Scalable Deferred Update Replication

Daniele Sciascia
University of Lugano (USI)
Switzerland

Fernando Pedone
University of Lugano (USI)
Switzerland

Flavio Junqueira
Yahoo! Research
Spain

***Abstract*—Deferred update replication is a well-known approach to building data management systems as it provides both high availability and high performance. High availability comes from the fact that any replica can execute client transactions; the crash of one or more replicas does not interrupt the system. High performance comes from the fact that only one replica executes a transaction; the others must only apply its updates. Since replicas execute transactions concurrently, transaction execution is distributed across the system. The main drawback of deferred update replication is that update transactions scale poorly with the number of replicas, although read-only transactions scale well. This paper proposes an extension to the technique that improves the scalability of update transactions. In addition to presenting a novel protocol, we detail its implementation and provide an extensive analysis of its performance.**

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

I. Introduction

Deferred update replication is a well-established approach to fault-tolerant data management systems. The idea is conceptually simple: a group of servers fully replicate the database; to execute a transaction, a client chooses one server and submits the transaction commands to this server [19]. During the execution of the transaction, there is no coordination among different servers. When the client issues a commit request, the transaction starts termination: its updates (and some meta data) are atomically broadcast to all servers. Atomic broadcast ensures that all servers deliver the updates in the same order and can certify the transaction in the same way. Certification guarantees that the database remains consistent despite the concurrent execution of transactions. The transaction passes certification and commits in a server only if it can be serialized with other committed transactions; otherwise the transaction is aborted—essentially, the technique relies on optimistic concurrency control [12].

Deferred update replication has two main characteristics, which contribute to its performance. First, an update transaction is executed by a single server; the other servers only certify the transaction and apply its updates to their database, should the transaction pass certification. Applying a transaction's updates is usually cheaper than executing the transaction. Second, read-only transactions do not need to be certified. A replica can serialize a read-only transaction by carefully synchronizing it locally (e.g., using a multiversion database). Consequently, read-only transactions scale with the number of replicas.

Several database replication protocols are based on deferred update replication (e.g., [1], [11], [14], [17], [19]), which can be explained by its performance advantages with respect to other replication techniques, such as primary-backup and state-machine replication. With state-machine replication, every update transaction must be executed by all servers [24]. Thus, adding servers does not increase the throughput of update transactions; throughput is limited by what one replica can execute. With primary-backup replication [29], the primary first executes update transactions and then propagates their updates to the backups, which apply them without re-executing the transactions; the throughput of update transactions is limited by the capacity of the primary, not by the number of replicas. Servers act as "primaries" in deferred update replication, locally executing transactions and then propagating their updates to the other servers.

Although deferred update replication performs better than state-machine replication and primary-backup replication, update transactions scale poorly with the number of replicas. The reason is that even though applying transaction updates is cheaper than executing the transactions, it must be done by each replica for every committed transaction. Intuitively, the problem is that while increasing the number of replicas also increases the total number of transactions executed per time unit, it does not reduce a replica's load of certifying transactions and applying their updates. Asymptotically, the system throughput is limited by the number of transactions atomic broadcast can order and a *single* replica can certify and apply to its database per time unit.

This paper proposes Scalable Deferred Update Replication (S-DUR), an extension to deferred update replication that makes read-only transactions and some types of update transactions scale with the number of servers. Our solution is to divide the database into partitions, replicate each partition among a group of servers, and orchestrate the execution and termination of transactions across partitions. Local transactions, those that read and write items within a single partition, are handled as in traditional deferred update replication. Global transactions, those that access items in more than one partition, undergo a different execution and

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

termination procedure. During execution, a global transaction has its read operations submitted to servers in different partitions. During termination, partitions coordinate to ensure consistency using a two-phase commit-like protocol, where each participant is a partition. Different than two-phase commit [3], the termination of global transactions is non-blocking since each partition is highly available. Moreover, S-DUR is faithful to the original deferred update replication approach in that read-only transactions, both local and global, are serialized without certification.

We have implemented and evaluated the performance of S-DUR under various conditions. The experiments reported in the paper show that read-only transactions and local update transactions scale linearly with the number of servers. In workloads with global update transactions only, S-DUR performs similarly to the original deferred update replication. In workloads with a mix of local and global transactions, S-DUR's performance depends on the percentage of global transactions: with update transactions only, S-DUR's performance is at least as good as the performance of deferred update replication; with read-only transactions only, S-DUR compares to deferred update replication when at most 20% of transactions are global and worse otherwise.

The remainder of the paper is structured as follows. Section II presents our system model and definitions used throughput the paper. Section III recalls the deferred update replication approach in detail and discusses the scalability of update transactions. Section IV introduces scalable deferred update replication and argues for its correctness. 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

We consider a system composed of an unbounded set $C = \{c_1, c_2, ...\}$ of client processes and a set $S = \{s_1, ..., s_n\}$ of database server processes. Processes communicate through message passing and do not have access to a shared memory. 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*.

The system is asynchronous: there is no bound on messages delays and on relative process speeds. 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, with primitives $abcast(m)$ and $adeliver(m)$. Atomic broadcast ensures two properties: (1) if message m is delivered by a process, then every correct process eventually delivers m; and (2) no two messages are delivered in different order by their receivers.[1]

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. 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.

The isolation property is *serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [3].

III. Deferred Update Replication

In this section we review the deferred update replication technique, explain how read-only transactions are serialized without certification, and examine scalability issues.

A. The deferred update replication approach

In Deferred Update Replication (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 committed or aborted.

Before starting a transaction t, a client c selects the replica s that will execute t's operations; other replicas will not be involved in t's execution. 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* the client will see upon executing other read operations for t. Write operations are locally buffered by c. It is only during transaction termination that updates are propagated to the servers.

