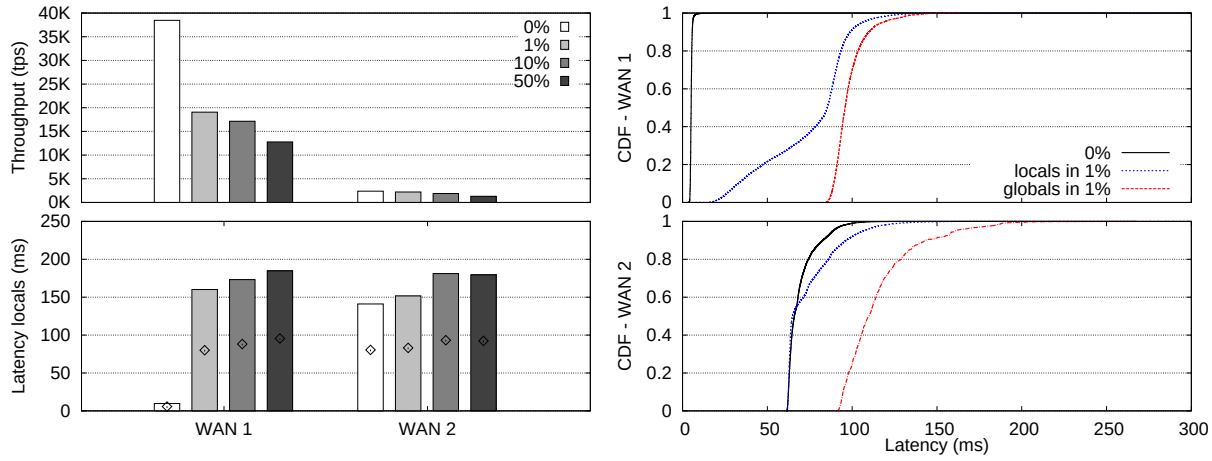

Figure 4: S-DUR's local and global transactions in two geographically distributed deployments (WAN 1 and WAN 2). Throughput in transactions per second (tps), latency 99-th percentile (bars) and average latency (diamonds in bars) in milliseconds (ms).

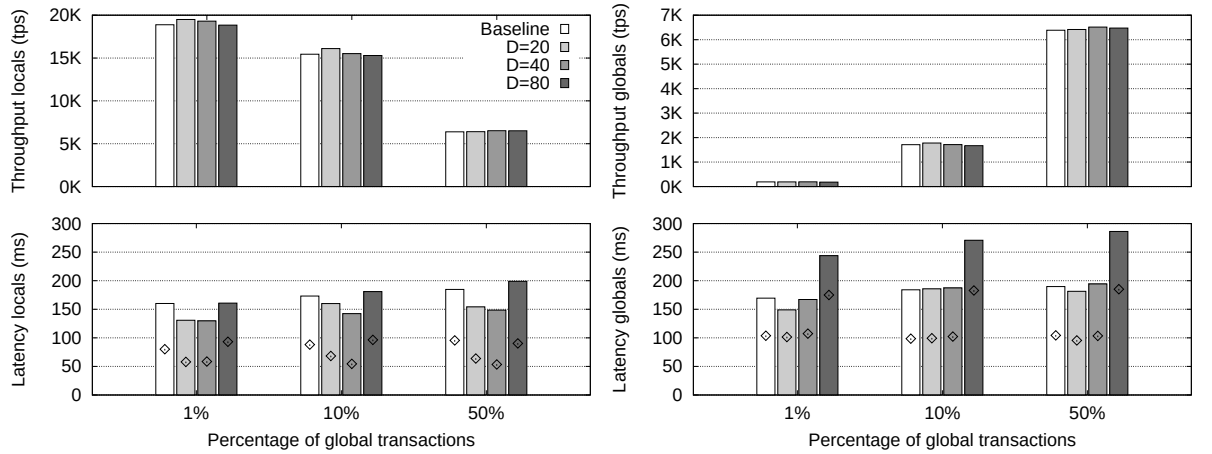

Figure 5: Throughput and latency of local and global transactions with delayed transactions in WAN 1.

