Geo-replicated storage with scalable deferred update replication

Daniele Sciascia
University of Lugano (USI)
Switzerland

Fernando Pedone
University of Lugano (USI)
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.**

Keywords-**Database replication, scalable data store, fault tolerance, high performance, transactional systems**

I. 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., [1], [2], [3], [4]). The idea behind a scalable deferred update replication (SDUR) protocol is conceptually simple: the database is divided into partitions and each partition is fully replicated by a group of servers [5]. 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 [6]. 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 partitions it is committed; otherwise it is aborted.

Scalable deferred update replication offers very good performance, which under certain workloads grows proportionally with the number of database partitions [5], but it is oblivious to the geographical location of clients and servers. While the actual location of clients and servers is irrelevant for the correctness of SDUR, it has important consequences on the latency perceived by the clients. SDUR 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

This work was supported in part by the Swiss National Science Foundation under grant number 127352.

global transactions in the workload is enough to increase the average latency of local transactions by up to 10 times.

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 two 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 II presents our system model and some definitions. Section III recalls the scalable deferred update replication approach. Section IV discusses how to deploy SDUR in a geographically distributed system, points out performance issues with these deployments, and details solutions to the problems. Section V describes our prototype and some optimizations. Section VI evaluates the performance of the protocol under different conditions. Section VII reviews related work and Section VIII concludes the paper.

II. System model and definitions

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

A. 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-stop failure model (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 [7], we use Paxos to implement atomic broadcast within a group of servers [8]. Paxos requires a majority of correct servers within a group and additional assumptions to ensure liveness, notably a leader-election oracle at each group [8].

B. 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 [9].

III. Scalable Deferred Update Replication

Scalable deferred update replication (SDUR) is an extension of deferred update replication that accounts for partitioned data. In this section, we recall how SDUR works.

A. Transaction execution

In SDUR, 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*

[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.

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.

B. 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 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, SDUR 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.

IV. Scalable Deferred Update Replication in geo-replicated environments

SDUR 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 SDUR in these settings, point out performance issues with these deployments, and propose solutions to overcome the problems.

A. 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. Messages exchanged between processes located in different regions are subject to larger latencies. 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 [10], [11]. We account for client-data proximity by assuming that each database partition p has a preferred server [10], 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.

B. SDUR in geographically distributed systems

We now consider two deployments of SDUR in a geographically distributed system. The first deployment ("WAN 1" in Figure 1) places a majority of the servers that replicate a partition in the same region, possibly in different datacenters. A local transaction executed against the preferred server

[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.

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 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.[3] In both deployments, a global transaction that executes at P_1 (respectively, P_2) will read items from P_2 (P_1) within 2δ.

Deployments one and two tolerate the failure of servers in a partition as long as a majority of the servers is available in the partition (see Section II). 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.

C. Performance considerations

In SDUR, terminating transactions are totally ordered within a partition. 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 SDUR 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 VI) and found that in WAN 1, global transactions can increase the latency of local transactions by up to 10 times. In the next section, we discuss two techniques that reduce the effects of global transactions on the latency of local transactions in SDUR.

D. 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 $\mathit{partitions}(t_i)$ 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 1), 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 1), then delaying the broadcast of t_i at p by Δ will not increase t_i's overall latency.

E. Reordering transactions

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

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

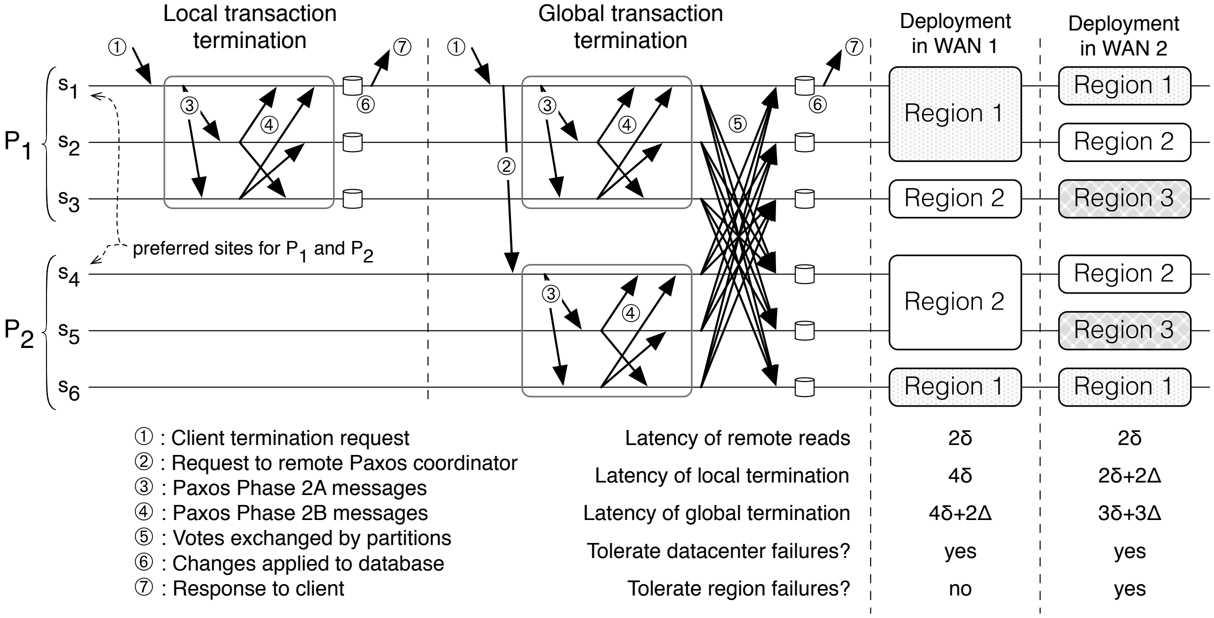
If there is a position l that satisfies the constraints above, t_j passes certification and is "inserted" at position l, which essentially means that it will become the l-th transaction to be applied to the database, after transactions $t_{i_0}, ..., t_{i_{l-1}}$ have completed. If more than one position 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.[4]

