High Performance Deferred Update Replication

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Daniele Sciascia

under the supervision of
Prof. Fernando Pedone

March 2015

Dissertation Committee

Prof. Mehdi Jazayeri	Università della Svizzera Italiana, Switzerland
Prof. Olaf Schenk	Università della Svizzera Italiana, Switzerland
Prof. Jose Enrique Armendariz-Inigo	UPNa, Spain
Prof. Luis Rodrigues	INESC-ID, Portugal
Prof. Willy Zwaenepoel	EPFL, Switzerland

Dissertation accepted on March 2015

Prof. Fernando Pedone
Research Advisor
Università della Svizzera Italiana, Switzerland

Stefan Wolf, Igor Pivkin
PhD Program Directors

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Daniele Sciascia
Lugano, March 2015

To Laura

An expert is a man who has made
all the mistakes which can be
made in a very narrow field.

Neils Bohr

Abstract

Replication is a well-known approach to implementing storage systems that can tolerate failures. Replicated storage systems are designed such that the state of the system is kept at several replicas. A replication protocol ensures that the failure of a replica is masked by the rest of the system, in a way that is transparent to its users. Replicated storage systems are among the most important building blocks in the design of large scale applications. Applications at scale are often deployed on top of commodity hardware, store a vast amount of data, and serve a large number of users. The larger the system, the higher its vulnerability to failures. The ability to tolerate failures is not the only desirable feature in a replicated system. Storage systems need to be efficient in order to accommodate requests from a large user base while achieving low response times. In that respect, replication can leverage multiple replicas to parallelize the execution of user requests.

This thesis focuses on Deferred Update Replication (DUR), a well-established database replication approach. It provides high availability in that every replica can execute client transactions. In terms of performance, it is better than other replication techniques in that only one replica executes a given transaction while the other replicas only apply state changes. However, DUR suffers from the following drawback: each replica stores a full copy of the database, which has consequences in terms of performance.

The first consequence is that DUR cannot take advantage of the aggregated memory available to the replicas. Our first contribution is a distributed caching mechanism that addresses the problem. It makes efficient use of the main memory of an entire cluster of machines, while guaranteeing strong consistency.

The second consequence is that DUR cannot scale with the number of replicas. The throughput of a fully replicated system is inherently limited by the number of transactions that a single replica can apply to its local storage. We propose a scalable version of the DUR approach where the system state is partitioned in smaller replica sets. Transactions that access disjoint partitions are parallelized.

The last part of the thesis focuses on latency. We show that the scalable DUR-based approach may have detrimental effects on response time, especially when replicas are geographically distributed. The thesis considers different deployments and their implications on latency. We propose optimizations that provide substantial gains in geographically distributed environments.

Acknowledgements

I am truly grateful to my advisor Fernando Pedone. Fernando taught me a lot and has always been generous in guiding me with all his knowledge and experience. I admire him for his dedication towards his students. The only secret he has not shared with me is where all his energy, enthusiasm and patience come from.

I wish to thank the members of the dissertation committee, Mehdi Jazayeri, Olaf Schenk, Jose Enrique Armendariz-Inigo, Luis Rodrigues and Willy Zwaenepoel for the time they dedicated to this thesis and for their feedback.

I am also grateful to Willy Zwaenepoel for hosting me at EPFL, where I felt part of his LABOS team. I spent most of the time there working with Jiaqing Du, and remotely with Sameh Elnikety. Thank you all for the fun and fruitful summer.

I thank the current and former members of the Distributed Systems Lab: Marco Primi, Nicolas Schiper, Amir Malekpour, Ricardo Padilha, Parisa Jalili Marandi, Eduardo Bezerra, Leandro Pacheco, Samuel Benz, Odorico Mendizabal, Daniel Cason and Tu Dang.

For most of the time, I have been commuting to Lugano by train. Many friends on the train, "gli amici del treno", made my daily ride more fun: Marcello Scipioni, Giacomo Inches, Davide Eynard, and Fernanda Gallo.

I wish to thank my parents and my brother, Angela, Salvatore, and Claudio. It is nice to know that there is always somebody I can rely on. Finally, thanks to my partner Laura, who stood by me throughout this experience.

Preface

The result of this research appears in the following papers:

D. Sciascia and F. Pedone. *RAM-DUR: In-Memory Deferred Update Replication.* 31st International Symposium on Reliable Distributed Systems (SRDS 2012)

D. Sciascia, F. Pedone and F. Junqueira. *Scalable Deferred Update Replication.* 42nd International Conference on Dependable Systems and Networks (DSN 2012)

D. Sciascia and F. Pedone. *Geo-replicated storage with scalable deferred update replication.* 43rd International Conference on Dependable Systems and Networks (DSN 2013)

And was later extended in the following publications:

A. Tomic, D. Sciascia and F. Pedone. *MoSQL: An Elastic Storage Engine for MySQL.* 28th ACM Symposium on Applied Computing, DADS Track (SAC 2013)

L. Pacheco, D. Sciascia and F. Pedone. *Parallel Deferred Update Replication.* 13th IEEE International Symposium on Network Computing and Applications (NCA 2014)

Outside of the main scope of this research, I worked on the design of a State Machine Replication protocol based on physical clocks. This work has been published in the following paper:

J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel and F. Pedone *Clock-RSM: Low-Latency Inter-Datacenter State Machine Replication Using Loosely Synchronized Physical Clocks.* 44th International Conference on Dependable Systems and Networks (DSN 2014)

Contents

Figures

Tables

Algorithms

Chapter 1

Introduction

In recent years we have witnessed increased demand for reliable storage systems that run on top of cheap commodity hardware. Applications at scale rely on servers that are distributed over several data centers, located in different geographical areas to provide high availability and low response times to their clients. These applications store a vast amount of data and serve a large number of users. As systems become larger, failures are more frequent. Failures may originate from human errors, single machine failures, network equipment and communication failures, and sometimes disasters that shutdown data centers entirely. One way to cope with such failures is replication, where several replicas cooperatively store the state of the system. In the event of failures, replication protocols can mask the failure of replicas to ensure continued service. The main challenge for replication protocols is to provide the illusion of operating a single high-performance and fault-tolerant system.

1.1 Context

Storage systems have evolved rapidly over the past two decades. The internet industry is one of the major driving forces behind the evolution of storage systems. The main reason is that web applications at scale store a large amount of data (e.g., user profiles, posts, photos, shopping carts) and serve a massive number of users spread around the globe.

Early web applications in the 90's were usually built using a scripting language in combination with a centralized SQL database. This kind of solution gave enough power to build web applications that served static and dynamically generated contents. Static content was simply stored on a file system. While dynamic content was usually generated based on the user that is con-

nected to the application, and the particular actions performed by the user. Centralized relational databases were flexible enough to capture most storage requirements of web application builders: they were fast enough and stored data reliably. Moreover, the SQL language offered powerful features such as transactions, joins, indexes and views.

In the early 2000s the largest internet companies started to build their own storage solutions from scratch [12; 14; 20; 30; 35]. Their platforms had to sustain a rapid growth in users and data. Three major trends arose: (1) The use of commodity hardware and the scale of these applications called for storage systems that were available in the event of failures and outages. Failures are costly in that they have immediate consequences on the income generated by the applications (e.g. stopping sales, not serving advertisement). (2) SQL was not necessarily the best language to handle the data. Simpler interfaces such as key-value and columnar stores became popular. These interfaces usually did not provide transactions, in some cases they provided limited forms of atomicity. (3) Relaxed consistency criteria reduce latency and can sustain higher transaction/operation rates. For instance, *eventual consistency* [69] achieves low latency and high throughput. However, data may diverge under concurrent access or network partitions, with the complexity of reconciliation pushed to the application layer. For some applications, or parts of the application, reconciling data is relatively easy (e.g., the operations are commutable).

More recently, many large web applications have moved to the cloud, where they share large computing infrastructures. Companies can simply buy computing power and storage by the hour. Building infrastructures that are distributed over several geographical locations is not only feasible, it can be done without upfront investments. The focus now is on storage systems that can be distributed over several geographically distributed datacenters. There are two main reasons for this trend: (1) Storing data close the user reduces the perceived latency. Web applications have typically a large fan-out. A client request to a web application typically translates to tens of requests to backend web services and storage systems [19]. These requests are executed in parallel and the overall latency depends on the slowest component, or those components with the highest latency variability. (2) Geo-replication tolerates disasters, such as the failure of an entire datacenter. Cloud computing vendors offer several storage solutions, such as Amazon's S3 and SimpleDB, or Microsoft Azure's Blobs and Tables. These solutions provide a variety of interfaces, from simple key-value, to simplified SQL-like languages.

1.2 Motivation

This thesis focuses on distributed and replicated database systems that guarantee strong consistency. This type of systems have recently seen a renovated interest in both academia [3; 33; 55; 66] and industry [7; 15; 59]. Strongly consistent storage systems offer the illusion and semantics of a single non-replicated system. These systems mask failures and do not allow replicas to diverge, leading to simpler application designs. However, building a system that maintains low-latency and high-throughput with strong consistency guarantees, despite machine failures, is challenging. Replicas have to synchronize in order to avoid inconsistencies that would break the illusion of a single high performance and fault-tolerant database.

In this context, we revisit Deferred Update Replication, a well-known approach to building fault-tolerant data management systems. This approach has been mainly studied in the context of group communication-based database replication (e.g., [47; 52]), where fault-tolerant group communication primitives are employed to order transactions in multiple replicas. This approach has several advantages in terms of performance, although most existing solutions have focused on full replication, which is not always acceptable. Full replication requires each replica to hold a complete copy of the database and limits the scalability of update transactions since every transaction must be executed by every replica. More recently, partial replication protocols based on deferred update replication have been proposed. Though the existing solutions typically make use of atomic multicast primitives [6; 27; 55], which require more communication steps and are therefore more expensive.