In the termination phase, the client atomically broadcasts t's readset and writeset to all replicas. Upon delivering t's termination request, s certifies t. Certification ensures a serializable execution; it essentially checks whether t's read operations have seen values that are still up-to-date when t is certified. If t passes certification, then s executes t's writes against the database and assigns each new value the same version number k, reflecting the fact that t is the k-th committed transaction at s.

Algorithms 1 and 2 illustrate the technique for the client and server, respectively. A read operation on item k starts by updating transaction t's readset (line 6, Algorithm 1) and

[1] Solving atomic broadcast requires additional assumptions [5], [9]. In this paper we simply assume the existence of an atomic broadcast oracle.

Algorithm 1 Deferred update replication, client c's code

```
1:  begin(t):
2:      t.rs ← ∅                                   {initialize readset}
3:      t.WS ← ∅                                   {initialize writeset}
4:      t.st ← ⊥                                   {initially transactions have no snapshot}
5:  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:         send(read, k, t.st) to some s ∈ S       {send read request}
11:         wait until receive(k, v, st) from s    {wait for response}
12:         if t.st = ⊥ then t.st ← st             {if first read, init snapshot}
13:         return v                               {return value from server}
14: write(t, k, v):
15:     t.ws ← t.ws ∪ {(k, v)}                     {add key to writeset}
16: commit(t):
17:     if t.ws = ∅ then                           {if transaction is read-only...}
18:         return commit                          {commit it right away}
19:     else                                       {else, if it is an update...}
20:         abcast(c, t)      {abcast it for certification and wait outcome}
21:         wait until receive(outcome) from s ∈ S {ditto}
22:         return outcome                         {return outcome}
```

Algorithm 2 Deferred update replication, server s's code

```
1:  Initialization:
2:      SC ← 0                                     {initialize snapshot counter}
3:      WS[...] ← ∅                                {initialize committed writesets}
4:  when receive(read, k, st) from c
5:      if st = ⊥ then st ← SC                     {if first read, initialize snapshot}
6:      retrieve(k, v, st) from database           {most recent version ≤ st}
7:      send(k, v, st) to c                        {return result to client}
8:  when adeliver(c, t)
9:      outcome ← certify(t)                       {outcome is either commit or abort}
10:     if outcome = commit then                   {if it passes certification...}
11:         apply t.ws to database                 {apply committed updates to db}
12:     send(outcome) to c                         {return outcome to client}
13: function certify(t)                            {used in line 9}
14:     for i ← t.st to SC do                      {for all concurrent transactions...}
15:         if WS[i] ∩ t.rs ≠ ∅ then               {if some intersection...}
16:             return abort                       {transaction must abort}
17:     SC ← SC + 1                                {here no intersection: one more snapshot}
18:     WS[SC] ← items(t.ws)                       {keep track of committed writeset}
19:     return commit                              {transaction must commit}
```

then checking whether t has previously updated k (line 7), in which case this must be the value returned (line 8); otherwise the client selects a server s to handle the request (line 10). The client then waits for a response from s (line 11). If c suspects that s crashed, it simply contacts another replica (not shown for brevity). Upon the first read, c updates t's snapshot (line 12), which will determine the version of future reads performed by t. To perform a write as part of t, the client simply adds item (k, v) to $t.ws$ (lines 14–15). The commit of a read-only transaction is local (lines 17 and 18). If t is an update transaction, upon commit the client atomically broadcasts t to all servers and waits for a response (lines 20–22).

Servers maintain variables SC and WS (lines 2–3, Algorithm 2). SC has the latest snapshot created by server s. WS is a vector, where entry $WS[i]$ contains the items updated by the i-th committed transaction at s. When s receives the first read operation from a client, say, on key k, s returns the value of k in the latest snapshot (lines 4–7) together with the snapshot id. The client will use this snapshot id when executing future read operations. If s receives a read request with a snapshot id, then it returns a value consistent with the snapshot (line 6; see also next section). Upon delivering t (line 8), s certifies it (line 9) and replies to c (line 12). If the outcome of certification is commit, s applies t's updates to the database (lines 10–11). Certification checks the existence of some transaction u that (a) committed after t received its snapshot and (b) updated an item read by t. If u exists, then t must abort (lines 14–16). If t passes certification, one more snapshot is created (line 17) and t's updated entries are recorded for the following certification (lines 17–18).

B. Read-only transactions

Database snapshots guarantee that all reads performed by a transaction see a consistent view of the database. Therefore, a read-only transaction t is serialized according to the version of the value t received in its first read operation and does not need certification. Future reads of a transaction return versions consistent with the first read. A read on key k is consistent with snapshot SC if it returns the most recent version of k equal to or smaller than SC (line 6, Algorithm 2). This rule guarantees that between the version returned for k and SC no committed transaction u has modified the value of k (otherwise, u's value would be the most recent one).

C. Scalability issues

While read-only transactions scale with the number of replicas in deferred update replication, the same does not hold for update transactions. There are two potential bottlenecks in the termination protocol: (1) every update transaction needs to be atomically broadcast; and (2) every server needs to certify and apply the updates of every committing transaction. Throughput is therefore bounded by the number of transactions that can be atomically broadcast or by the number of transactions that a server can execute, certify and apply to the database.

If performance is determined by the execution and termination of transactions, then adding replicas to deferred update replication may increase throughput, although the expected gains are limited. Asymptotically, the throughput of update transactions is determined by the number of transactions that atomic broadcast can order and a single replica can terminate per unit of time. This is an inherent limitation of DUR, which we discuss in the next section.

IV. Scalable Deferred Update Replication

The idea of scalable deferred update replication is to divide the database into P partitions and replicate each one among a group of servers. In this section, we first show a straightforward extension of deferred update replication to account for partitioned data. We then justify the need for more sophisticated solutions and present a complete and correct protocol. We also discuss read-only transactions, some special cases, and conclude with a correctness argument for the new protocol.