To guarantee deterministic reordering of transactions, we introduce a *reordering threshold* of size k per pending

[3] Note that we do not place server s_4 in Region 1 because this would result in Region 2 having no preferred server.

[4] 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.


Local transaction termination
Global transaction termination
Deployment in WAN 1
Deployment in WAN 2
P1
P2
s1
s2
s3
s4
s5
s6
preferred sites for P1 and P2
①
②
③
④
⑤
⑥
⑦
Region 1
Region 2
Region 1
Region 2
Region 3
Region 2
Region 3
Region 1
① : Client termination request
② : Request to remote Paxos coordinator
③ : Paxos Phase 2A messages
④ : Paxos Phase 2B messages
⑤ : Votes exchanged by partitions
⑥ : Changes applied to database
⑦ : Response to client

	Deployment in WAN 1	Deployment in WAN 2
Latency of remote reads	2δ	2δ
Latency of local termination	4δ	2δ+2Δ
Latency of global termination	4δ+2Δ	3δ+3Δ
Tolerate datacenter failures?	yes	yes
Tolerate region failures?	no	yes

Figure 1. 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.

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 1 Geo-SDUR, client c's code

```
 1: begin(t):
 2:     $t.rs \leftarrow \emptyset$                         {initialize readset}
 3:     $t.ws \leftarrow \emptyset$                         {initialize writeset}
 4:     $t.st[1...P] \leftarrow [\bot ... \bot]$   {initialize vector of snapshot times}
 5: read(t, k):
 6:     $t.rs \leftarrow t.rs \cup \{k\}$                    {add key to readset}
 7:     if $(k, \star) \in t.ws$ then                {if key previously written...}
 8:         return $v$ s.t. $(k, v) \in t.ws$           {return written value}
 9:     else                                    {else, if key never written...}
10:         $p \leftarrow partition(k)$                  {get the key's partition}
11:         send(read, $k$, $t.st[p]$) to $s \in S_p$        {send read request}
12:         wait until receive $(k, v, st)$ from $s$          {wait response}
13:         if $t.st[p] = \bot$ then $t.st[p] \leftarrow st$   {if first read, init snapshot}
14:         return $v$                             {return value from server}
15: write(t, k, v):
16:     $t.ws \leftarrow t.ws \cup \{(k, v)\}$                {add key to writeset}
17: 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}
```

F. Algorithm in detail

Algorithm 1 shows the client side of the protocol. To execute a read, the client sends a request to a server in the partition that stores the accessed key (lines 10–12). The snapshot of a transaction is represented by an array of integers, one per partition (line 4). Upon receiving the first response from the server, the client initializes its snapshot time for the corresponding partition (line 13). Subsequent reads to the same partition will include the snapshot count so that the transaction sees a consistent database view. Writes are buffered at the client, and only sent to servers at commit time. When the execution phase ends the transaction is sent to a preferred server possibly near the client, which in turn broadcasts the transaction for certification to all partitions concerned by the transaction. The client then waits for the transaction's outcome (line 17–20). Note that a client can choose to commit a transaction against a server it did not contact previously while executing read operations.

Algorithm 2 shows the protocol for a server s in partition

p. When s receives a request to commit transaction t, s calls procedure submit (lines 11–12), which broadcasts t to each one of the partitions involved in t, possibly delaying the broadcast at partition p, if transaction delaying is enabled (lines 41–45). In Algorithm 2, $delay(x, p)$ (line 44) returns the estimated delay between partitions x and p.