1.3 Contributions

The major goal of this thesis is to revisit the Deferred Update Replication approach in light of current architectural and deployment trends. Where possible, we preferred solutions that resulted in the same guarantees provided by the original approach, while providing increased performance in terms of latency, throughput, or both. We next outline the three main contributions of this thesis.

In-memory execution in Deferred Update Replication. RAM-DUR is an extension of the Deferred Update Replication (DUR) approach that allows for fast in-memory execution of transactions. As pointed out in [64], given the current cost of main memory, many workloads should be considered as in-memory only.

The main assumption in RAM-DUR is that retrieving items remotely (through the network) is more efficient than accessing the local disk. We devised a protocol that leverages this assumption. Servers only store a subset of the dataset and retrieve items from remote servers when needed. The resulting protocol still meets the same consistency level of DUR.

Scalable Deferred Update Replication. S-DUR improves the overall scalability of DUR. The assumption here is that the database can be partitioned and that most transactions access contents from one partition (local transactions), while a minority of transactions access two or more partitions (global transactions). We devised a protocol in which each partition is fully replicated in a small set of servers. Local transactions are handled as in the baseline DUR protocol, while global transactions require only one additional inter-partition roundtrip. The resulting protocol guarantees the same consistency level of DUR. Moreover, the S-DUR protocol is less expensive compared to previous partial-replication approaches that are based on atomic multicast [6; 27; 55].

Deferred Update Replication in geographically distributed environments. Large scale web services and applications are often deployed over several geographically distributed datacenters. The main goal of geographical replication is to keep data "close" to the actual user to reduce the perceived latency. Although desirable, efficient and strongly consistent replication over wide-area networks is hard to achieve due to the high latency of wide-area links. One solution is to deploy S-DUR partitions over multiple geographically distributed datacenters. However, even if local transactions are cheaper than global ones, the following problem arises in mixed workloads. Within a partition, the commitment of transactions is serialized. Therefore local transactions may be delayed by global transaction that are pending due to the extra roundtrip. To overcome this limitation we propose two different termination strategies, based on the idea of reordering transactions.

1.4 Outline

The following chapters are structured as follows. Chapters 2 and 3 provide useful background on this work. Chapter 2 states the system model and the definitions that are assumed in the rest of the thesis. Chapter 3 illustrates the Deferred Update Replication protocol that will serve as a baseline in comparing the protocols developed in the following chapters. Chapter 4 presents the first

contribution, an optimized version of the baseline protocol of Chapter 3, with support for in-memory execution of transactions. Chapter 5 presents S-DUR, a scalable partial replication protocol, based on deferred update replication. Chapter 6 considers the problem of partial replication in the context of wide-area networks, and proposes an optimization and two alternative termination protocols that, in this context, outperform the S-DUR protocol of Chapter 5. Chapter 7 concludes the thesis and proposes future work. Appendices A, B and C argue about the correctness of the proposed protocols.

Chapter 2

System Model and Definitions

In this chapter, we first review the system model we assume and present the definitions we will use throughout the thesis.

2.1 Processes and failures

We consider a system composed of an unbounded set $C = \{c_1, c_2, ...\}$ of client processes and a bounded set $S = \{s_1, ..., s_n\}$ of database server processes. We assume the crash-recovery model in which processes may fail by crashing but never perform incorrect actions (i.e., no Byzantine failures). Correct processes are *non-faulty*. A correct process is operational "forever" and can reliably exchange messages with other correct processes. Notice that in practice, "forever" means long enough for some useful computation to be performed. Faulty processes lose all of their volatile state. However, processes have access to a local stable storage that can be used for storing and retrieving state that survives failures. The state on stable storage can be used for recovery.

2.2 Communication

Processes communicate through message passing and have no access to a shared memory or a global clock. Processes communicate using either one-to-one or one-to-many communication.

2.2.1 One-to-one 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.

2.2.2 One-to-many communication

Server processes may use one-to-many communication that relies on total order broadcast [21]. One-to-many communication uses primitives *abcast*(m) and *adeliver*(m). Total order broadcast ensures the following properties:

- **Validity.** If a correct process broadcasts m, then all correct processes eventually deliver m.
- **Agreement.** If a process delivers m, then all correct processes eventually deliver m.
- **Integrity**. For any message m, every process delivers m at most once, and only if m was previously
- **Total Order**. If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

2.3 Synchrony assumptions

Solving total order broadcast requires some synchrony assumptions [10; 26]. We assume that the system is *partially synchronous* [24], that is, it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [24], and it is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown. Moreover, in order to ensure liveness, we assume that *after* GST all remaining processes are correct.

2.4 Database and transactions

We assume a multiversion database composed of a set of data items $D = \{x_1, x_2, \dots\}$. Each data item is a tuple $\langle k, v, ts \rangle$, where k is a key, v its value,

and ts its version. 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 $\langle id, st, rs, ws \rangle$ where id is a unique identifier for t, st is the database snapshot version seen by t, rs is the set of data items read by t, denoted *readset*(t), and ws is the set of data items written by t, denoted *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. In the algorithms, we use *dot notation* to access the attributes of a transaction, for instance $t.rs$ denotes the readset of transaction t.

2.5 Serializability

Several consistency criteria for database transactions exist [1]. In this thesis, we are mainly interested in *one-copy serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [8].

Chapter 3

Deferred Update Replication

Deferred Update Replication (DUR) is a well-established approach to fault-tolerant data management systems. The idea behind Deferred Update Replication 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. 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 [34].

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. In this chapter we present a protocol for DUR, which will serve as a baseline in the following chapters.

3.1 General idea

In Deferred Update Replication, 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.

The version of DUR we detail next closely resembles the Database State Machine Approach [47], and assumes a multiversion database at each replica to allow local termination of read-only transactions. This optimization was first introduced in [51].

3.2 The algorithm in detail

Algorithms 1 and 2 illustrate the technique for the client and server, respectively. Before submitting transaction operations, the client application first initializes the transaction using function *begin*() (lines 1 – 4). A read operation on item k starts by updating transaction t's readset (line 6, Algorithm 1) and 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 de-

Algorithm 1 Deferred Update Replication, client c

```
1:  function begin(t)
2:      t.rs ← ∅                                   {initialize readset}
3:      t.ws ← ∅                                   {initialize writeset}
4:      t.st ← ⊥                                   {initially transactions have no snapshot}
5:  function read(t, k)
6:      t.rs ← t.rs ∪ {k}                          {add key to readset}
7:      if (k, ⋆) ∈ t.ws then                      {if key previously written...}
8:          return v s.t. (k, v) ∈ t.ws            {return written value}
9:      else                                       {else, if key never written...}
10:         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: function write(t, k, v)
15:     t.ws ← t.ws ∪ {(k, v)}                     {add key to writeset}
16: function 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:         send(commit, t) to a chosen server s ∈ S
21:         wait until receive(outcome) from s
22:         return outcome                         {outcome is either commit or abort}
```

termine 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 17–18). If t is an update transaction, the client simply submits t to some server and waits for a response, either *commit* or *abort* (lines 20–22).

Servers maintain variables *SC* and *DB* (lines 2–3, Algorithm 2). *SC* has the latest snapshot created by server s. *DB* is a vector, where entry $DB[i]$ contains 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

Algorithm 2 Deferred Update Replication, server s

```
1:  Initialization
2:     SC ← 0                                          {snapshot counter}
3:     DB[...] ← ∅                                     {list of applied transactions}
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 DB                      {most recent version ≤ st}
7:     send(k, v, st) to c                             {return result to client}
8:  when receive(commit, t)
9:     abcast(t) to all servers in S
10: when adeliver(t)
11:    outcome ← certify(t)                            {outcome is either commit or abort}
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 DB[i].ws ∩ t.rs ≠ ∅ then                  {if some intersection...}
16:          return abort                              {transaction must abort}
17:    SC ← SC + 1                                     {here no intersection: one more snapshot}
18:    DB[SC] ← t                                      {keep track of committed writeset}
19:    return commit                                   {transaction must commit}
```

(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).

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).

3.3 DUR and other replication techniques

Several database replication protocols are based on deferred update replication (e.g., [2; 32; 38; 44; 47]), 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 [56]. Thus, adding servers does not increase the throughput of update transactions; throughput is limited by what one replica can execute. With primary-backup replication [63], 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.

Chapter 4

In-Memory Deferred Update Replication

In typical data management systems, servers strive to store the database in memory (i.e., a cache) when executing transactions to avoid reading from the on-disk database image and thus improving performance. Disk writes are needed at commit time to ensure durability; however, disk writes are typically sequential, in the form of log append operations [28]. Since caching the whole database (or a large portion of it) in memory is only possible if the data fits in the memory of the server, some data management architectures recur to partitioning the database across multiple servers in order to increase the chances that the data assigned to each server (i.e., a partition) fits its main memory.

Partitioning schemes, however, sometimes restrict transaction execution to a single partition or sacrifice consistency of multi-partition transactions. Partitioning the database across multiple servers without restricting transaction execution and sacrificing consistency is possible with variations of deferred update replication (see Chapter 5 and 6). In all such protocols, however, multi-partition transactions have a higher response time than single-partition transactions due to additional communication between partitions during the certification of transactions — essentially, multi-partition transactions have to pass through a voting phase during termination to ensure that they are serializable across the system

This chapter introduces in-memory deferred update replication (RAM-DUR), an extension to deferred update replication where the database is partitioned among a set of servers. Ideally, the database will fit the aggregated memory of the compound, but not necessarily the main memory of any individual server. If a server needs a data item it does not cache, instead of retrieving the item from

disk, the server retrieves it from the cache of a remote server. The rationale behind RAM-DUR is that network access is faster than a local disk access. Differently from previous approaches, RAM-DUR does not increase the response time of transactions. Transactions are certified following the traditional deferred update replication procedure and there is no voting phase during transaction termination. RAM-DUR's key insight is a distributed cache mechanism that allows servers to cache portions of the database without sacrificing serializability.