A. Additional definitions and assumptions

The set of servers that replicate partition p is denoted by S_p . For each key k, we denote $partition(k)$ the partition to which k belongs. Transaction t is said to be *local* to partition p if $\forall(k, -) \in Items(t) : partition(k) = p$. If t is not local to any partition, then we say that t is *global*. The set of partitions that contain items read or written by t is denoted by $partitions(t)$.

Hereafter, we assume that partitions do not become unavailable and that the atomic broadcast primitive within each partition is live. Moreover, 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, $writeset(t) \subseteq readset(t)$.

B. A straightforward (and incorrect) extension

Transactions that are local to a partition p can be handled as in regular deferred update replication. Instead, global transactions need special care to execute and terminate.

During the execution phase of a global transaction t, client c submits each read operation of t to the appropriate partition.[2] Since each partition is implemented with deferred update replication, reads issued to a single partition see a consistent view of the database.

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 $writset(t)_p$, respectively. Client c uses one broadcast operation per partition. When a server $s \in S_p$ delivers t's $readset(t)_p$ and $writeset(t)_p$, s certifies t against transactions delivered before t in partition p, as in traditional deferred update replication, and then sends the outcome of certification, the partition's *vote*, to the servers in $partitions(t)$. Since certification within a partition is deterministic, every server in S_p will compute the same vote for t. Then, s waits for the votes from $partitions(t)$. If every partition votes to commit t, then s applies t's updates to the database and commits t; otherwise s aborts t.

[2]This assumes that clients are aware of the partitioning scheme. Alternatively, a client can connect to a single server and submit all its read requests to this server, which will then route them to the appropriate partition.

We now show an execution of this protocol that violates serializability. Recall that transactions are certified in the order in which they are delivered.

Example. In the following example, partition P_x stores item x, and partition P_y stores item y. Let t_i and t_j be two global transactions such that t_i reads x and then reads and writes y; t_j reads y and then reads and writes x (see Figure 1). During termination, servers in P_x first deliver t_i's commit request and then t_j's commit request; servers in P_y deliver t_j and then t_i—this is possible because the termination of global transactions requires multiple invocations of atomic broadcast, one per partition. Transaction t_i passes certification at P_x because no transaction updated x since t_i's snapshot; it passes certification at P_y because no other transaction updates y at P_y. Thus t_i commits. By a similar argument, t_j also commits. However, their execution cannot be serialized (i.e., in any serial execution involving t_i and t_j, either t_i must read t_j's writes or the other way round).

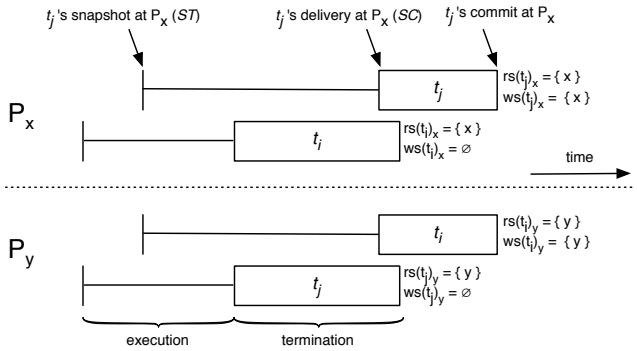

Figure 1: Problematic execution.

C. A complete and correct protocol

The problem with the protocol presented in the previous section stems from the fact that global transactions are not delivered and certified in the same order across partitions, something that cannot happen with the original deferred update replication technique since transaction termination is totally ordered by atomic broadcast.

In the examples shown before, certification at partition P_x determined that t_j can be serialized after t_i and certification at partition P_y determined that t_i can be serialized after t_j, but this is obviously not enough. To solve the problem, we use a stronger condition for certification, where each partition checks whether t_i and t_j can be serialized in any order with regards to one another (i.e., both t_i before t_j and t_i after t_j).

More precisely, let σ be the set of transactions that are (a) delivered and certified before t_i but (b) not included in t_i's snapshot (i.e., because they committed after t_i started). At certification, servers in p check whether t_i's readset intersects the writeset of any t_j in σ. If t_i passes this test,

Algorithm 3 Scalable DUR, client c's code

```
1: begin(t):
2:    t.rs ← ∅                                   {initialize readset}
3:    t.ws ← ∅                                   {initialize writeset}
4:    t.st[1...P] ← [⊥...⊥]                      {initialize vector of snapshot times}
5: 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: write(t, k, v):
16:   t.ws ← t.ws ∪ {(k, v)}                     {add key to writeset}
17: commit(t):
18:   for all p s.t. t.st[p] ≠ ⊥ : abcast(c, t) to S_p
19:   wait until receive(outcome) from s ∈ S
20:   return outcome                             {outcome is either commit or abort}
```