Upon delivering transaction t (line 15), s updates the delivery counter DC and set t's reordering threshold, which will be only used by global transactions (lines 16–17). Then s certifies and possibly reorders t (line 18). The reorder function (lines 48–64) first certifies t against transactions that committed after t started (line 49). This check uses function ctest (lines 46–47), which distinguishes between local and global transactions: while a local transaction has its readset compared against the writeset of committed transactions, a global transactions has both its readset and writeset compared against committed transactions (see Section III for a description of why this is needed). If some conflict is found, t must abort (line 50); otherwise the check continues.

A global transaction t is further checked against all pending transactions (lines 51–52), 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 53).

A local transaction t will be possibly reordered among pending transactions (lines 55–64). 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 55), 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 56); (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 57); (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 58); and finally (d) the reordering of t must not invalidate the votes of any previously certified transactions (lines 59–60). If no position satisfies the conditions above, t must abort (line 61); otherwise, s inserts t in the appropriate position in the list of pending transactions (lines 62–63) and t is declared committed (line 64).

A local transaction is committed as soon as it is the head of the pending list (lines 23–25). The complete function (lines 34–40) first removes the terminating transaction t from the pending list (line 35) and if t's outcome is commit (line 36), it applies t's writes to the database (line 37), exposes t's changes to new transactions (line 38), and updates the snapshot counter (line 39). Whatever the outcome of t, server s notifies the client (line 40).

A global transaction t that reaches the head of the pending list (lines 26–27) can only be completed at server s if (a) s received votes from all partitions involved in t (line 28) and (b) t has reached its reordering threshold (line 29). If these conditions hold, s checks whether all partitions voted to commit t (lines 30–32) and completes t accordingly (line 33).

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 (lines 13–14) and transactions are constantly delivered, increasing the value of the DC counter (line 16). 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 t or transaction t; servers in p' will act according to the first message delivered (see [5] for further details).

G. Correctness

In this section, we argue that SDUR implements serializability. We briefly recall how SDUR ensures serializable executions (a detailed discussion can be found in [5]) and then extend our argument to include the delaying and reordering of transactions.

1) The correctness of SDUR: In SDUR, transactions in a partition p, both local and global, are serialized following their delivery order. Certification checks whether a delivered transaction t can be serialized after all previously delivered and committed transactions t'. If t has received a database snapshot that includes t''s writes, then t''s commit *precedes* t's start at p (i.e, t' and t executed sequentially) and t can be obviously serialized after t'. If t' committed after t received its database snapshot, then in order for t to be serialized after t', t must not have read any item written by t'.

The procedure above guarantees that local transactions are serialized within a partition. The certification of global transactions is more complex, to account for the lack of total order across partitions.

Global transactions t and t' can interleave in three different ways [5]: (a) t' precedes t in all partitions, in which case t' can be trivially serialized before t; (b) t' precedes t in partition p and they are concurrent in p', in which case the certification test at p' guarantees that t' can be serialized before t in p'; and (c) t' and t are concurrent in p and p', in which case the certification test at p and p' ensures that they can be serialized in any order at every partition. Since a transaction only commits in a partition after it receives the votes from all other involved partitions, it is impossible for t' to precede t in p and for t to precede t' in p'.