4.1 General idea

RAM-DUR distinguishes between two types of servers: *core servers*, which execute the traditional deferred update replication, and *volatile servers* (*vnodes*), which store a subset of the database in memory only. The purpose of vnodes is to increase the performance of transaction execution by ensuring that data is always in memory. The system does not rely on vnodes for durability. Vnodes essentially implement a caching layer that ensures consistent execution of transactions. As we discuss in Section 4.5, transaction durability is guaranteed by the core servers.

RAM-DUR partitions database entries among vnodes based on their key, using a user defined partitioning scheme (such as range or hash partitioning). A vnode is the *owner* of all entries assigned to it as a result of the partitioning scheme. A vnode executes both read operations from the clients and *remote lookup* operations from other vnodes. A vnode v issues a remote lookup for key k to the owner of k when some client requests to read k and v does not store k locally (e.g., v is not the owner of k). In addition to its assigned entries, vnodes can cache any other entries, as long as memory is available. A vnode is required to store new versions of the keys it owns, but it may discard other entries, as long as consistency is not violated, as detailed next.

We define two operations to manipulate the local storage of vnodes: (1) $lookup(k, st)$, which returns a tuple $\langle key, value, version \rangle$ from the local storage of the vnode or from the owner of k (by remotely fetching the tuple); and (2) $apply(k, v, st)$, which stores item (k, v) with version st in the vnode's storage. The lookup request must return the largest version of k equal to or smaller than st. Figure 4.1 depicts a simplified view of the various storage abstractions of a vnode. Clients (i.e., applications) have access to data through the *read* and *write* primitives, which are implemented on top of *lookup* and *apply* primitives. A *lookup* will be translated into a local *retrieve* operation or a *remote-lookup*

operation, depending on whether the item read is cached or not, respectively; *apply* is implemented by a call to *store*. The *remote-lookup* request is submitted to the owner of the missing entry, which will translate it into a *lookup* to its local storage.

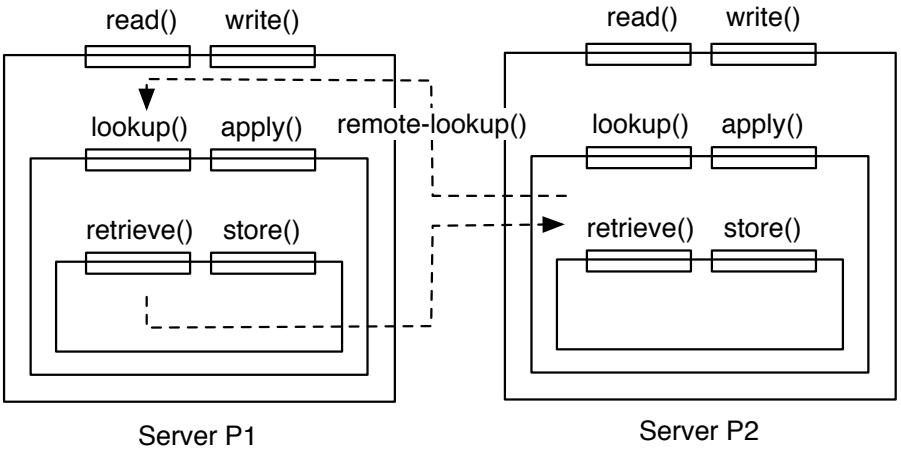

Figure 4.1. Simplified storage abstractions of a vnode.

4.2 Ensuring Serializability

In order to preserve serializability, *lookup* and *apply* have to follow a number of rules. We introduce these rules next and justify their need with a few executions. In all executions described next, vnode P_1 is the owner of data item x and vnode P_2 does not own x and does not have a cached version of x. Transaction t issues read operations on x against P_2.

Execution 1. Transaction t issues a read operation on x for version 11 against P_2 (see Figure 4.2). As a result, P_2 issues a remote lookup operation against P_1. When P_1 handles the request, its *SC* is 10 while P_2 had delivered snapshot 11 before starting t (recall from Algorithm 2 that *SC* is the latest snapshot created by a server). P_1 returns the largest version of x it knows about equal to or smaller than 11, which in this case is version 10. However, snapshot 11 may include an update to x, which should be visible to transaction t.

To avoid this case, we introduce Rule 1:

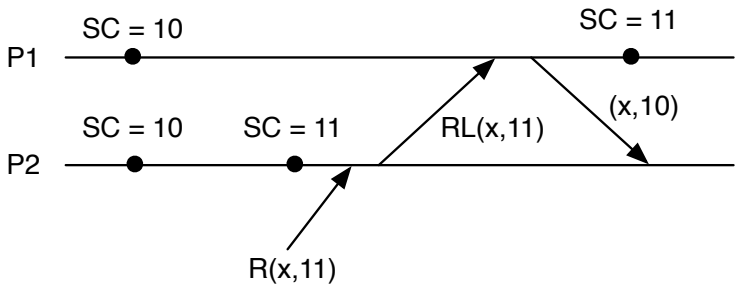

Figure 4.2. Problematic execution 1. Dots show the delivery of a transaction that created snapshot *SC*. R(x,i) is a read request for version *i* of data item x. RL(x,i) is a remote lookup request for version *i* of data item *x*.

- *Rule 1. A vnode can only reply to a remote lookup for version v if v is less than or equal to the vnode's SC.*

Execution 2. In this case (Figure 4.3), t issues a read operation for key x with version 10 against P_2. P_2 sends a remote lookup request to P_1 and then delivers snapshot 11, before receiving a reply from P_1. If snapshot 11 contains an updated version of x, P_2 is allowed to discard it (because P_2 is not the owner of x). Later P_2 receives a reply for x with version 10. To avoid future requests to x, P_2 caches x with version 10 locally. Suppose a later transaction t' reads x with version 11, P_2 returns x with version 10, i.e. the most recent x with version less than or equal to 11. This execution is not serializable because t' should see version 11 of x.

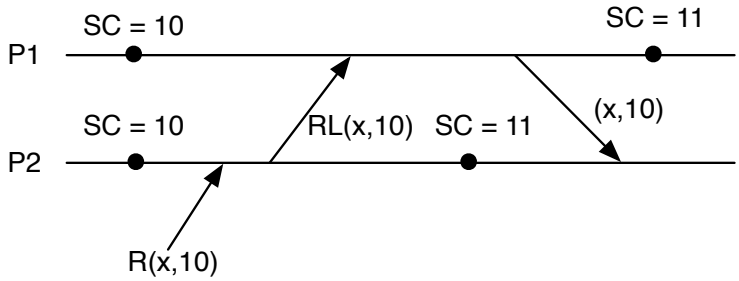

Figure 4.3. Problematic execution 2.

This case is excluded by Rule 2:

- *Rule 2. A vnode must not discard a delivered version of k if it has a pending remote request for k.*

Execution 3. Assume that P_2 is ahead in the execution by two snapshots (Figure 4.4). P_2 requests version 8 of key x. When P_1 receives the request it sends key x at version 8 to P_1. According to the rules defined so far, P_2 can cache $(x, 8)$ locally. However, a later transaction t with $st = 9$ might request $(x, 9)$, and because P_2 is ahead in the execution it already discarded version 9 and returns $(x, 8)$.

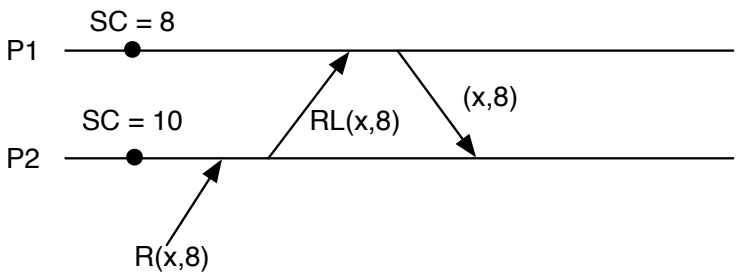

Figure 4.4. Problematic execution 3.

Rule 3 avoids this case.

- *Rule 3. A key can only be cached if the SC of the vnode handling a request for k is bigger than or equal to the SC of the requesting vnode. Also, k can be cached only if it is the newest version smaller than SC.*

Execution 4. Figure 4.5 shows an execution in which both P_1 and P_2 deliver snapshot 10. Right after delivering the snapshot, P_1 garbage collects item x with version 10. Later transaction t requests to read item x with version 10, and accordingly P_2 sends a remote request to P_1. Since version 10 of x was garbage collected, P_1 returns the most recent version less than 10, namely x with version 9.

Rule 4 applies to both cached and non-cached entries.

- *Rule 4. A vnode can garbage collect version v of item k only if it has already garbage collected all versions of k with version less than v. The owner of k must preserve at least one version (the latest one), while vnodes caching k can remove all versions.*

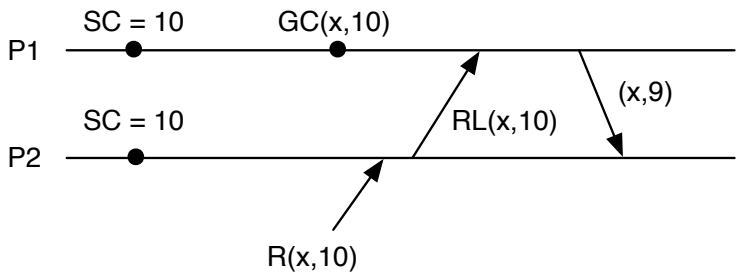

Figure 4.5. Problematic execution 4. GC(x,i) shows the garbage collection for version *i* of item *x*.