Algorithm 4 Scalable DUR, server s's code in partition p

```
1: Initialization:
2:    SC ← 0                                     {snapshot counter}
3:    PSC ← 0                                    {pending snapshot counter}
4:    RS[...] ← ∅                                {committed readsets}
5:    WS[...] ← ∅                                {committed writesets}
6:    VOTES ← ∅                                  {votes for global transactions}
7:    PENDING ← ∅                                {transactions delivered but not completed}
8:    DECIDED ← ∅                                {ids of completed transactions}
9: when receive(read, k, st) from c
10:   if st = ⊥ then st ← SC                     {if first read, init snapshot}
11:   retrieve(k, v, st) from database           {most recent version ≤ st}
12:   send(k, v, st) to c                        {return result to client}
13: when adeliver(c, t)
14:   v ← certify(t)                             {compute the local outcome}
15:   PENDING ← PENDING ⊕ (c, t, v)
16:   if t is global then                        {send vote if transaction is global}
17:      send(t.id, p, v) to all servers in partitions(t)
18: when receive(tid, p, v) and tid ∉ DECIDED
19:   VOTES ← VOTES ∪ (tid, p, v)
20: when head(PENDING) is local
21:   (c, t, v) ← head(PENDING)
22:   finish(c, t, v)
23: when head(PENDING) is global
24:   (c, t, v) ← head(PENDING)
25:   if ∀k s.t t.st[k] ≠ ⊥ : (t.id, k, ⋆) ∈ VOTES then
26:      outcome ← commit
27:      if (t.id, ⋆, abort) ∈ VOTES then
28:         outcome ← abort
29:      finish(c, t, outcome)
30:      DECIDED ← DECIDED ∪ t.id
31:      VOTES ← {(i, p, v) ∈ VOTES | i ≠ t.id}
32: function finish(c, t, outcome)               {used in lines 21, 28}
33:   if outcome = commit then                   {if transactions commits}
34:      apply t.ws with version SC to database
35:   SC ← SC + 1                                {expose snapshot to clients}
36:   send(outcome) to c                         {return outcome to client}
37:   PENDING ← PENDING ⊖ (c, t, ⋆)
38: function certify(t)                          {used in line 13}
39:   for i ← t.st[p] to PSC do                  {for all concurrent transactions}
40:      if WS[i] ∩ t.rs ≠ ∅ or                  {if some intersection...}
41:         (t is global and RS[i] ∩ t.ws ≠ ∅) then
42:         return abort
43:   PSC ← PSC + 1                              {no intersection, next pending snapshot}
44:   RS[PSC] ← items(t.rs)                      {keep track of readset}
45:   WS[PSC] ← items(t.ws)                      {keep track of writeset}
46:   return commit
```

then it can be serialized after every t_j in σ. To ensure that t_i can be serialized before transactions in σ, servers in p also check that t_i's writeset does not intersect the readsets of transactions in σ.

With the new certification test, in the execution of example 1, both t_i and t_j will fail certification: t_j will fail certification at P_x since its writeset intersects the t_i's readset; t_i will fail certification for the same reason at P_y.

D. Algorithm in detail

Algorithm 3 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 now an array of snapshots, one for each partition (line 4). 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 subsequent reads to the same partition observe a consistent view. At commit time (lines 18–20), transactions are broadcast for certification to all partitions concerned by the transaction (line 18). The client then waits for the transaction's outcome (line 21).

Algorithm 4 shows the server side. When local transaction t is delivered, it is certified and appended to the queue of *PENDING* transactions (lines 14–15). The order of the transactions in queue *PENDING* follows the delivery order, and it is the same on every server within the same partition. Queue *PENDING* is consumed in order; when t is at the head of *PENDING* it can be finished (lines 20–22). To finish t, we proceed as usual: 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 32–37). As opposed to Algorithm 2, we now keep track of two separate counters: SC and PSC. PSC is incremented whenever some transaction is certified and enters the *PENDING* queue. SC is incremented whenever a transaction is taken from the *PENDING* queue and is applied to the database. Therefore, counter SC is incremented only when a transaction has finished and created a new snapshot to be exposed to clients.

The delivery of a global transaction t requires additional steps: it is certified (line 14), appended to the *PENDING*

queue (line 15), and the outcome of the local certification test is exchanged between servers in different partitions (lines 16–17). Servers keep track of the received votes in a set called *VOTES* (lines 18–19). Global transaction t can be finished when it is at the head of the *PENDING* queue (line 23), and enough votes have been received (lines 24–25). 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 decided as follows: if every partition voted for committing t, then it is committed, otherwise it is aborted (lines 26–28).

E. 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 serious disadvantage with respect to the original deferred update replication approach, where read-only transactions are serialized without certification. In the following we describe how global transactions can build a consistent global view of the database. Consequently, read-only transactions do not need to be certified (see [25] for more details).

Assume that a global transaction t reads from snapshots SC_x and SC_y, from partitions P_x and P_y, respectively. There are two reasons why t may see an inconsistent view of the database: (a) SC_x contains at least one global transaction t_i that is not included in SC_y. (b) Both SC_x and SC_y contain the same committed global transactions, but t sees transactions t_i and t_j in different orders in SC_x and SC_y.

To address case (a), servers build a vector of snapshot identifiers, one per partition, which together form a consistent global view of the database. When t issues the first read, instead of receiving the SC for the partition, it receives a vector of SCs, one SC per partition. This vector corresponds to a set of consistent snapshots, a *global snapshot*, and is used by t for all future read operations.

A server builds the vector of snapshot ids as follows. The vector is initially filled with zeros, as the first global snapshot contains no global transactions. When a global transaction t commits, servers communicate their local SC create by t's commit—exchanging the SCs can be done by a background task and piggybacked in vote messages of global transactions. With the SC for each committed global transaction, a server can update the vector of SCs with a more recent global snapshot: When the server observes that the same set of transactions has been committed in all partitions, the vector is updated with the corresponding SCs.

With regards to case (b), we note that t can only tell the order of t_i and t_j in a snapshot if they write the same data item in the snapshot. It follows from Algorithm 4 that if two concurrent global transactions t_i and t_j write a common data item, then at most one of them can commit, and consequently case (b) cannot happen.

F. 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. Since atomicity is guaranteed within a partition only, it may happen that some partitions deliver t's termination request while others do not. 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.

G. Correctness

In the following, we argue that any execution of S-DUR is *serializable*, that is, it is equivalent to a sequential execution involving the same transactions.

We start by introducing three definitions involving transactions t_i and t_j.

Definition 1. t_i is *serialized* before t_j if there is a serial execution in which t_i appears before t_j.