2) The correctness of 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

Algorithm 2 Geo-SDUR, server s's code in partition p

```
 1: Initialization:
 2:     $DB \leftarrow [\ldots]$                                  {list of applied transactions}
 3:     $PL \leftarrow [\ldots]$                                  {list of pending transactions}
 4:     $SC \leftarrow 0$                                         {snapshot counter}
 5:     $DC \leftarrow 0$                                         {delivered transactions counter}
 6:     $VOTES \leftarrow \emptyset$                              {votes for global transactions}
 7: when receive($read, k, st$) from $c$
 8:     if $st = \bot$ then $st \leftarrow SC$                    {if first read, init snapshot}
 9:     retrieve($k, v, st$) from database                        {most recent version $\leq st$}
10:     send($k, v, st$) to $c$                                   {return result to client}
11: when receive($commit, t$)
12:     submit($t$)                                               {see line 41}
13: when receive($tid, v$) from partition $p$
14:     $VOTES \leftarrow VOTES \cup (tid, p, v)$                 {one more vote for tid}
15: when adeliver($c, t$)
16:     $DC \leftarrow DC + 1$                                    {one more transaction delivered}
17:     $t.rt \leftarrow DC + ReorderThreshold$                   {set t's Reorder Threshold}
18:     $v \leftarrow$ reorder($t$)                               {see line 48}
19:     if $v = abort$ then                                       {reordering resulted in abort?}
20:         complete($t, v$)                                      {see line 34}
21:     if $t$ is global then
22:         send($t.id, v$) to all servers in $partitions(t)$     {send votes}
23: when head($PL$) is local
24:     $t \leftarrow$ head($PL$)                                 {get head without removing entry}
25:     complete($t, commit$)                                     {see line 34}
26: when head($PL$) is global
27:     $(c, t, v) \leftarrow$ head($PL$)                         {get head without removing entry}
28:     if $\forall k$ s.t $t.st[k] \neq \bot : (t.id, k, \star) \in VOTES$ and {has all votes?}
29:             $t.rt = DC$                                       {and t reached threshold?}
30:         $outcome \leftarrow commit$                           {a priori commit, but...}
31:         if $(t.id, \star, abort) \in VOTES$ then              {one abort vote and...}
32:             $outcome \leftarrow abort$                        {t will be aborted}
33:         complete($t, outcome$)                                {see line 34}
34: function complete($t, outcome$)                               {used in lines 20, 25, 33}
35:     $PL \leftarrow PL \ominus t$                              {remove t from PL}
36:     if $outcome = commit$ then                                {if t commits...}
37:         apply $t.ws$ with version $SC$ to database            {apply changes}
38:         $DB[SC + 1] \leftarrow t$                             {create next snapshot and...}
39:         $SC \leftarrow SC + 1$                                {...expose snapshot to clients}
40:     send($outcome$) to client of $t$
41: procedure submit($t$):
42:     let $P$ be $partitions(t) \setminus \{p\}$                {broadcast t to each...}
43:     for all $x \in P : abcast(x, t)$                          {...remote partition}
44:     $\Delta \leftarrow max(\{delay(x, p) \mid x \in P\})$     {determine maximum delay}
45:     $abcast(p, t)$ after $\Delta$ time units                  {delay local broadcast}
46: function ctest($t, t'$):
47:     $(t.rs \cap t'.ws = \emptyset) \wedge (t \text{ is local} \vee (t.ws \cap t'.rs = \emptyset))$
48: function reorder($t$):
49:     if $\exists t' \in DB[t.st[p] \ldots SC] : ctest(t, t') = false$ then
50:         return $abort$                                        {t aborts if conflicts with committed $t'$}
51:     if $t$ is global then
52:         if $\exists t' \in PL : ctest(t, t') = false$ then return $abort$
53:         append $t$ to $PL$                                    {include t in pending list if no conflicts}
54:     else
55:         let $i$ be the smallest integer, if any, such that
56:         $\forall k < i : PL[k].ws \cap t.rs = \emptyset$ and  {t's reads are not stale}
57:         $\forall k \geq i : (PL[k]$ is global and             {no leaping local transactions}
58:             $PL[k].rt < DC$ and                               {no leaping globals after threshold}
59:             $t.ws \cap PL[k].rs = \emptyset$ and              {previous votes still valid}
60:             $t.rs \cap PL[k].ws = \emptyset)$                 {ditto!}
61:         if no $i$ satisfies the conditions above then return $abort$
62:         for $k$ from size($PL$) downto $i$ do $PL[k + 1] \leftarrow PL[k]$
63:         $PL[i] \leftarrow t$                                  {after making room (above), insert t}
64:         return $commit$                                       {t is a completed transaction!}
```

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.

3) The correctness of reordering transactions: 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 case (a) above, from Algorithm 2, the reordering of a local transaction t (lines 48–64) is a deterministic procedure that depends on $DB[t.st[p]..SC]$ (line 49), PL (lines 56–60), and DC(line 58). 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 15). PL is possibly modified in the reorder procedure (line 63) 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 (lines 34–40), called (i) after t is delivered (line 20), (ii) when the head of PL is a local transaction (line 25), and (iii) when the head of PL is a global transaction u (line 33).

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 (lines 28 and 29). 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 16) DC depends only on delivered messages and from atomic broadcast all servers deliver the same transactions in the same order.

Finally, 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 59–60). Moreover, t's readset must not intersect the writeset of any concurrent transaction delivered before t (lines 49 and 56), which is essentially the certification test for local transactions in SDUR.

V. Implementation and optimizations

We use Paxos as our atomic broadcast primitive. There is one instance of Paxos per partition. Our Paxos implementation uses Berkeley DB to log delivered values to disk. Therefore, the committed state of a server can be recovered from the log. Our prototype differs from Algorithms 1 and 2 in the following aspects:

- A client connects to a single server and submits all of its read and commit requests to that server. When a server receives a read request for key k that is not local, the server routes it to a server in the partition that stores k. Partitioning is transparent to the client.
- Servers use bloom filters to check for intersections between transactions, and to store past transactions. The implementation only keeps track of the last K bloom filters, where K is a configurable parameter. Bloom filters have negligible memory requirements and allow us to broadcast only the hash values of the read set, thus reducing network bandwidth. Using bloom filters results in a small amount of transactions aborted due to false positives.