In WAN 1, reordering has a positive impact on local transactions for all three workload mixes). For example, for 1% global transactions, a reordering threshold of 640 reduces the 99-th percentile latency of local transactions from 160 ms (in baseline) to 114 milliseconds, a 29% improvement. For mixes with 10% and 50% of global transactions the improvement is 24% and 30% respectively. The best results are achieved with reordering with broadcasting of votes, where the improvement is always greater than 90%, regardless of workload mix. We observed no significant impact on the 99-th percentile latency of the corresponding global transactions, for both reordering techniques.

Local transactions in WAN 2 (Figure 7) also benefit from reordering. Although, there is a tradeoff between the latency of locals and globals, an effect not seen for WAN 1. For example, in the workload with 10% of global transactions, a reordering threshold of 80 reduced the 99-th percentile latency of local transactions from 181 milliseconds (in baseline) to 130 milliseconds. The corresponding latency of global transactions increased from 211 milliseconds to 238 milliseconds. Reordering with broadcasting of votes achieves the best results for local transactions, though it also experiences the worst latency for global transactions. This is not surprising, as broadcasting the votes is expensive across geographical regions (atomic broadcast requires two more communication steps). Similar trends are seen for workloads with 1% and 50% of global transactions.

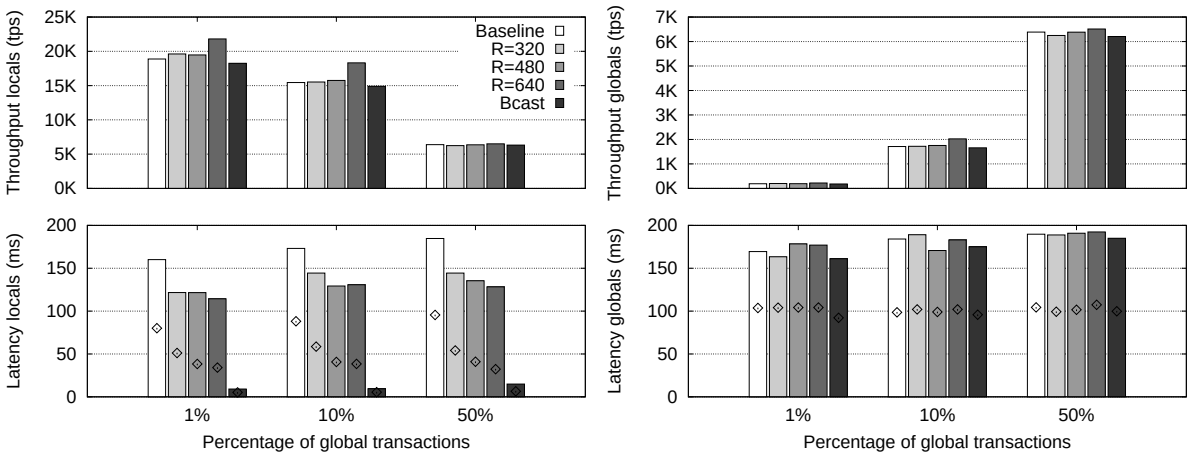

Figure 6: Throughput and latency of local and global transactions with reordering in WAN 1.