Definition 2. t_i and t_i *intersect* if and only if $readset(t_i) \cap writeset(t_j) \neq \emptyset$.

Definition 3. t_i and t_j are *concurrent* at partition P_x if they overlap in time at P_x. If t_i and t_j are not concurrent at P_x, then either t_i *precedes* t_j or t_j *precedes* t_i at P_x.

Traditional DUR ensures that at each partition, local transactions are serializable. From the certification test, the delivery order of transactions in a partition defines one serial execution equivalent to the real one. Since no two local transactions from different partitions intersect, any serial execution that (a) is a permutation of all transactions and (b) does not violate the delivery order of each partition is equivalent to the actual execution, and therefore every execution of local transactions only is serializable.

With global transactions, however, the above does not hold since global transactions may intersect with local transactions in multiple partitions. In order to show that S-DUR guarantees serializable executions with both local and global transactions, we introduce a few facts about the algorithm. Hereafter, ST_x^i and SC_x^i are the snapshot and the delivery order of transaction t_i at partition P_x, respectively.

Fact 1. If t_i passes certification at P_x, then it can be serialized anywhere after ST_x^i, up to SC_x^i.

To see why, note that t_i is certified against each t_j that committed after t_i started (otherwise it would be in t_i's snapshot SC_x^i) and finished before t_i (otherwise t_i would not know about t_j). From the certification test, t_i and t_j do not intersect, and thus t_i can be serialized before or after t_j.

Fact 2. If t_i and t_j are concurrent at P_x, then they can be serialized in any order.

Fact 2 is a consequence of Fact 1. Since t_i and t_j are concurrent at P_x, they overlap in time, and it must be that $ST_x^j < SC_x^i$ and $ST_x^i < SC_x^j$. Without loss of generality, assume that $SC_x^j < SC_x^i$. Obviously, t_j can be serialized before t_i. Transaction t_i can be serialized before t_j since, from Fact 1, t_j can be serialized at SC_x^j, t_i can be serialized at ST_x^i, and $ST_x^i < SC_x^j$.

Fact 3. The lifespan of a committed transaction in every two partitions in which it executes must overlap in time.

This follows from the fact that a transaction only commits in a partition after receiving the votes from all other partitions in which it executes.

We now proceed with a case analysis and show that for any interleaving involving global transactions t_i and t_j, there is a serial execution that is equivalent to the real execution, and therefore S-DUR is serializable (see Figure 2).

- *Case 1.* t_i precedes t_j in all partitions. Then trivially t_i can be serialized before t_i.
- *Case 2.* t_i precedes t_j in partition P_x and they are concurrent in some partition P_y. From Fact 2, t_i can be serialized before t_j in P_y.
- *Case 3.* t_i and t_j are concurrent in all partitions. Then, from Fact 2, they can be serialized in any order at every partition.
- *Case 4.* t_i precedes t_j in partition P_x and t_j precedes t_i in partition P_y. This case is impossible from Fact 3.

H. 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—although as we show in Section VI, the difference is very small. 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 be conservative and abort both transactions. This is similar to deferred update replication algorithms that rely on atomic commit to terminate transactions [18].

V. Implementation and optimizations

We use Ring Paxos [16] as our atomic broadcast primitive. There is one instance of Ring Paxos per partition. Servers log delivered values on disk, as part of the atomic broadcast execution. Therefore, the committed state of a server can be recovered from the log. Our prototype differs from Algorithm 4 in the following aspects:

- Each client connects to a single server only and submits all its read requests to this server. When the server receives a request for a key k that is not stored locally, the server routes it to a server in the partition that stores k.
- Our implementation reduces the number of vote messages exchanged for global transactions. Only one designated replica in a partition sends vote messages. If the other replicas suspect the failure of the assigned replica, they also send their votes and choose another replica as responsible for propagating the partition's votes.
- We use bloom filters to efficiently check for intersections between transaction readsets and writesets. Every server needs to keep writesets of both local and global transactions that executed in the past. The implementation keeps track of only the past K writeset bloom filters, where K is a configurable parameter of the system.[3]
- The implementation broadcasts transactions in small batches. This is essentially the well-known *group commit* optimization in centralized databases. In the case of DUR and S-DUR, it amortizes the cost of the atomic broadcast primitive over several transactions.

VI. Performance evaluation

In this section we assess the performance of scalable deferred update replication under different workloads. We

[3]There are two more advantages in using bloom filters: (1) bloom filters have negligible memory requirements; and (2) they allow us to send just the hashes of the readset when broadcasting a transaction, thus reducing network bandwidth. However, using bloom filters results in a small number of transactions aborted due to false positives.

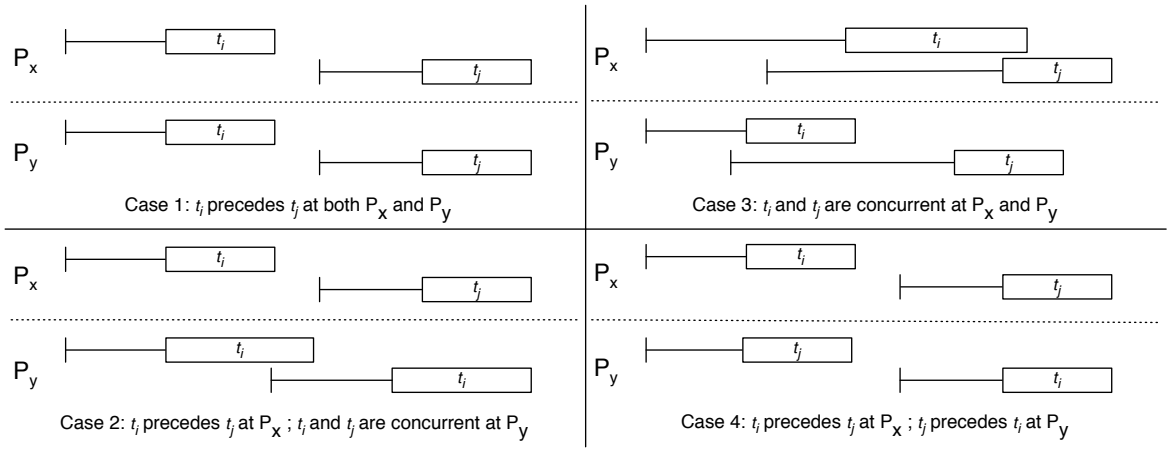

Figure 2: Interleaved executions of transactions (Cases 1–3 are possible under S-DUR but not Case 4).

look into throughput, latency and abort rate of S-DUR and compare them to standard DUR.