VI. 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 techniques introduced earlier.

A. Setup and benchmarks

We ran the experiments using Amazon's EC2 infrastructure. We used medium instances equipped with a single core (two EC2 compute units) and 3.75 GB of RAM. We deployed servers in three different regions: Ireland (EU), N. Virginia (US-EAST), and Oregon (US-WEST). We observed the following inter-region latencies: (a) $\approx$100 ms between US-EAST and US-WEST, (b) $\approx$90 ms between US-EAST and EU, and (c) $\approx$170 ms between US-WEST and EU.

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 transactions updates one local object and one remote object. We use one million data items per partition, where each data item is 4 bytes long.

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 its 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 75% of the maximum performance, for both benchmarks.

B. Baseline

We implemented and deployed SDUR in a geographically distributed environment following the two alternatives discussed in Section IV-B, "WAN 1" and "WAN 2". Figure 2 shows the throughput and latency for both WAN 1 and WAN 2 deployments with workloads mixes containing 0%, 1%, 10% and 50% of global transactions. For 0% and 10% of globals, we also show the cumulative distribution function (CDF) of latency. Latency values correspond to their 99-th percentile and average.

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 IV-C). In the absence of global transactions, local transactions can execute within 32.6 ms in WAN 1. The latency of locals increases to 321 ms with 1% of global transactions, a 10x increase. We observed that in workloads with 10% and 50% of global transactions, latency of locals reduced to 176.8 ms (5.4x increase to the 0% configuration) and

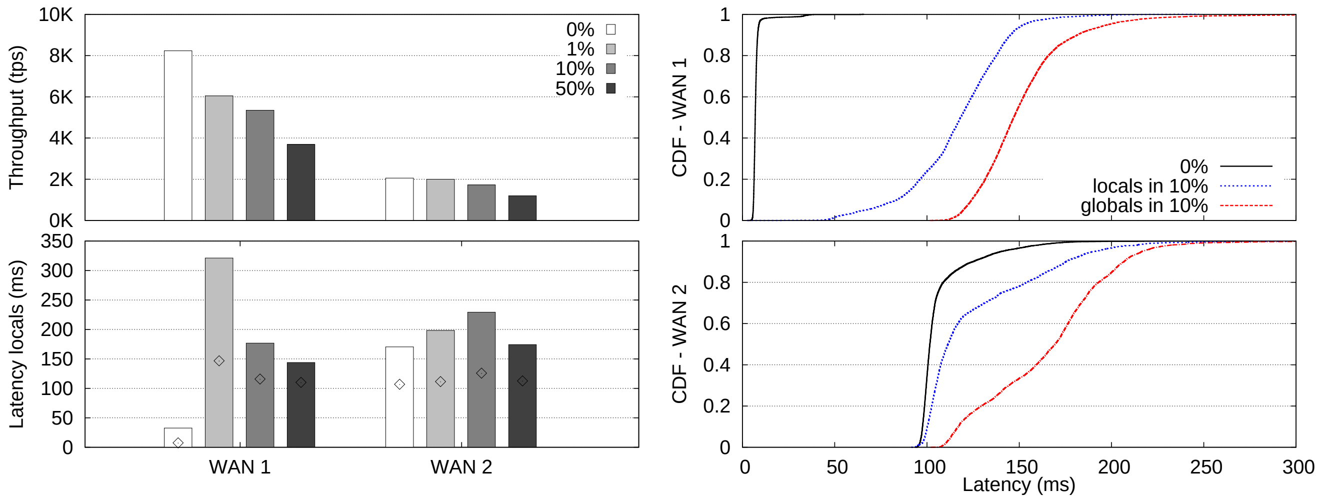


Figure 2. SDUR's local and global transactions in two geographically distributed deployments (WAN 1 and WAN 2). Throughput in transactions per second (tps), latency 99-th percentile (bars) and average latency (diamonds in bars) in milliseconds (ms), and CDFs of latencies.

143.9 ms (4.41x), respectively. We attribute this behavior to the fact that with 1% of global transactions, messages are sent over different regions relatively infrequently. This effect tends to disappear as we increase the number of globals and hence the traffic between regions. In WAN 2, local transactions alone experienced a latency of 170.4 ms, while in workload mixes of 1%, 10% and 50% of global transactions latency increased to 198.4 ms (1.16x), 229.3 ms (1.34x) and 174.3 ms (1.02x) , respectively. The CDFs 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 IV-C).