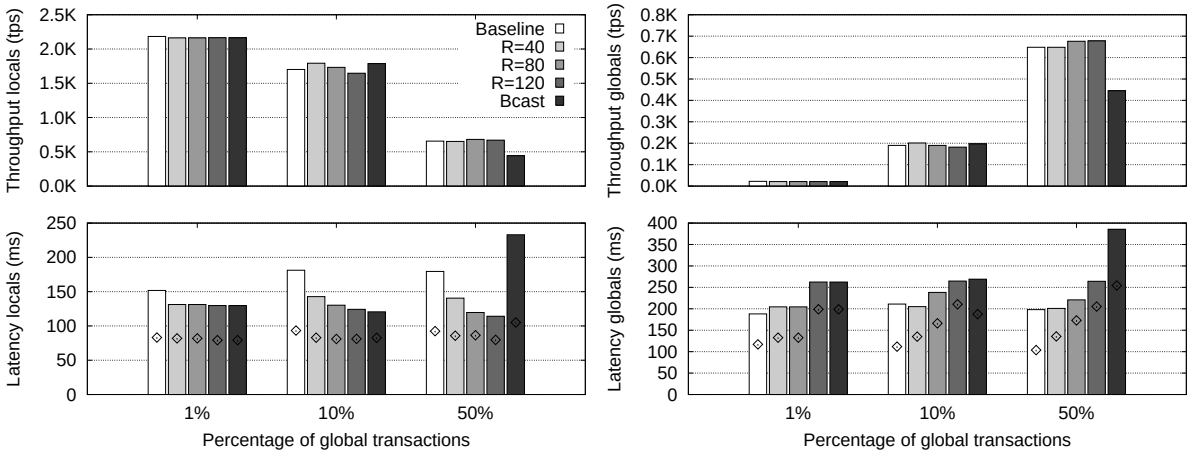

Figure 7: Throughput and latency of local and global transactions with reordering in WAN 2.

5.6 Social network application

We next discuss the effects of reordering in our social network application.

In WAN 1 (Figure 8 left), reordering with broadcasting of votes performs best. It achieves the best improvement on latency of local Post and Follow transactions (88% and 85% respectively), with minimal impact on the latency of global Follow transactions (less than 5%).

In WAN 2 (Figure 8 right), reordering with broadcasting of votes achieves the best latency for local Post and Follow transactions, although it is slower in committing global Follow transactions. As explained in the previous experiment, reordering with broadcasting of votes requires an additional invocation of atomic broadcast across wide-area links. In this deployment, reordering with a fixed threshold of size 60, achieves 40% improvement for local Post and Follow transactions, and only 7% increased latency on global Follow transactions.

In this experiment, Timeline transactions use the snapshot created with the snapshot mechanism described in Section 3.5. Timeline transactions perform equally well using both techniques: both reordering mechanisms have no impact on read-only transactions. The latency of Timeline transaction is low, even in WAN 2, because the snapshotting algorithm allows transactions to read data items from the closest replica.

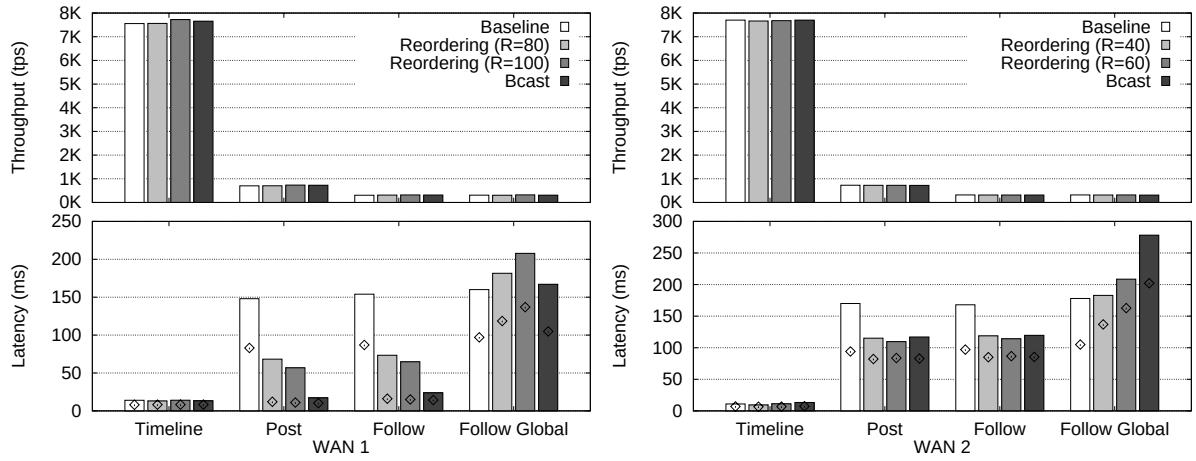

Figure 8: Throughput of social network application in WAN 1 (left) and WAN 2 (right).