A. Setup and benchmarks

We ran the experiments in a cluster of Dell SC1435 servers equipped with two dual-core AMD-Opteron 2.0 GHz processors and 4 GB of main memory, and interconnected through an HP ProCurve2900-48G Gigabit Ethernet switch. Replicas are attached to a 73 GB 15krpm SAS hard-disk and perform asynchronous writes.

We evaluated the performance of S-DUR using four different workloads, which we summarize in Table I. Unless stated otherwise, we use three servers per partition, which means there are a total of three replicas for each data item. Clients were evenly distributed across servers and ran on separate machines. Global transactions access items stored in two different partitions.

Type	Reads (ops)	Writes (ops)	Key size (bytes)	Value size (bytes)	DB size (items/partition)
A	4	4	4	4	3M
B	2	2	4	1K	1M
C	8	0	4	4	3M
D	4	0	4	1K	1M

Table I: Workload types - Varying number of read and write operations per transaction (ops), and key and value sizes (bytes).

B. Throughput

In Figure 3 we show the normalized throughput of S-DUR while varying the percentage of global transactions. Throughput is normalized over the performance of standard DUR with three replicas. We repeated the experiment for 2, 4 and 8 partitions, under all workloads. Shaded areas in Figure 3 correspond to setups in which the protocol scales linearly with the number of partitions (or replicas, for DUR).

For comparison, we also show deferred update replication with 6, 12 and 24 replicas (last three columns in the graphs). The experiments confirm that update transactions in DUR do not scale as replicas are added to the system. In fact, increasing the number of replicas resulted in almost no performance improvement for update transactions in DUR.

Under update transactions (workload types A and B), S-DUR's throughput scales linearly with the number of partitions when all transactions are local. For instance, with workload A and 0% of global transactions, doubling the number of partitions also doubles the total throughput. The performance of S-DUR degrades as the percentage of global transactions increases. This is due to the additional cost of exchanging votes and to the fact that global transactions slow down local transactions (i.e., if a local transaction t_i is delivered after a global transaction t_j, then t_i will terminate after t_j terminates). However, with 10% or less global update transactions, S-DUR in a configuration with eight partitions (i.e., 24 nodes) can execute four times more update transactions that DUR with the same number of nodes. With 100% of global transactions, S-DUR performs slightly worse than DUR in most cases, and better in one scenario (8 partitions in workload A). The break-even point is between 40% and 60% of global transactions, in which case DUR and S-DUR execute the same number of transactions per time unit in all our setups.

We now consider read-only transactions (workload types C and D). As opposed to update transactions, we observe that in all configurations read-only transactions scale linearly with the number of partitions, both in DUR and in S-DUR. With local transactions only, DUR and S-DUR have almost identical performance. S-DUR's throughput degrades as the percentage of global transactions increases. This is a limitation of our implementation as requests not stored by a server are routed to the appropriate server. Consequently, for global read-only transactions, read operations for keys

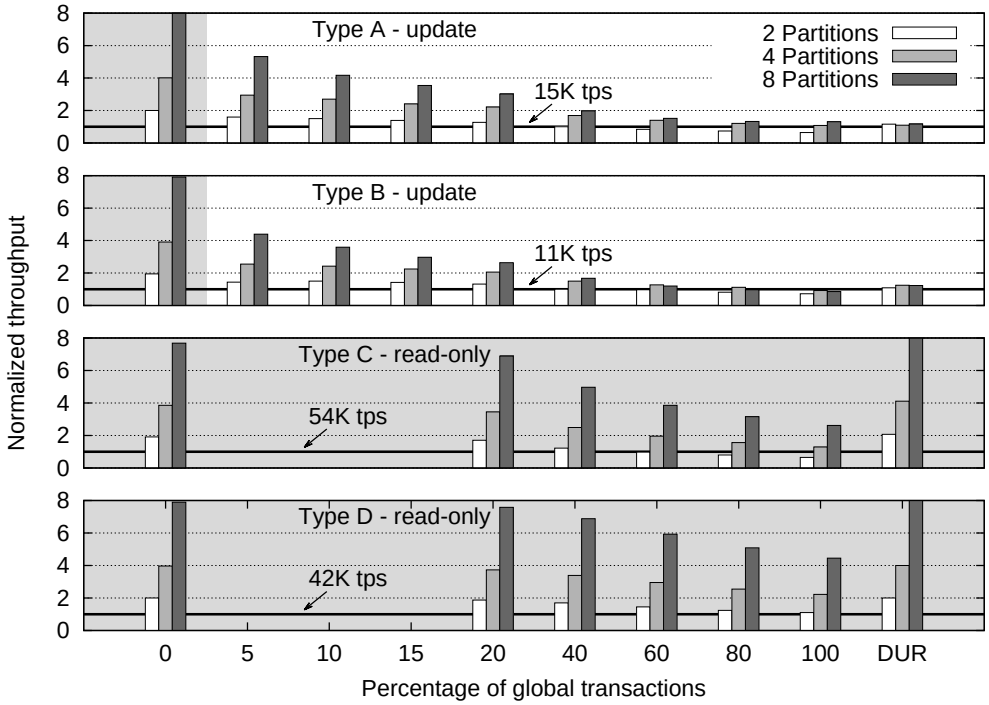

Figure 3: Normalized throughput versus percentage of global transactions.