4.3 The algorithm in detail

RAM-DUR is an extension to the deferred update replication that does not require modifications to the client side. In this section we only discuss the protocol for vnodes, shown in Algorithm 3, while we assume clients follow Algorithm 1 (see page 13).

Read requests are executed at vnodes by the *lookup*() primitive (line 6), which either returns the entry stored locally (line 22) or requests the entry to the key owner (lines 24 and 25). According to Rule 3, the returned entry is cacheable if it is the newest entry stored by its owner and the owner's snapshot counter *SC* is at least as big as the requested version (line 17). Before responding to a remote lookup request, the owner of a key makes sure that its *SC* is equal to or greater than the requested version, ensuring Rule 1 (line 14).

Upon delivering a committing transaction *t* (line 8), a vnode certifies *t* using the same procedure as core servers (line 9, 31-37). If the outcome of certification is commit, *t*'s modification to the database are applied to the local storage (lines 11 and 12). The apply primitive only keeps an entry in local storage if it satisfies Rule 2 (lines 29 and 30).

4.4 Cache-only vnodes

So far we assumed that each vnode owns a portion of the dataset. A cache-only vnode is a regular vnode that does not own any data items, in which case it will only cache data items owned by other vnodes and does not store any data items permanently.

Algorithm 3 In-Memory Deferred Update Replication, server s

```
1:  Initialization
2:     SC ← 0                                          {snapshot counter}
3:     DB[...] ← ∅                                     {list of applied transactions}
4:  when receive(read, k, st) from c
5:     if st = ⊥ then st ← SC                          {if first read, initialize snapshot}
6:     v ← lookup(k, st)                               {most recent version ≤ st in database}
7:     send(k, v, st) to c                             {return result to client}
8:  when adeliver(t)
9:     outcome ← certify(t)                            {outcome is either commit or abort}
10:    if outcome = commit then                        {it passes certification...}
11:       for all (k, v) ∈ t.ws do                     {for each update in t's writeset...}
12:          apply(k, v)                               {...apply update to database}
13:    send(outcome) to c                              {return outcome to client}
14: when receive(remote-lookup, k, sc, st) from r and SC ≥ st      {Rule 1}
15:    v ← lookup(k, st)
16:    cacheable ← false
17:    if v is newest version stored locally and SC ≥ sc then       {Rule 3}
18:       cacheable ← true
19:    send(k, v, st, cacheable) to r
20: function lookup(k, st)                             {used in lines 6 and 15}
21:    if k is stored locally then
22:       return retrieve(k, st)
23:    else
24:       send(remote-lookup, k, SC, st) to owner(k)
25:       wait until receive(k, v, st, cacheable)
26:       if cacheable then store(k, v, st)
27:       return v
28: function apply(k, v)                               {used in line 7}
29:    if s = owner(k) or k is stored locally or pending remote request for k
       then                                            {Rule 2}
30:       store(k, v, SC)
31: function certify(t)                                {used in line 9}
32:    for i ← t.st to SC do                           {for all concurrent transactions...}
33:       if DB[i].ws ∩ t.rs ≠ ∅ then                  {if some intersection...}
34:          return abort                              {transaction must abort}
35:    SC ← SC + 1                                     {here no intersection: one more snapshot}
36:    DB[SC] ← t                                      {keep track of committed writeset}
37:    return commit                                   {transaction must commit}
```

Cache-only vnodes can be deployed in an existing RAM-DUR cluster on-the-fly, with minimal impact on the performance of the cluster. A cache-only vnode starts *empty*, with no data items in its storage. Therefore adding a cache-only vnode does not require a recovery step, and will not require to redistribute data over the cluster of vnodes (which is possibly an expensive operation).

Deploying cache-only vnodes can be advantageous in many circumstances. For instance, some services follow diurnal or seasonal patterns, where the service is subject to workload spikes during particular times of the day. Cache-only vnodes do not add storage capacity, instead they increase the available processing power to execute transactions in the workload. Cache-only vnodes are thus useful for absorbing workload spikes due to increased number of clients.

4.5 Discussion

The motivation for vnodes in RAM-DUR is performance. Two mechanisms implemented by vnodes improve performance. First, by requesting a missing data from a remote vnode, no local disk reads are necessary. This approach is quite effective since retrieving a missing item from the main memory of a remote vnode is much faster than retrieving the item from the local disk (with current hardware, a round-trip in a local area network is in the order of 0.1 milliseconds, while a non-cached disk access may take 10ms, a discussion maybe found in [42]). Second, items often accessed are cached by vnodes, thus reducing network traffic and execution delay. Cached data is updated as part of the termination of transactions, ensuring "data freshness".

Our mechanism to broadcast terminating transactions to servers is based on Paxos [36]. Paxos's "acceptors" can be co-located with core servers or deployed at independent nodes, the approach used in all our DUR and RAM-DUR experiments. Therefore, when a transaction is delivered by a server, the transaction's contents and order have been safely stored by the acceptors and will not be forgotten, despite failures. This mechanism ensures transaction durability despite the crash of vnodes.

Although recovering a vnode from the state of core servers is a simple operation, until it recovers, all entries the vnode owns will be unavailable for other vnodes. To prevent blocking due to the crash of a vnode, we extend the logic of vnodes to request missing items from core servers, should they suspect the crash of one of its peers. Notice that as long as core servers respect the rules presented in the previous sections, consistency will be preserved.

4.6 Implementation and optimizations

For the implementation of RAM-DUR, we use Ring Paxos [41] as our atomic broadcast primitive. Acceptors log delivered values on disk asynchronously, as part of the atomic broadcast execution—we assume that there is always a majority of operational acceptors in order to ensure durability. Optionally, RAM-DUR servers can store a consistent snapshot of their in-memory state to disk. This checkpointing mechanism can be used for backups, or to restart the whole system by replaying only the tail of the Paxos log since the last checkpoint.

Our DUR and RAM-DUR prototypes broadcast transactions in small batches. This is essentially the well-known *group commit* optimization in centralized databases. In our case, it amortizes the cost of the atomic broadcast primitive over several transactions.

We use bloom filters to efficiently check for intersections between transaction readsets and writesets. The implementation keeps track of only the past K writeset bloom filters, where K is a configurable parameter of the system. 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. Using bloom filters results in a negligible number of transactions aborted due to false positives.

We used the same code base to implement standard DUR. In the case of DUR, we disabled those features that are unique to RAM-DUR (i.e., each server has a full copy of the database, does not issue remote requests, and has no cache mechanism). Local storage of DUR is implemented using Berkeley DB (BDB); for RAM-DUR, we provide an alternative storage implemented as a multiversion hash-table, optimized for keeping data in-memory only.

4.7 Performance evaluation

We assess next the performance of RAM-DUR. We measured throughput and latency of RAM-DUR and compared them against standard DUR, under workloads that both fit and do not fit in the memory of a single server. We also assess the effects of adding cache-only vnodes to a running RAM-DUR cluster.

4.7.1 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. Servers are attached to a 73 GB 15krpm SAS hard-disk.

We evaluated the performance of RAM-DUR using a simple micro-benchmark. The benchmark consists of two types of transactions (see Table 4.1): (1) *update transactions* perform a read and a write on the same key; (2) *read-only transactions* perform two read operations on different keys. Keys are 4 bytes long, while values consist of 1024 bytes.

Clients are evenly distributed across servers. Each client issues a mix of 10% update and 90% read-only transactions. Keys are selected uniformly at random from a portion of $1/n$ of the total number of items in the dataset, where n is the number of servers. Clients connected to the same vnode access the same portion, which ensures some data locality.

We consider two datasets (see Table 4.2): (1) a *small database*, where servers are loaded with 100 thousand data items per vnode and the dataset fits in the memory of a single server; and (2) a *large database*, where servers are loaded with 400 thousand items per vnode and the dataset does no fit in the memory of a single server, but it fits the aggregated memory of all servers.

Moreover, to make the comparison between DUR and RAM-DUR fair, we use the same number of nodes. Therefore, in DUR and RAM-DUR 3, only three servers in total execute transactions, core servers in DUR and vnodes in RAM-DUR. We assess the benefit of additional vnodes with RAM-DUR 6, a configuration with 6 vnodes.

Type	**Operations**	**Frequency**
Read-only	2 reads	90%
Update	1 read, 1 write	10%

Table 4.1. Transaction types in workload

Dataset	**Size**	**Characteristic**
Small DB	100K items per server	fits RAM of a single server
Large DB	400K items per server	fits aggregated servers' RAM

Table 4.2. Datasets in workload

4.7.2 Throughput and latency

Figure 4.6 shows throughput and latency of DUR and RAM-DUR under maximum load. When data fits in memory (i.e., Small DB) we notice that both DUR and RAM-DUR perform well, although RAM-DUR performs two times faster than DUR. We attribute this difference to the fact that DUR is fully replicated, and thus, every update must be applied to storage. Also, RAM-DUR's in-memory only storage is faster than DUR's Berkeley DB-based storage. In terms of latency, DUR does better than RAM-DUR, especially for read-only transactions. This difference is due to the fact that RAM-DUR sustains more clients (in this experiment we run 8 and 32 clients per server for respectively DUR and RAM-DUR).