5.7 Impact of Snapshots

We next assess the impact of the snapshot mechanism described in Section 3.5. We consider average and 99-th percentile latency while varying the think time between invocations of the snapshot algorithm. Figure 9 shows the results for WAN 1 and WAN 2.

The latency of local transactions grows steadily (with or without broadcasting of votes) as we decrease the snapshot interval. We notice that the average grows slower than the 99-th percentile, suggesting that the snapshot algorithm impacts only few, unlucky, transactions. In the case of reordering with broadcasting of votes, local transaction are not affected by the snapshot algorithm. This is due to the fact that local transactions never wait for global transactions to finish. In the case of WAN 1, reordering with broadcast is advantageous, in that local transactions experience no impact due to global transactions or the snapshot mechanism. While the commitment of a global transaction takes the same time for both techniques in the worst case (i.e., when there is no think time between invocations of the snapshot algorithm).

In WAN 2, global transactions are more expensive with the reordering with broadcasting of votes, and therefore reordering with a carefully selected threshold performs better than broadcasting votes.

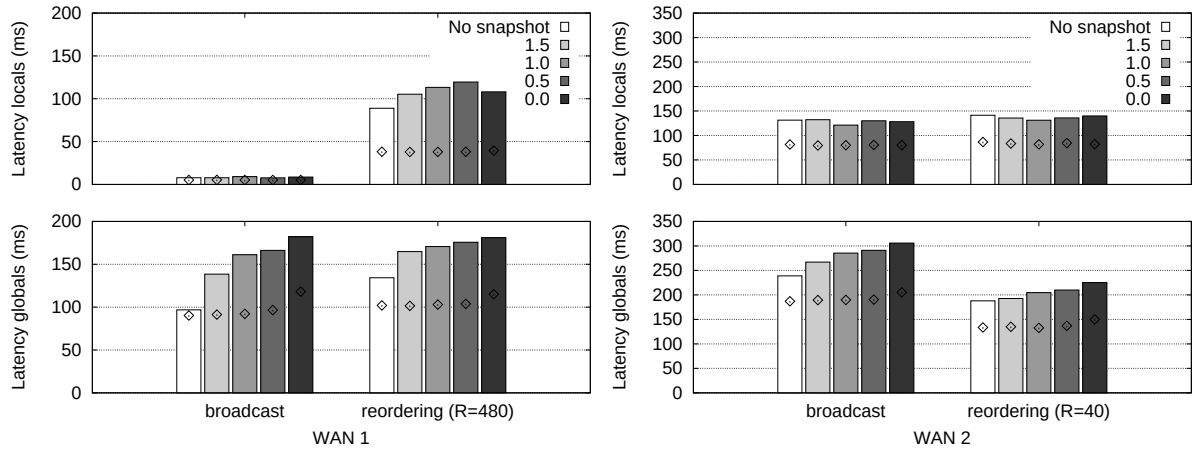

Figure 9: Impact of snapshots in WAN 1 (left) and WAN 2 (right).

6 Related work

Several protocols for deferred update replication where servers keep a full copy of the database exist (e.g., [27], [16], [21], [24], [1]). In [32] it is suggested that the scalability of these protocols is inherently limited by the number of transactions that can be ordered, or by the number of transactions that a single server can certify and apply to the local database.

Many storage and transactional systems have been proposed recently. Some of these systems (e.g., Cassandra [19], Dynamo [13], Voldemort[3]) guarantee *eventual consistency*, where operations are never aborted but isolation is not guaranteed. Eventual consistency allows replicas to diverge in the case of network partitions, with the advantage that the system is always available. However, clients are exposed to conflicts and reconciliation must be handled at the application level.

Spinnaker [29] is similar to the approach presented here in that it also use several instances of Paxos to achieve scalability. However, Spinnaker does not support transactions across multiple Paxos instances.