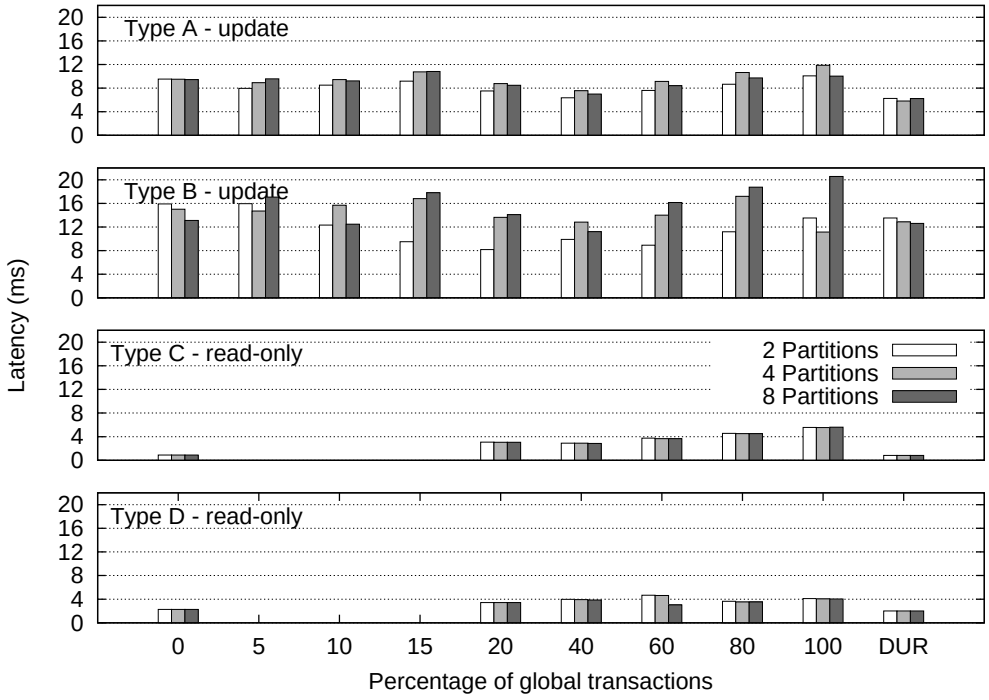

Figure 4: Latency versus percentage of global transactions.

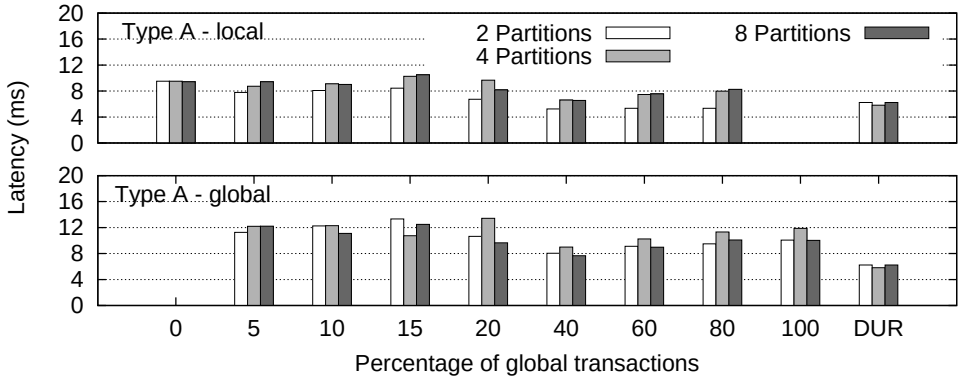

Figure 5: Latency of global and local update transactions versus percentage of global transactions.

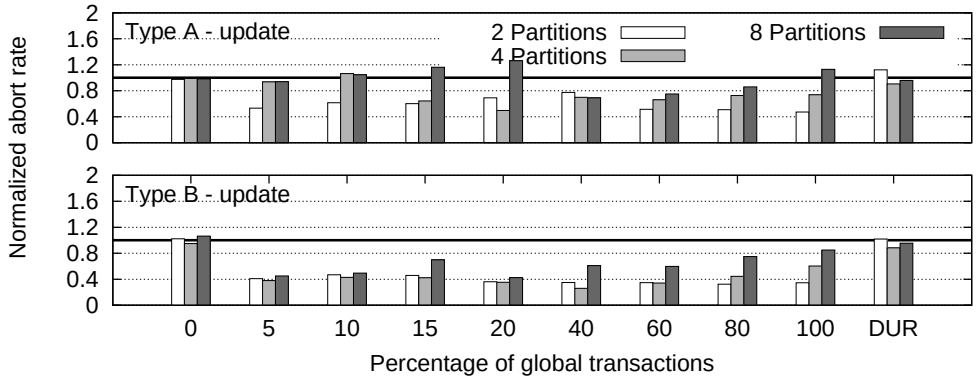

Figure 6: Normalized abort rate versus percentage of global transactions.

replicated in a different partition require one more round trip. In fact, workload type C degrades more quickly than type D because transactions in workload D perform fewer read operations (even if workload B reads larger values).

C. Latency

Figure 4 shows the latency for the experiments reported in the previous section (global and local transactions combined). For workload type A, all S-DUR configurations show similar latency, in most cases a few milliseconds higher than DUR. Latency values for workload type B present more variation, although they tend to increase with the percentage of global transactions. Latency of global read-only transactions in S-DUR, for both workloads, is consistently higher than in traditional DUR. This is not surprising since global read-only transactions involve communication among partitions, which is not the case with DUR where the database is fully replicated in each replica. This difference tends to decrease for larger data items (workload D).