C. *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 3 (bottom left graph) shows that while the technique has a positive effect in workloads with 1% of global transactions—delaying globals by 20 ms resulted in a reduction in the latency of local transactions from 321 ms to 232.2 ms—it did not present any significant improvements in workloads with 10% and 50% of global transactions. In settings with 1% of globals, global transactions also benefit from the delaying technique as their latency is also reduced. This happens because not only local transactions are less prone to waiting for a pending global transactions, but also global transactions delivered after the pending transaction will wait less.

D. *Reordering transactions*

Figures 4 and 5 show the effects of reordering in the latency of local transactions under various workloads and deployments WAN 1 and WAN 2. We assess different reordering thresholds in configurations subject to a similar throughput. In WAN 1 (Figure 4), reordering has a positive impact on both local and global transactions for all three workload mixes. For example, for 1% global transactions, a reordering threshold of 320 reduces the 99-th percentile latency of local transactions from 321 ms (in baseline) to 168 ms, a 48% improvement. For mixes with 10% and 50% of global transactions the improvement is 58% and 69% respectively. The 99-th percentile of the corresponding global transactions experience a decrease in latency of 28%, 15% and 12%, respectively. Local transactions in WAN 2 (Figure 5) also benefit from reordering, although there is a tradeoff between the latency of locals and globals, something we did not experience in 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 229.3 ms (in baseline) to 161.1 ms, with a small increase in the latency of global transactions from 251.1 ms to 253.4 ms. Similar trends are seen for workloads with 1% and 50% of global transactions.

E. *Social network application*

Figure 6 shows the effects of reordering in our social network application. In WAN 1, both the 99-th percentile and the average latency of all operations improve with respect to the baseline (SDUR). The timeline, post, local follow, and global follow operations present latency improvements (99-th percentile) of 67%, 70%, 71% and 12%, respectively. In WAN 2, timeline, post, and local follow experienced a reduction in latency (99-th percentile) of 55%, 20% and 21%, respectively, while global follow remained constant.

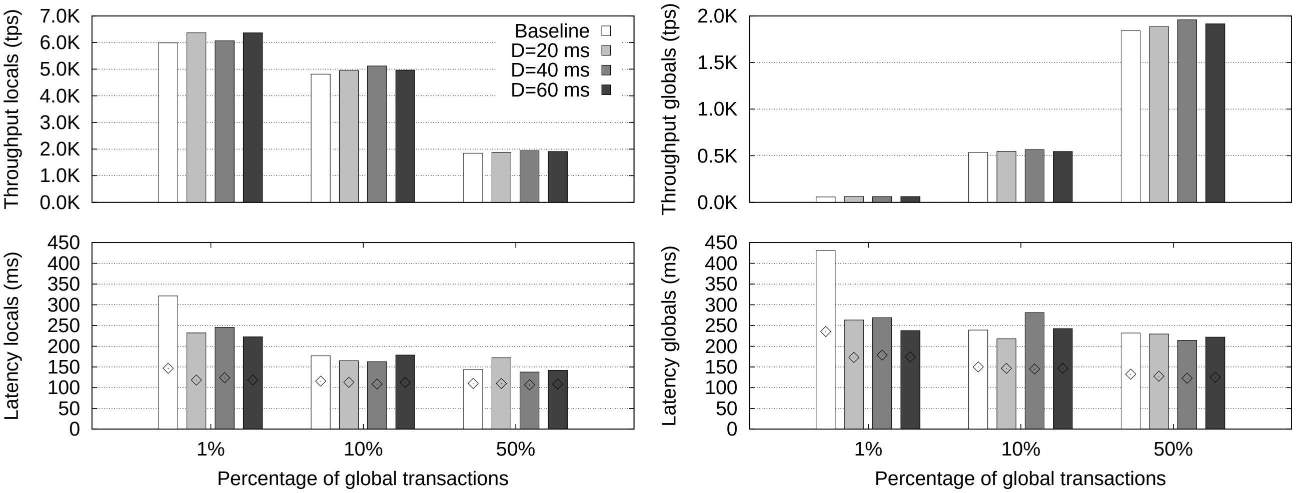


Figure 3. Throughput and latency of local and global transactions with delayed transactions.