Differently from previous works, Sinfonia [2] offers stronger guarantees by means of minitransactions on unstructured data. Similarly to S-DUR, minitransactions are certified upon commit. Differently from S-DUR, both update and read-only transactions must be certified in Sinfonia, and therefore can abort. Read-only transactions do not abort in S-DUR.

COPS [22] and Eiger [23] are both storage systems that ensure a stronger version of causal consistency, which in addition to ordering causally related write operations also orders writes to the same data items. COPS provides read-only transactions, but does not provide multi-key update transactions. While Eiger provides multi-key update transactions. Both systems do not allow general read and write transactions and require full replication at all datacenters.

Walter [35] offers an isolation property called Parallel Snapshot Isolation (PSI) for databases replicated across multiple data centers. PSI guarantees snapshot isolation and total order of updates within a site, but only causal ordering across data centers.

PSI can be further weakened to a consistency criterion called Non-monotonic Snapshot Isolation (NMSI) [3]. Differently from PSI, NMSI allows a transaction to observe a snapshot that was committed after its start. While PSI assumes full replication, NMSI assumes partial replication [3].

Vivace [9] is a storage system optimized for latency in wide-area networks. Vivace's replication protocol prioritizes small critical data exchanged between sites to reduce delays due to congestion. Vivace does not provide transactions over multiple keys.

Google's Bigtable [8] and Yahoo's Pnuts [10] are distributed databases that offer a simple relational model (e.g., no joins). Bigtable supports very large tables and copes with workloads that range from throughput-oriented batch processing to latency-sensitive applications. Pnuts provides a richer relational model than Bigtable: it supports high-level constructs such as range queries with predicates, secondary indexes, materialized views, and the ability to create multiple tables.

None of the above systems provides strongly consistent execution for multi-partition transactions over WANs. Among the ones that offer guarantees closer to S-DUR.

P-Store [31] is the closest to our work in that it implements deferred update replication optimized for wide-area networks. Unlike S-DUR, P-Store uses genuine atomic multicast to terminate transactions, which is more expensive than atomic broadcast. P-Store also avoids the *convoy* effect in that it can terminate transactions in parallel. S-DUR can also terminate transactions in parallel, and in addition to that we use reordering to further reduce delays.

Spanner [11] is a distributed database for WANs. Like S-DUR the database is partitioned and replicated over several Paxos instances. Spanner uses a combination of two-phase commit and a TrueTime API to achieve consistent multi-partitions transactions. TrueTime uses hardware clocks to derive bounds on clock uncertainty, and is used for assigning globally valid timestamps and for consistent reads across partitions.

Clock-SI [15] is similar to Spanner in that it also uses physical clocks to order transactions. Unlike Spanner, Clock-SI relies on loosely synchronized clocks, but does not provide replication.

MDDC [17] is a replicated transactional data store that also uses several instances of Paxos. MDCC optimizes for commutative transactions, and uses Generalized Paxos which allows to relax the order of transaction delivery of commuting transactions.

Recently, a framework called G-DUR [4] has been proposed for developing and comparing protocols that use the deferred update replication technique. This work includes a comparison of S-DUR [32] (without the reordering techniques of Sections 4.5 and 4.6), P-Store [31], Serrano [33], Walter [35], NMSI [4], and GMU [28].

[3] http://project-voldemort.com

Their results confirm our considerations made in this thesis: protocols based on snapshot isolation (such as [33]) are not necessarily more efficient than protocols that provide serializability (such as S-DUR). However, weaker forms of Snapshot Isolation (such as PSI [35] and NMSI [3]) provide substantial gains. The use of atomic multicast in systems such as P-Store [31] is more expensive than the separate and independent atomic broadcasts of S-DUR. Slightly weakening read-only transactions with update serializability, as in GMU [28], also allows for substantial gains.

Our reordering technique with fixed threshold (Section 4.5) is based on the algorithm described in [27], originally designed for reducing the abort rate. In our context, we extend the idea to avoid the delay imposed by global communication on local transactions.