When the dataset does not fit in the memory of a single node (i.e., Large DB) we notice a significant impact on DUR's performance. Performing reads from the local storage and applying updates requires disk access. In this setup, DUR performs only 94 update transactions and 816 read-only transactions per second. On the other hand, we observed a minimal performance impact in the case of RAM-DUR. DUR's latency is also heavily impacted: latency of update transactions more than doubled; latency of read-only transactions went from only 0.4 milliseconds to slightly less than 6 milliseconds.

Finally, increasing the number of servers in RAM-DUR is beneficial. Going from 3 to 6 vnodes roughly doubled the performance of RAM-DUR, at the same time keeping latency low.

4.7.3 Performance under remote requests

In the following experiment, we show how quickly a vnode builds its working set and how its performance compares to a core server in DUR, where remote requests never take place. In this experiment, an equal number of clients issue transactions against DUR and RAM-DUR and, for the sake of the comparison, we only consider the case where the dataset fits in memory. Figure 4.7 shows throughput, latency, and the number of remote requests over time. At time 0, RAM-DUR servers where loaded with 100 thousand items per node, however they store locally only data items they own, and do not store any cached items. Transaction execution starts at time 1, and at this time RAM-DUR's throughput and latency are worse than DUR's. As the number of remote requests diminish, RAM-DUR quickly catches up with DUR, and at time 5 both protocols roughly perform the same. At time 10 the number of remote requests per second is one fifth of the value in the beginning of the execution, and RAM-DUR starts

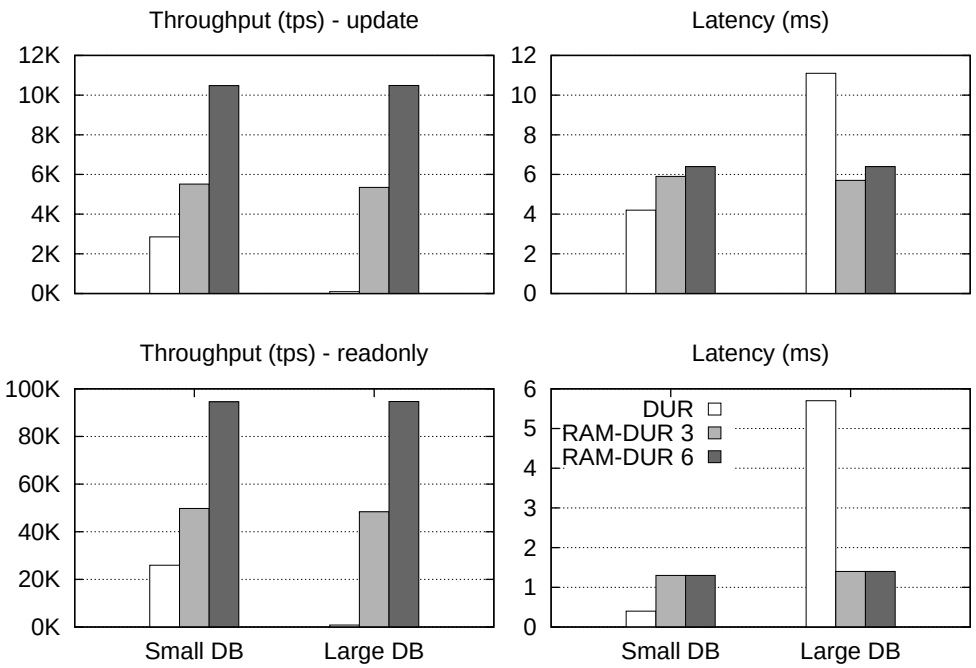

Figure 4.6. Throughput and latency.

approaching its peak performance.

4.7.4 Cache-only vnodes

We evaluate next the effects of online additions of cache-only vnodes (Figure 4.8). A cache-only vnode is a vnode which does not own any data items. Initially there are 6 vnodes. At time 20, a cache-only vnode is added to the compound and at time 60 another one is included. System throughput increases as more vnodes are added. The first graph, on the top of Figure 4.8, shows the aggregated throughput of all vnodes in the system as well as the additional throughput added by the 7th vnode. In terms of latency, we observe that the addition of a new vnode slightly increases the average aggregated latency. This effect is due to the fact that cache-only vnodes start without any items, and therefore every read results in a remote request initially. The increase in latency is experienced only at the added vnode. However as new vnodes cache

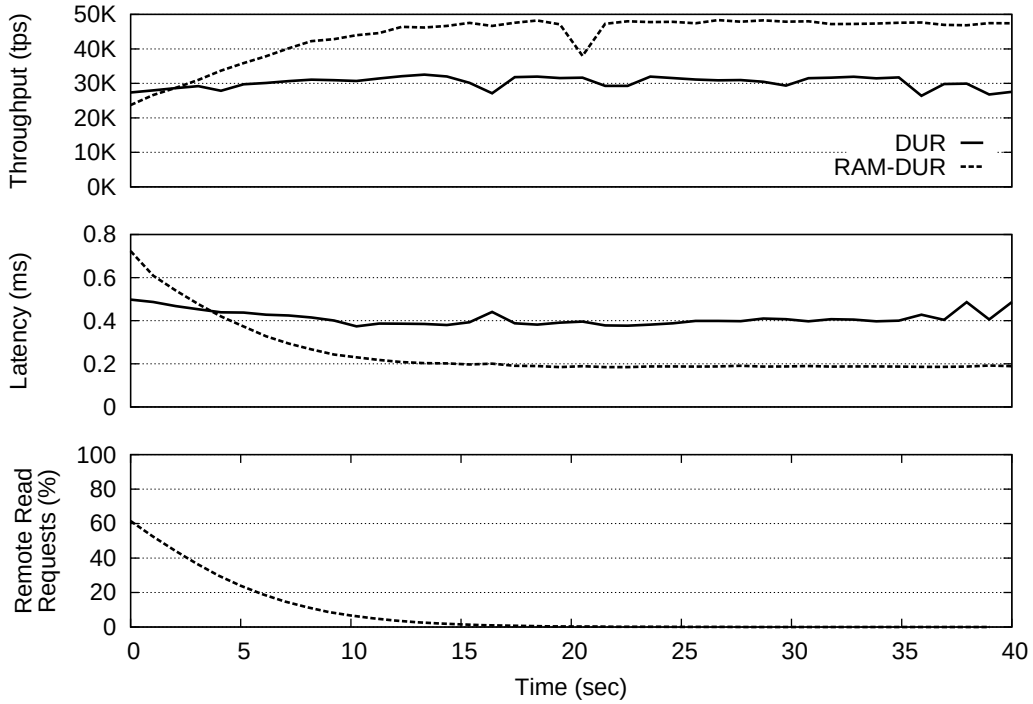

Figure 4.7. Performance under remote requests (Small DB).

their working set, remote requests decrease. After 10–15 seconds, the 7th vnode contributes the same throughput as the vnodes that were already present at the beginning of the execution, its latency approaches the aggregated average latency, and the number of remote requests per second approaches zero.

4.8 Related Work

A number of protocols for deferred update replication where servers keep a full copy of the database have been proposed (e.g., [2; 32; 38; 44; 47]). Similar to RAM-DUR some protocols provide partial replication to improve the performance of deferred update replication (e.g., [54; 58; 60]). This also improves scalability in that only the subset of servers addressed by a transaction applies updates to their local database. These protocols, including RAM-DUR, require transactions to be atomically broadcast to all participants. However, only RAM-

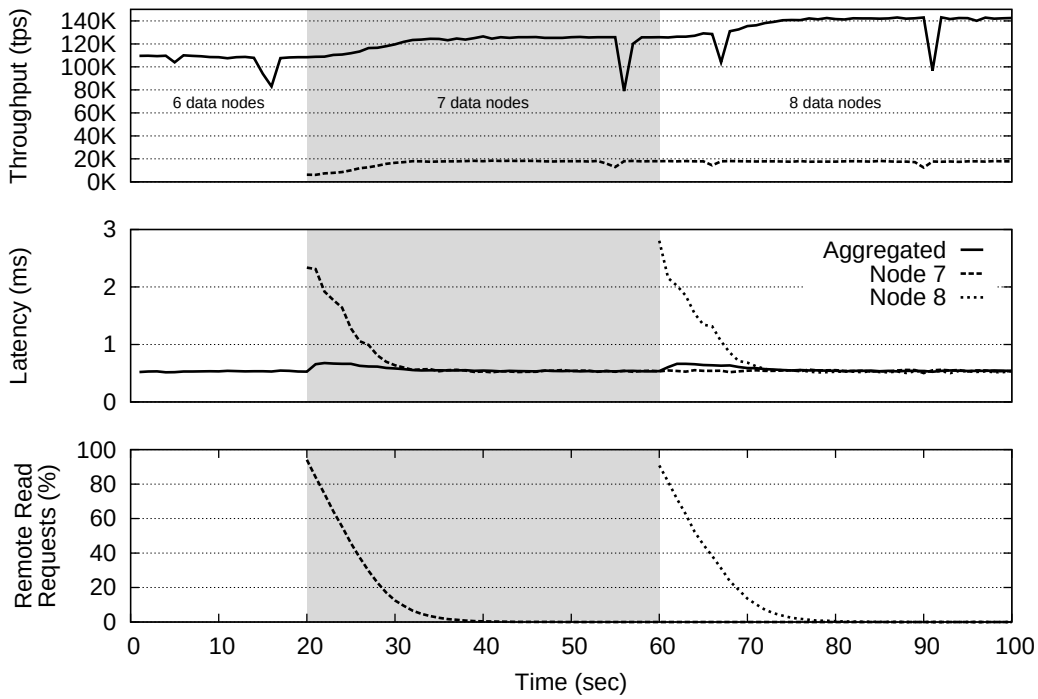

Figure 4.8. Adding cache-only vnodes (Large DB).