Figure 5 compares the latency of global transactions with the latency of local transactions. In all cases, global transactions are 2 to 4 milliseconds slower than local transactions. This is due to the extra communication needed to terminate global transactions.

D. Abort rate

Figure 6 reports the normalized abort rates of update transaction (read-only transactions never abort). For all configurations, we observed (a) less than 1% of aborts and (b) fewer aborts in workload type B than in workload type A since type B transactions perform fewer operations.

In general, aborts are influenced by a combination of two factors: decreasing the number of global transactions results in higher throughput and thus more aborts; increasing the number of global transactions tends to augment aborts since certification of global transactions is more restrictive.

With local transactions, abort rate remains constant with the number of partitions, even though more partitions accommodate higher throughput. This happens because the database grows proportionally to the number of partitions. With global transactions, aborts tend to increase with the number of partitions since the throughput of each partition is a fraction of the throughput of DUR and thus there is less contention within partitions.

VII. Related work

A number of protocols for deferred update replication where servers keep a full copy of the database have been proposed (e.g., [1], [11], [14], [17], [19]). As explained in Section III-C, in the case of full database replication, the scalability of update transactions is inherently limited by the number of transactions that can be ordered, and by the number of transactions that can be certified and applied to the local database at each site.

Some protocols provide partial replication (e.g., [22], [26], [27]) to improve the performance of deferred update replication. This alleviates the scalability problem in that only the subset of servers addressed by a transaction applies updates to their local database. However, these protocols require transactions to be atomically broadcast to all participants. When delivering a transaction, a server may discard those transactions that do not read or write items that are replicated locally.

Alternatively, some protocols implement partial database replication using atomic multicast primitives (e.g., [10], [23]). Fritzke et al. [10] describe a protocol where each read operation of a transaction is multicast to all concerned partitions, and write operations are batched and multicast at commit time to the partitions concerned by the transaction. P-Store [23] implements deferred update replication with optimizations for wide-area networks. Upon commit, a transaction is multicast to the partitions containing items read or written by the transaction; partitions certify the transaction and exchange their votes, similarly to S-DUR. On the one hand, multicasting a transaction to all involved partitions has the advantage of ordering certification across partitions and aborting fewer transactions (e.g., it would avoid the problematic execution depicted in Figure 1). On the other hand, genuine atomic multicast is more expensive than atomic broadcast in terms of communication steps [21], and thus transactions take longer to be certified.

Many scalable storage and transactional systems have been proposed recently. Some of these systems resemble our implementation of scalable deferred update replication in that they expose a similar interface to clients, based on read and write (i.e., *get* and *put*) operations. Some storage systems (e.g., [8], [30], [4]) guarantee some form of relaxed consistency, *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. COPS [15] is a wide-area storage system that ensures a stronger version of causal consistency, which in addition to ordering causally related write operations also orders writes on the same data items. Walter [28] 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. S-DUR also partitions the database to leverage the capacity of independent clusters to increase throughput, and enforces a global total order on the transactions that update multiple partitions. However, S-DUR overall provides stronger guarantees (i.e., serializability) than Walter. Similarly to Walter, S-DUR also uses a vector of timestamps to build global snapshots. The vectors are built in different ways though, since in S-DUR transactions always see a serializable committed state of the database, while in Walter transactions may see committed transactions in different orders.

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.

Google's Bigtable [6] and Yahoo's Pnuts [7] 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. However, none of these databases offer full transactional support.

Rao et al. [20] proposed a storage system similar to the approach presented here in that it also uses several instances of Paxos [13] to achieve scalability. Differently than S-DUR, however, it does not support transactions across multiple Paxos instances.

VIII. Conclusion

This paper proposes an extension of the deferred update replication approach. Deferred update replication is implemented by several database protocols due to its performance advantages, namely, good throughput in the presence of update transactions and scalability under read-only transactions. Scalable Deferred Update Replication, the paper's main contribution, makes the original approach scale under both read-only transactions and local update transactions. Under mixed workloads, with global and local update transactions, system throughput depends on the percentage of global transactions. In the worst case (i.e., 100% of global update transactions), performance is similar to the traditional deferred update replication.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar*, 1997.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, 2007.

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

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

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] 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 Trans. Comput. Syst.*, 26(2):1–26, 2008.

[7] 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. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[8] 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 Oper. Syst. Rev.*, 41(6):205–220, 2007.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[10] Jr. Fritzke, U. and P. Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 284–291, 2001.

[11] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'2000)*, Cairo, Egypt, 2000.

[12] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[13] L. Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32, 2001.

[14] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *International Conference on Management of Data (SIGMOD)*, Baltimore, Maryland, USA, 2005.

[15] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011.

[16] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *International Conference on Dependable Systems and Networks (DSN)*, 2010.

[17] 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*, 23(4):375–423, 2005.

[18] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar*, 1998.

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

[20] J. Rao, E.J. Shekita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.

[21] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *Proceedings of the 9th International Conference on Distributed Computing and Networking (ICDCN)*, pages 147–157. Springer, 2008.

[22] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS)*, pages 81–93. Springer, 2006.

[23] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 214–224. IEEE, 2010.

[24] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[25] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. Technical report, University of Lugano, 2012.

[26] D. Serrano, M. Patino-Martinez, R. Jiménez-Peris, and B. 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)*, pages 290–297. IEEE, 2007.

[27] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of the 1st International Symposium on Network Computing and Applications (NCA)*, pages 298–309. IEEE, 2001.

[28] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 385–400, 2011.

[29] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, SE-5:188–194, 1979.

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