The transactions delaying technique of Section 4.4 increases the likelyhood that transactions are delivered approximately at the same time in every partition. A similar problem has been explored in the context of Optimistic Atomic Broadcast protocols. These protocols take advantage of the spontaneous order in which networks deliver messages so that the computation is overlapped with the final total order delivery. While spontaneous ordering is highly probable in local-area networks, it is not tipically the case in wide-area networks. Spontaneous order can be achieved by injecting artificial delays [34] [30] or by timestamping messages (using loosely synchronized clocks) to optimistically deliver in timestamp order [6].

This paper does not investigate any partitioning techniques. In the experiments, we generally used simple range partitioning. However, the assignment of data items to S-DUR partitions, together with the workload, determines the number of transactions involved in two or more partitions. Devising partitioning techniques is fundamental to achieve scalable performance. Several partitioning techniques have been devised which could be used in S-DUR (e.g., [12] [25] [36]).

7 Conclusions

This paper discusses scalable deferred update replication in geographically distributed settings. S-DUR scales deferred update replication, a well-established approach used in several database replicated systems, by means of data partitioning. S-DUR distinguishes between fast local transactions and slower global transactions. Although local transactions scale linearly with the number of partitions (under certain workloads), when deployed in a geographically distributed environment they may be significantly delayed by the much slower global transactions—in some settings global transactions can slow down local transactions by a factor of more than 10. We presented two techniques that account for this limitation: Transaction delaying is simple, however, produces limited improvements; reordering, a more sophisticated approach, provides considerable reduction in the latency of local transactions, mainly in deployments where global transactions harm local transactions the most. Our claims are substantiated with a series of microbenchmarks and a Twitter-like social network application.

APPENDIX

A Read-only snapshots

We first show that a snapshot S built by our algorithm can be serialized with committed update transactions. Since committed update transactions are serializable, they can be organized as a sequence $H = T_1; T_2; ...$ and there is some l such that S succeeds all transactions $T_k, k \leq l$ and S precedes all transactions $T_k, k > l$, as we show next.

Define S as an n-tuple $S = \langle SC_1, SC_2, \ldots, SC_n \rangle$, where n is the number of partitions, and SC_i is the count of transactions that were committed at partition i. We define transaction T as an n-tuple $T = \langle SC_1, SC_2, \ldots, SC_n \rangle$, where SC_i is the count created by the commit of T, if T modified partition i, and $\perp$ if T did not modify any items in partition i. S succeeds T, denoted $T \rightarrow S$, if for all i, $S[i] \geq T[i]$ or $T[i] = \perp$; S precedes T, denoted $S \rightarrow T$, if for all i, $S[i] < T[i]$ or $T[i] = \perp$.

Therefore, S can be serialized at position l in H if for each $T_k, k \leq l$, $T_k \rightarrow S$ and for each $T_k, k > l$, $S \rightarrow T_k$, which is what our algorithm ensures. For a contradiction, assume there is some transaction T that neither succeeds nor precedes S, that is, there are i and j such that (a) $T[i] \leq S[i]$ and $T[i] \neq \perp$; and (b) $S[j] < T[j]$ and $T[j] \neq \perp$ and.

From (a), the commit of T at partition i precedes the snapshot of S at i. In other words, processes in i received votes for T before receiving the last snapshot marker needed to create S. For case (b), we distinguish

two cases:

- (b.1) Processes in partition j delivered T after receiving the last snapshot marker that created S. Thus, processes in j send S's snapshot marker before sending T's vote, but from (a), processes in i received T's vote from j before the marker from j, which contradicts the property of FIFO channels.
- (b.2) Processes in j delivered T before receiving the last snapshot marker that created S. In this case, either j delivers and commits T before receiving the first marker, in which case $S[j] \geq T[j]$. Or j received the first snapshot marker before delivering T. Since the algorithm suspends the delivery of new transactions until a snapshot is found, we contradict again the fact that processes in i received T's vote from j before the marker from j.