DUR addresses the case where data is kept always in memory.

Full database replication hinders the scalability of update transactions, which in RAM-DUR is inherently limited by the number of transactions that can be ordered. We address the scalability issue of update transactions in Chapter 5. Nothing prevents combining RAM-DUR's mechanisms described here with other techniques for improving scalability.

G-Store [18] proposes a *key group protocol* that allows transactional multi-key access over dynamic and non-overlapping groups of keys. To execute a transaction that accesses multiple keys, the key group protocol must transfer ownership for all keys in a group to a single node. Once a node owns a group, it can efficiently execute multi-key transactions. RAM-DUR has no such constraints.

H-Store [64] shares some design considerations with RAM-DUR, although the main approach is different. In H-Store the execution is optimized depending on transaction classes and schema characteristics in order to distinguish

two-phase, *strictly two-phase* and *sterile* transactions and attain high performance. This can be done automatically, or by pre-declaring transaction classes, which makes the system less flexible.

Recent trends in networking hardware suggests that in-memory storage systems should make use of modern networking equipment supporting *Remote Direct Memory Access* (RDMA) [22; 31]. RDMA allows applications to access memory of a remote node directly, without interrupting the remote CPU, yielding significant improvements in latency. RAM-DUR's remote reads could be implemented using RDMA.

As discussed in Section 4.6, the implementation of RAM-DUR makes use of Bloom filters to efficiently perform certification. A similar technique was first introduced in the D^2STM distributed transactional memory system [16].

Tashkent+ [25] exploits information about the working set to load balance transactions over a set of replicas. The idea is to better utilize the memory of a cluster by submitting related transactions to the same replica. Similar techniques could be used in RAM-DUR to improve cache effectiveness of vnodes.

RAM-DUR has been extended with indexing and integrated with MySQL to support full relational semantics [67; 68].

4.9 Conclusion

This chapter presented an extension to deferred update replication. Deferred update replication is widely used by several database protocols due to its performance advantages: scalable read-only transactions, and good throughput under update transactions. RAM-DUR extends deferred update replication with the goal of in-memory execution. We introduce two mechanisms, remote reads and caching, which allow us to significantly speedup the execution phase in workloads that do not fit the memory of a single server. Moreover, we do so without sacrificing consistency. We assessed the performance of RAM-DUR under different scenarios and showed how RAM-DUR can quickly add cache-only vnodes to further improve system throughput online.

Chapter 5

Scalable Deferred Update Replication

This chapter introduces Scalable Deferred Update Replication (S-DUR), which improves the scalability of read-only and update transactions. 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 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 [8], the termination of global transactions is non-blocking since each partition is highly available.

5.1 Motivation

While read-only transactions scale with the number of replicas in deferred update replication (Chapter 3), 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. 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 is an inherent limitation of DUR, which we address in the next sections.

5.2 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, $writeset(t) \subseteq readset(t)$.

5.3 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. 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. 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.

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 1. 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 5.1). 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: in any serial execution involving t_i and t_j, either t_i must read t_j's writes or the other way around.

5.4 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

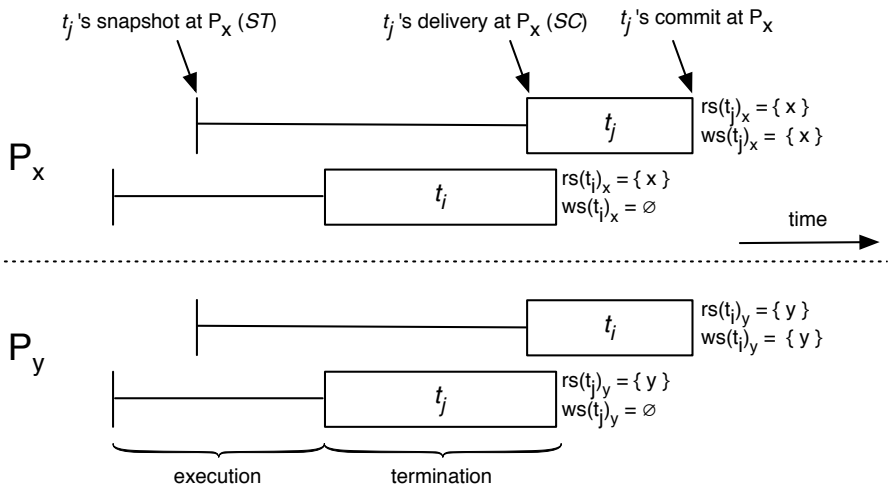

Figure 5.1. Example 1. Transactions t_i and t_j are delivered in different order at partitions P_x and P_y. At partition P_x the readset of t_j does not intersect the writeset of t_i. Similarly, at partition P_y the readset of t_i does not intersect the writeset of t_j. Both partitions commit, but the execution is not serializable.

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, 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.

5.4.1 The algorithm in detail

Algorithm 4 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

Algorithm 4 Scalable Deferred Update Replication, client c

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

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 reads to the same partition observe a consistent view. At commit time (lines 18–20), the client submits a transaction to one server (line 18), which will be broadcast for certification to all partitions concerned by the transaction. The client then waits for a server that replies with the transaction's outcome (lines 19–20).

Algorithm 5 shows the server side. When local transaction t is delivered, it is certified (line 15) and appended to the queue of pending transactions (line 39), if local certification for t results in a commit. The order of the transactions in the pending queue *PL* follows the delivery order, and it is the same on every server within the same partition. The pending queue is consumed in order; when t is at the head of *PL* it can be completed (lines 20–23). To complete t, the protocol proceeds as follows: if t passes certification, it is applied

to the local database, thus generating a new snapshot, and the outcome of t is sent to the client (lines 24–29).

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

5.5 Handling partially terminated transactions

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

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.

Algorithm 5 Scalable Deferred Update Replication, server s in partition p

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

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.

5.6 Certification—less read-only transactions

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

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

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

Example 2. The problem with local read-only transactions is that although they may individually see a consistent database snapshot, together they may lead to an inconsistent execution.

Assume global transaction t_i updates both x_1 and y_1, global transaction t_j updates x_2 and y_2. Transactions t_a and t_b are local and readonly: t_a reads x_1 and x_2; t_b reads y_1 and y_2 (see Figure 5.2). At P_x, t_i is delivered first and passes certification; then t_a is executed and reads the value of x_1 written by t_i and the initial value of x_2. Finally, t_j is delivered and since its readset does not

intersect the writesets of t_i, it passes certification. At P_y, t_j is delivered first, followed by t_i. Transaction t_b reads the value of y_2 written by t_j, and the initial value of y_1. Similarly to what happens at P_x, all transactions commit, although the execution is not serializable. Notice that in this case global transactions t_i and t_j do not read or write any data items in common.

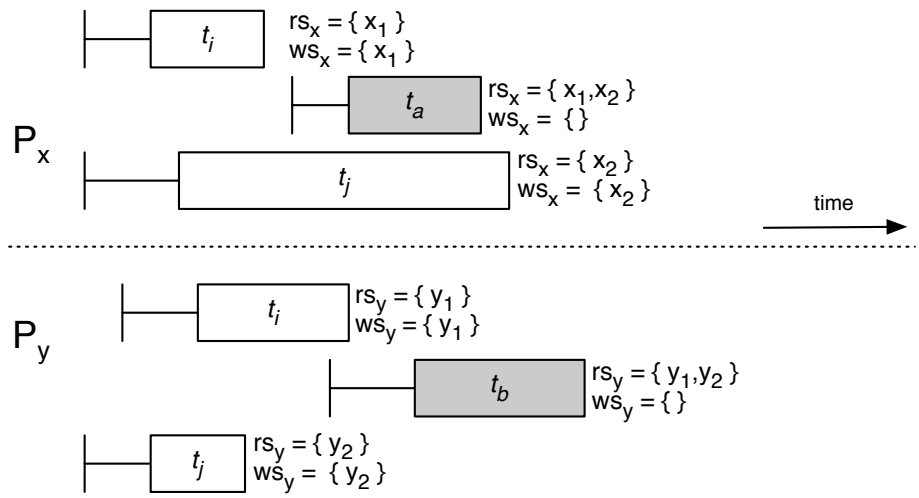

Figure 5.2. Example 2. At partitions P_x, read-only transaction t_a observes the effects t_i but not the effects of t_j. At partition P_y, read-only transaction t_b observes the effects of t_j but not the effects of t_i. Thus violating serializability.

Example 3. Consider now a global read-only transaction that reads from the latest snapshot counters available at different partitions. For instance, assume that read-only transaction t reads from snapshots SC_x and SC_y at partitions P_x and P_y respectively. Transaction t may see an inconsistent view of the database, in that SC_x may contain a global transaction t_g that is not included in SC_y. Potentially, t may see some of the updates of t_g at partition P_x, and miss some updates at partition P_y, thus violating serializability.

5.6.1 The algorithm in detail

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

counter created by T at partition i. Similarly, for all transactions T that succeed S, S's snapshot counter at partition i must be smaller than the snapshot counter created by T at partition i.