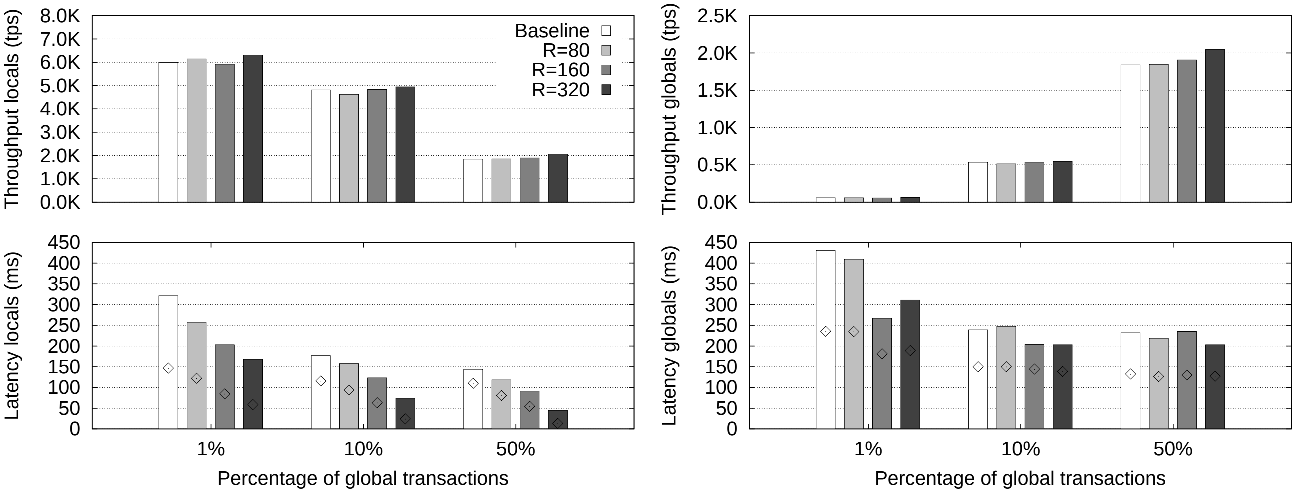


Figure 4. Throughput and latency of local and global transactions with reordering in WAN 1.

VII. Related work

Several protocols for deferred update replication where servers keep a full copy of the database exist (e.g., [1], [2], [3], [4], [12]). In [5] 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.

Our reordering algorithm is based on the algorithm described in [1], originally for reducing the abort rate. In this paper, we extend the idea to avoid the delay imposed by global communication on local transactions.

Many storage and transactional systems have been proposed recently. Some of these systems (e.g., Cassandra,[5] Dymano [13], Voldemort[6]) 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 [14] 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 [15] offers stronger guarantees by means of minitransactions on unstructured data. Similarly to SDUR, minitransactions are certified upon commit. Differently from SDUR, both update

[5] http://cassandra.apache.org

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

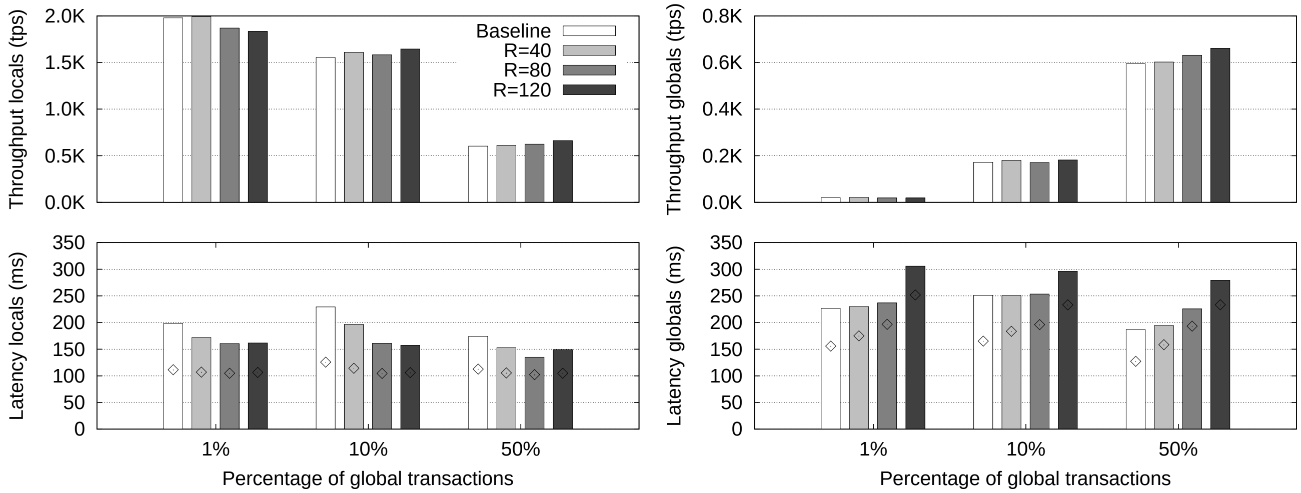


Figure 5. Throughput and latency of local and global transactions with reordering in WAN 2.