We now show that creating snapshots that can be serialized with committed update transactions is not enough to guarantee serializable executions that combine update and read-only transactions.

Suppose two partitions i and j initiate the snapshot algorithm at about the same time. To distinguish the markers for the snapshot created at i and j, let m_i respectively m_j be their markers. Consider the following sequence of events. Partition i sends snapshot marker m_i, receives snapshot marker m_j from j, delivers a local transaction T_i, and receives snapshot marker m_i from j. Similarly partition j snapshot marker m_j, receives snapshot marker m_i from i, delivers a local transaction T_j, and receives snapshot marker m_j from i. Thus the snapshot initiated at i includes T_i but not T_j, and viceversa the snapshot initiated at j includes T_j but not T_i. Which is not serializable. To avoid this problem, we have to ensure that only one instance of the snapshot algorithm is active at any time.

B Delaying transactions

Delaying the broadcast of a global transaction t in a partition may delay the delivery of t at p but this does not change the correctness of the protocol. To see why, notice that since we assume an asynchronous system, even if t is broadcast to all partitions at the same time, it may be that due to network delays t is delivered at any arbitrary time in the future.

C Reordering with fixed threshold

Consider a local transaction t, delivered after global transaction t' at partition p. We claim that (a) if server s in p reorders t and t', then every correct server s' in p also reorders t and t'; and (b) the reordering of t and t' does not violate serializability.

For claim (a) above, from Algorithm 5, the reordering of a local transaction t (lines 5 – 24) is a deterministic procedure that depends on $DB[t.st[p]\ldots SC]$ (line 7), PL (lines 16 – 20), and DC (line 18). We show next that DB, PL, SC and DC are only modified based on delivered transactions, which suffices to substantiate claim (a) since every server in p delivers transactions in the same order, from the total order property of atomic broadcast.

For an argument by induction, assume that up to the first i delivered transactions, DB, PL, SC and DC are the same at every correct server in p (inductive hypothesis), and let t be the $(i+1)$-th delivered transaction (line 5). PL is possibly modified in function *certify*() (line 23) and from the discussion above depends on DB, PL, SC and DC, which together with the induction hypothesis we conclude that it happens deterministically. DB, PL and SC are also possibly modified in the *complete*() procedure (Algorithm 2, lines 24 – 29), called (i) after t is delivered (Algorithm 2, line 17), (ii) when the head of PL is a local transaction (Algorithm 2, line 20), and (iii) when the head of PL is a global transaction u (Algorithm 2, line 20).

In cases (i) and (ii), since all modifications depend on t, PL and SC, from a similar reasoning as above we conclude that the changes are deterministic. In case (iii), the calling of the *complete*() procedure depends on receiving all votes for t and t having reached its reorder threshold (line 4). From the induction hypothesis, all servers agree on the value of DC. Different servers in p may receive u's votes at different times but we will show that any two servers s and s' will nevertheless reorder t in the same way. Assume that when s assesses u it already received all u's votes and proceeds to complete u before it tries to reorder t. Another server s' assesses u when it has not received all votes and does not call the complete procedure. Thus, s will not reorder t with respect to u. For a contradiction, assume that s' reorders t and u. From the reorder condition, it follows that u has not reached its reorder threshold at s', which leads to a contradiction since u

has reached its threshold at s, from the algorithm (line 6) *DC* depends only on delivered messages and from atomic broadcast all servers deliver the same transactions in the same order.

Finally, claim (b), to see that reordering transactions does not violate serializability, note that the condition for local transaction t to be placed before global transaction t' is that both transactions would be committed if t had been delivered before t'. Since t' passes certification, its readset and writeset do not intersect the readsets and writesets of concurrent transactions delivered before. Thus, in order for t to be reordered before t', t's readset and writeset must not intersect t''s readset and writeset (lines 19 – 20). Moreover, t's readset must not intersect the writeset of any concurrent transaction delivered before t (lines 5 and 16), which is essentially the certification test for local transactions in S-DUR.