Algorithm 6 shows how to find a serializable snapshot. For simplicity, we consider that each partition is composed of a single server. The complete protocol uses the broadcast primitive to ensure markers are received and processed in the same order within the same partition. The algorithm is similar to the Chandy-Lamport snapshot protocol [11], in that it uses markers, and relies on FIFO-channels. Function *snapshot*() (lines 1–3) is invoked by server s to initiate the snapshot algorithm. To take a snapshot a server sends the marker to itself. When a marker is received (lines 4–7) for the first time by server s, s will suspend the delivery of global transactions, and relay the marker to all other servers (in FIFO order). When server s receives a marker from all partitions, then a snapshot has been found and each server records its SC, the current snapshot counter. Collectively, the SC of all partitions is a snapshot that does not include partial views of transactions, thus avoiding the problem we illustrated in example 3. Notice that to ensure serializable snapshots, only one instance of the snapshotting algorithm can be active at any time. Otherwise, two concurrent instances of the algorithm could create snapshots that allow the execution of example 2. To ensure this, only one partition initiates the algorithm by calling function *snapshot*().

Algorithm 6 Snapshot Algorithm, server s in partition p

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

5.7 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 5.9, the difference is 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 [46].

5.8 Implementation and optimizations

We use LibPaxos [37] as our atomic broadcast primitive. There is one instance of LibPaxos per partition. Our prototype differs from Algorithms 4 and 5 in the following aspects:

- Each client connects to a single server only and submits all its read requests to the server it connected to. Clients are oblivious to the partitioning scheme. When a server receives a read request for key k that is not stored locally, the server routes the request to one server in the partition that is responsible for storing k. In the following experiments we use range partitioning.

- Our implementation reduces the number of vote messages exchanged for global transactions. Only one designated replica in a partition sends vote messages, reducing the number of messages required by the protocol. 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.

- Similar to the implementation of RAM-DUR (Section 4.6), we use bloom filters to efficiently check for intersections between transaction readsets and writesets. Differently from RAM-DUR, S-DUR requires to store both readsets and writesets to perform the extended certification test.
- 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.

5.9 Performance Evaluation

In this section we assess the performance of Scalable Deferred Update Replication under different workloads. We look into throughput, latency and abort rate of S-DUR and compare them to our baseline DUR of Chapter 3.

5.9.1 Setup and benchmarks

We ran experiments on Amazon EC2, using r3.large instances, equipped with two virtual cores and 15GB of main memory. These instances support enhanced networking capabilities, and have hardware support to efficiently virtualize network I/O. We deployed r3.large instances in one availability zone, in the EU West region.

We evaluated the performance of S-DUR using four different workloads, which we summarize in Table 5.1. For instance, workload type A reads two items and updates them, both keys and values are 4 bytes. Workload types A and B perform updates only, while workload types C and D perform read-only transactions. The number of operations and value sizes are chosen such that the implementation performs under two different conditions. Workload types A and C perform small operations, and in this particular deployment the throughput is CPU bound. Workload types B and D transactions perform few operations on relatively large data items (512 bytes). In which case the throughput of type B transactions is limited by the number of transactions that can be submitted through the atomic broadcast primitive per time unit. Workload type C is also network bound: throughput is limited by the number of read operations servers can reply to.

In the experiments, we vary the percentage of global transactions, in which case clients access items stored in two different partitions. Clients select the keys of the operations to perform uniformly at random.

Type	Reads (ops)	Writes (ops)	Key size (bytes)	Value size (bytes)	DB size (items/partition)
A	2	2	4	4	1M
B	2	2	4	512	1M
C	32	0	4	4	1M
D	8	0	4	512	1M

Table 5.1. Workload types - Varying number of read and write operations per transaction (ops), and key and value sizes (bytes).

We ran experiments with increasing number of partitions, each partition is composed of five nodes: three nodes ran S-DUR servers collocated with acceptors and proposers for the Paxos protocol; the remaining two nodes were allocated to clients for generating the load. Three servers per partition means there are a total of three replicas for each data item.

We report throughput and the corresponding latency at peak performance. In all experiments, we generate one new snapshot every second using the mechanism described in Section 5.6. Each experiment lasts 80 seconds; we discard 10 seconds from the beginning and the end of the execution, leaving 60 seconds worth of traces per experiments.

5.9.2 Throughput

Figure 5.3 shows the maximum 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. For comparison, we also show deferred update replication with 6, 12 and 24 replicas (last three columns in the graphs).

Under update transactions (workload types A and B), the throughput of S-DUR scales linearly with the number of partitions. For instance, with workload A and no global transactions, doubling the number of partitions also doubles the total throughput. Two partitions perform twice the number of transactions compared to the baseline, and similarly for 4 and 8 partitions.

The performance of S-DUR degrades as the percentage of global transactions increases. Global transactions are more expensive for the following reasons. Partitions involved by global transactions need to exchange votes in the termination phase. Also, 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). Finally, in our implementation clients connect to

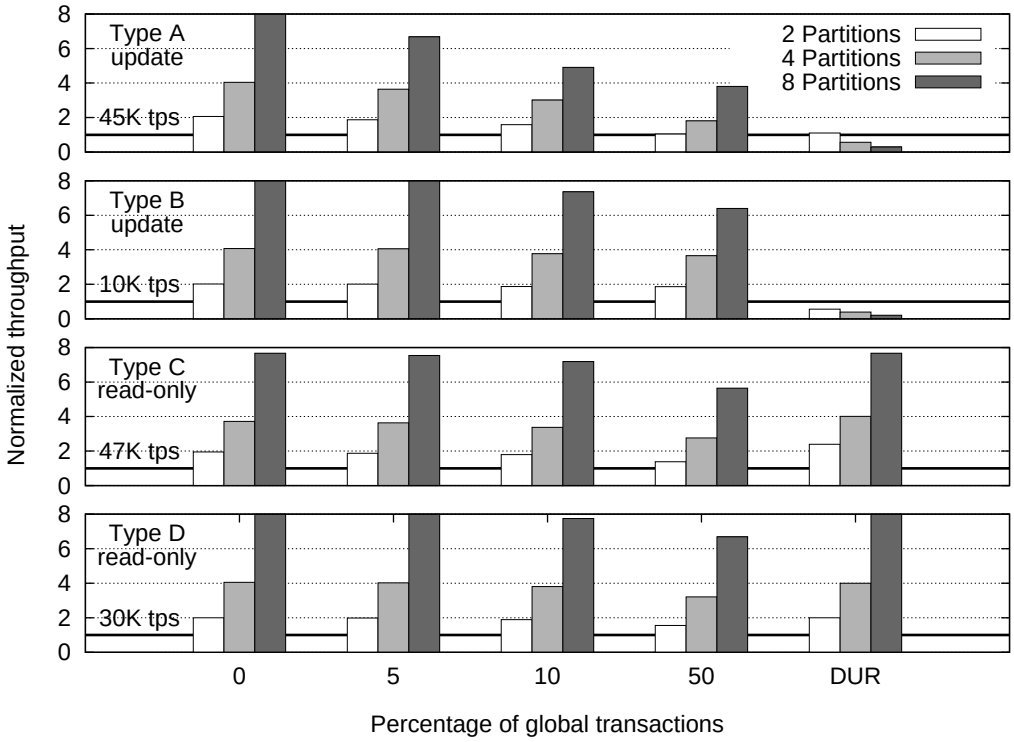

Figure 5.3. Normalized throughput versus percentage of global transactions.

one server only, and are oblivious to the partitioning scheme (see Section 5.8). In this case, reading from a different partition costs roughly twice as much compared to reading a value from the partition in which the client is connected to.

This effect can be noticed in both workload A and B. However, the effect is less pronounced in workload B. In this workload, transactions operate on larger value sizes and the system is bound by the number of transactions that can be broadcast within a partition. Adding the extra costs of exchanging votes and reading from remote partitions has a low impact, compared to workload A, where the system is CPU bound and adding extra costs has immediate impact on throughput.

Nonetheless, system throughput scales with the number of partitions. For instance, in workload A and 50% of global update transactions, two S-DUR partitions perform roughly the same as the baseline. However, doubling the number of partitions, doubles the throughput. A configuration with eight parti-

tions (i.e., 24 server nodes) can execute roughly four times more update transactions compared to the baseline. Similarly, in workload type B and 50% of global transactions, eight S-DUR partitions performs more than six times more update transaction compared to the baseline.

The results of Figure 5.3 confirm that update transactions in DUR do not scale as replicas are added to the system. In fact, notice that including more replicas reduces throughput. Adding replicas increases the overhead of broadcasting transactions to all replicas, which explains the negative effect.

We next consider read-only transactions (workloads C and D). Under read-only transactions, DUR and S-DUR have similar behavior. In workloads C and D, the baseline DUR protocol scales throughput linearly with the number of servers. Since the termination of a read-only transaction is completely local to the executing server, the addition of a server increases throughput proportionally.

With 0% of global transactions S-DUR has identical performance to the baseline DUR protocol. In the case of read-only transactions, the additional cost induced by global transactions is due to the increased cost of reading from remote partitions: read operations for keys replicated in a different partition require one extra lookup and round-trip to the partition that stores the requested item. In fact, workload C degrades more quickly than workload D because transactions in workload D perform fewer read operations (even if workload B reads larger values).

5.9.3 Latency

Figure 5.4 shows average and 99th percentile latency for workloads B and D reported in the previous section, while the black horizontal bars represent the 99-th percentile latency measured for the baseline experiment with 1 partition. The graph distinguishes between local and global transactions.

For workload B, all S-DUR configurations show similar latency compared to the baseline; in most cases there is less than one millisecond difference. However, the 99-th percentile latency increases noticeably with 8 partitions and 50% of global transactions. Since the average is slightly affected, the graph suggests that the increase in 99-th percentile latency is due to few unlucky transactions, that wait for slow global transactions to finish (we consider this phenomenon in the next chapter). In general, due to the extra overhead of global transactions, their latency is between 1 and 2 milliseconds higher than the latency of local transactions.