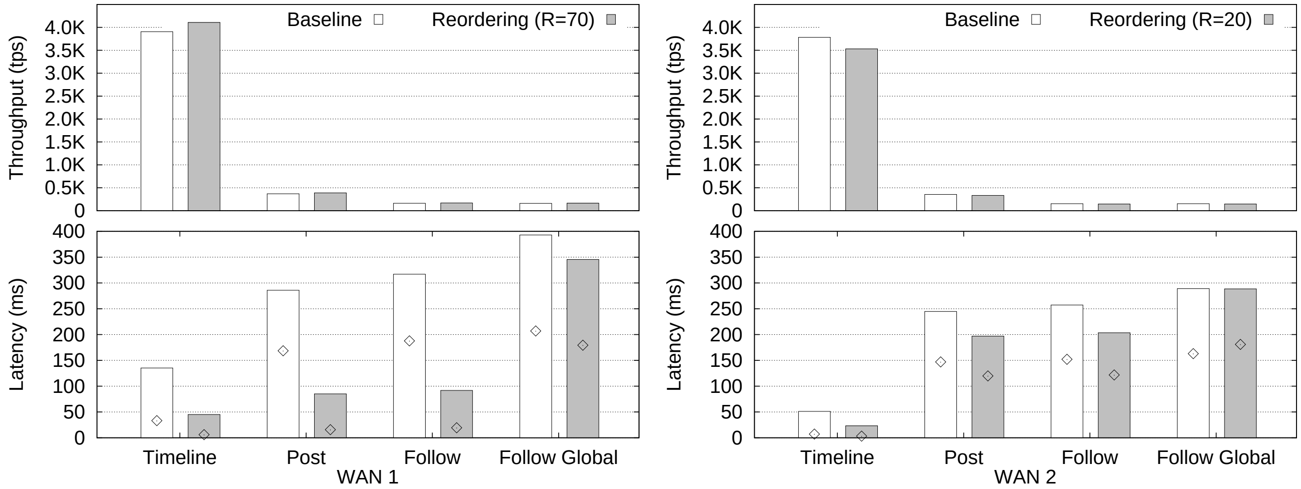


Figure 6. Social network application in WAN 1 and WAN 2.

and read-only transactions must be certified in Sinfonia, and therefore can abort. Read-only transactions do not abort in SDUR.

COPS [16] is storage system that ensures a strong version of causal consistency, which in addition to ordering causally related write operations also orders writes on the same data items. COPS provides read-only transactions, but does not provide multi-key update transactions.

Walter [17] 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.

Vivace [18] 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 [19] and Yahoo's Pnuts [20] 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 SDUR, we consider Spanner [21], MDCC and P-Store [22]

P-Store [22] is perhaps the closest to our work in that it implements deferred update replication optimized for wide-area networks. Unlike SDUR, 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 terminates transactions in parallel. SDUR can also terminate transactions in parallel, and in addition to that we use reordering to further reduce delays.

Spanner [21] is a distributed database for WANs. Like SDUR the database is partitioned and replicated over several Paxos instances. Spanner uses a combination of two-phase commit and a so called 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.

MDDC [10] 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.

VIII. Conclusion

This paper discusses scalable deferred update replication in geographically distributed settings. SDUR scales deferred update replication, a well-established approach used in several database replicated systems, by means of data partitioning. SDUR 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 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.

References

[1] F. Pedone, R. Guerraoui, and A. Schiper, "The Database State Machine approach," *Distrib. Parallel Databases*, vol. 14, pp. 71–98, July 2003.

[2] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *VLDB*, 2000.

[3] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Middleware based data replication providing snapshot isolation," in *SIGMOD*, 2005.

[4] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso, "MIDDLE-R: Consistent database replication at the middleware level," *ACM Transactions on Computer Systems*, vol. 23, no. 4, pp. 375–423, 2005.

[5] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, 2012.

[6] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, 1981.

[7] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[8] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

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

[10] T. Kraska, G. Pang, M. J. Franklin, and S. Madden, "MDCC: Multi-Data Center Consistency," *CoRR*, 2012.

[11] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*, 2011.

[12] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases," in *EuroPar*, 1997.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS*, 2007.

[14] J. Rao, E. Shekita, and S. Tata, "Using Paxos to build a scalable, consistent, and highly available datastore," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 243–254, 2011.

[15] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP*, 2007.

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS," in *SOSP*, 2011.

[17] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP*, 2011.

[18] B. Cho and M. K. Aguilera, "Surviving congestion in geo-distributed storage systems," in *USENIX ATC*, 2012.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, 2008.

[20] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, 2008.

[21] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Googles globally-distributed database," *OSDI*, 2012.

[22] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *SRDS*, 2010.