D Reordering with broadcasting of votes

Consider a local transaction t, delivered after global transaction t' at partition p. We claim that (a) if server s in p reorders t and t', then every correct server s' in p also reorders t and t'; and (b) the reordering of t and t' does not violate serializability.

For claim (a) above, from Algorithm 6, the reordering of a local transaction t (lines 10 – 19) is a deterministic procedure that depends on $DB[t.st[p]...SC]$ (line 11) and *PL* (line 13). We show next that *DB*, *PL* and *SC* are only modified based on delivered transactions, which suffices to substantiate claim (a) since every server in p delivers transactions in the same order, from the total order property of atomic broadcast.

For an argument by induction, assume that up to the first i delivered transactions, *DB*, *PL* and *SC* are the same at every correct server in p (inductive hypothesis), and let t be the $(i+1)$-th delivered transaction (line 10). *PL* is possibly modified in function *certify*() (line 16) and from the discussion above depends on *DB*, *PL*, *SC* and *DC*, which together with the induction hypothesis we conclude that it happens deterministically. *DB*, *PL* and *SC* are also possibly modified in function *complete*() (Algorithm 2, lines 24 – 29), called (i) after t is delivered (Algorithm 2, line 17), (ii) when t is committed in function *certify*() (line 10), and (iii) when s delivers a final vote for global transaction u (line 7).

In cases (i) and (ii), since all modifications depend on t, *DB PL* and *SC*, from a similar reasoning as above we conclude that the changes are deterministic. In case (iii), the calling of function *complete*() depends on the server in p receiving all votes for t and broadcasting u's final vote. From the total order property of atomic broadcast, all servers in partition p will deliver u's final vote in total order, and call *complete*() for transaction u in the same order.

Finally, claim (b), to see that reordering transactions based on broadcasting of votes does not violate serializability, note that the condition for local transaction t to be placed before global transaction t' is that both transactions would be committed if t had been delivered before t'. Since t' passes certification, its readset and writeset do not intersect the readsets and writesets of concurrent transactions delivered before. Thus, in order for t to be reordered before t', t's readset and writeset must not intersect t''s readset and writeset (line 13). Moreover, t's readset must not intersect the writeset of any concurrent transaction delivered before t (line 11), which is essentially the certification test for local transactions in S-DUR.

References

[1] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *Euro-Par'97 Parallel Processing*, pages 496–503. Springer, 1997.

[2] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[3] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 163–172, Washington, DC, USA, 2013. IEEE Computer Society.

[4] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-dur: A middleware for assembling, analyzing, and improving transactional protocols. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 13–24, New York, NY, USA, 2014. ACM.

[5] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[6] Carlos Eduardo Bezerra, Fernando Pedone, Benoît Garbinato, and Cláudio Geyer. Optimistic atomic multicast. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 380–389. IEEE, 2013.

[7] K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[9] Brian Cho and Marcos K Aguilera. Surviving congestion in geo-distributed storage systems. In *USENIX Annual Technical Conference*, pages 439–451, 2012.

[10] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[12] Carlo Curino, Evan Jones, Yang Zhang, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[14] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[15] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society.

[16] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 134–143, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[17] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[18] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[19] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[20] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[21] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 419–430, New York, NY, USA, 2005. ACM.

[22] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[23] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[24] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, November 2005.

[25] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[26] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, pages 513–520, London, UK, UK, 1998. Springer-Verlag.

[27] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[28] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 455–465, Washington, DC, USA, 2012. IEEE Computer Society.

[29] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

[30] Luís Rodrigues, José Mocito, and Nuno Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 723–727. ACM, 2006.

[31] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine partial replication in wide area networks. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 214–224. IEEE, 2010.

[32] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[33] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.

[34] António Sousa, José Pereira, Francisco Moura, and Rui Oliveira. Optimistic total order in wide area networks. In *21st IEEE Symposium on Reliable Distributed Systems.*, SRDS '02, pages 190–199. IEEE, 2002.

[35] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[36] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J ElmoreA, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.