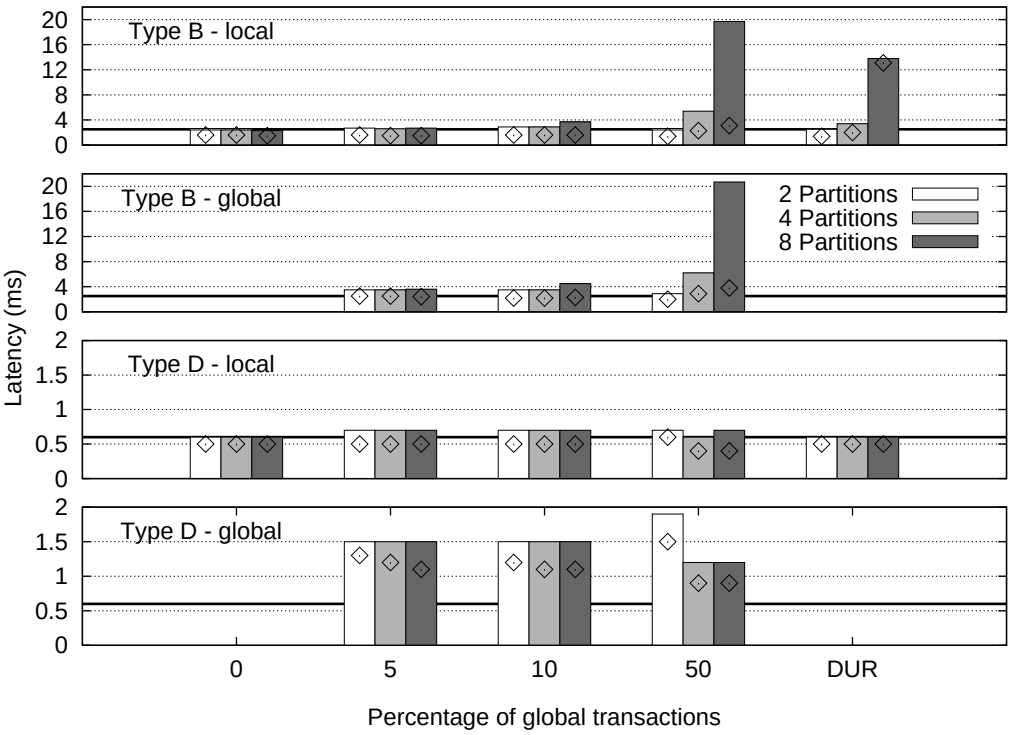

Figure 5.4. Latency versus percentage of global transactions. 99-th percentile latency (bars) and average latency (diamonds).

For workload D, the latency of local read-only transactions in S-DUR is essentially the same as the baseline latency, differences are about 0.1 milliseconds. The latency of global read-only transactions is about twice the latency of local read-only transactions. 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.

5.9.4 Abort rate

Figure 5.5 reports the abort rates of workload B as we increase the percentage of global transactions. The graph distinguishes local and global transactions. For all configurations, we observed less than 0.1% of aborts for local transactions and less than 1% of aborts for global transactions.

In general, aborts are influenced by a combination of two factors: decreas-

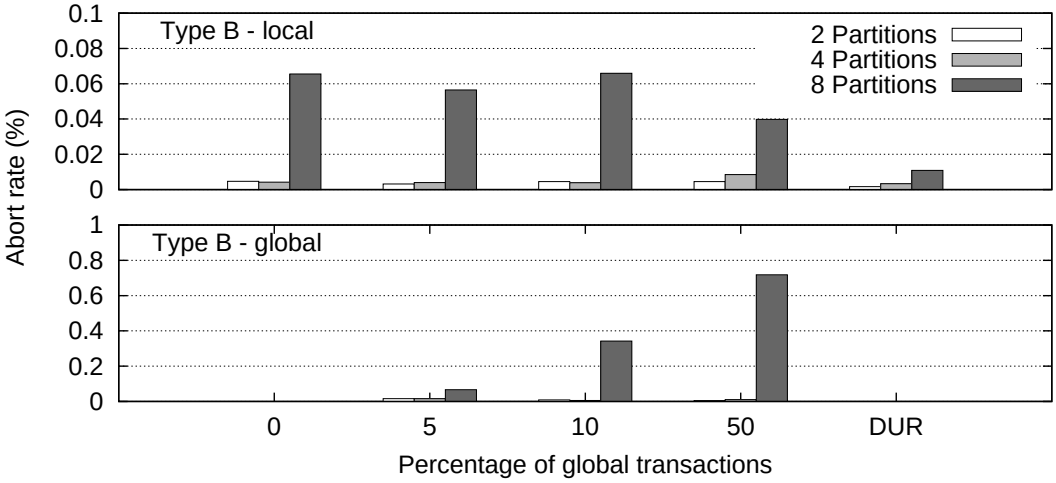

Figure 5.5. Abort rate versus percentage of global transactions.

ing 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 and their execution takes more time.

With local transactions, abort rate remains stable with respect to the percentage of global transactions. With 8 partitions, the abort rate tends to decrease with the percentage of global transactions. We attribute this behavior to the reduced throughput achieved with 8 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.

5.10 Related Work

We reviewed full replication protocols in Section 4.8 and discussed their limitations in Section 5.1. In the following, we focus the discussion on partial replication and partitioning.

Some protocols implement partial database replication using atomic multicast primitives (e.g., [6; 27; 55]). Fritzke et al. [27] 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. SIPRe [6] improves over the protocol of [27] in

that only two invocations of atomic multicast are needed in order to execute a transaction, one to begin the transaction, and one to terminate it. P-Store [55] 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 5.1). On the other hand, genuine atomic multicast is more expensive than atomic broadcast in terms of communication steps [53], and thus transactions take longer to be certified.

GMU [48] is a partially replicated protocol targeted for distributed transactional memories. GMU provides update serializable transactions [29], that is, update transactions are serializable, however read-only transactions may observe two update transactions in different orders (for instance the problematic execution of Figure 5.2). Weakening the guarantees of read-only transactions allows to observe fresh data, without aborting read-only transactions.

Differently from previous works, Sinfonia [3] 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.

Rao et al. [49] proposed a storage system called Spinnaker, which is similar to the approach presented here in that it also uses several instances of Paxos [36] to achieve scalability. Differently from S-DUR, however, it does not support transactions across multiple Paxos instances.

This thesis does not investigate any partitioning techniques. In the experiments, we generally used simple range partitioning. However, the assignment of data items to S-DUR partitions, together with the workload, determines the number of transactions involved in two or more partitions. Devising partitioning techniques is fundamental to achieve scalable performance. Several partitioning techniques have been devised which could be used in S-DUR (e.g., [17; 45; 65]).

P-DUR is an extended version of S-DUR that supports parallel execution on multicore servers [43]. In P-DUR a small set of servers fully replicate the database, and each server partitions its copy of the database across the different cores available to the server. In this case, partitioning helps in making efficient use of the multiple cores available in current hardware.

5.11 Conclusion

This chapter proposes an extension of the deferred update replication approach. Deferred update replication is implemented by several data management systems due to its performance advantages, namely, good throughput in the presence of update transactions and scalability under read-only transactions. Scalable Deferred Update Replication 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 technique.

Chapter 6

Geo-Replication using Deferred Update Replication

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.

6.1 Motivation

Scalable deferred update replication offers good performance, which under certain workloads grows proportionally with the number of database partitions, 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 S-DUR, it has important consequences on the latency perceived by the clients. S-DUR distinguishes between local transactions, those that access data in a single partition, and global transactions, those that access data in multiple partitions. Intuitively, a local transaction will experience lower latency than a global transaction since it does not require the two-phase commit-like termination needed by global transactions. Moreover, in a geographically distributed environment, the latency gap between local and global transactions is likely wider since the termination of global transactions may involve servers in re-

mote 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 the state of replicas would diverge. As a consequence, a local transaction delivered after a global transaction will experience a longer delay than if executed in isolation. We have assessed this phenomenon in a geographically distributed environment and found that even a fairly low number of global transactions in the workload is enough to increase the average latency of local transactions by more than 10 times.

This chapter discusses how S-DUR can be deployed in geographically distributed systems, and proposes solutions to the problem mentioned above. The chapter describes three optimizations. The first optimization is based on delaying the submission of global transactions. It is a simple technique, but provides limited improvements. The other two optimizations are based on *reordering* the delivery of transactions. Reordering relies on the observation that transactions can often be committed in an order that is different from the delivery order. Reordering transactions needs to be a deterministic, so that every replica of the same partition reorders transactions in the same way.

6.2 A system model for geo-replication

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

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

6.3 Geographically distributed deployments

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

The first deployment ("WAN 1" in Figure 6.1) places a majority of the servers that replicate a partition in the same region, possibly in different datacenters; other servers replicating p are distributed across regions. Having the majority of servers in the same region allows the partition to order messages quickly and therefore terminate local transactions without long delays; global transactions are subject to inter-region delays.

A local transaction executed against the preferred server of partition P_1 (s_1 in the figure) will terminate in 4δ, where δ is the maximum communication delay among servers in the same region. A global transaction that accesses partitions P_1 and P_2, executed against server s_1, will be subject to $4\delta + 2\Delta$, where Δ is the maximum inter-region delay.

The second deployment ("WAN 2") distributes the servers of a partition across regions. This deployment can tolerate catastrophic failures, as we discuss next. The termination of a local transaction will experience $2\delta + 2\Delta$ since Paxos will no longer run among servers in the same region. Global transactions are more expensive than local transactions, requiring $3\delta + 3\Delta$ to terminate. Note that we do not place server s_4 in Region 1 because this would result in Region 2 having no preferred server. Therefore, if a region r contains a server of p, read operations issued by transactions in execution at r on items in p will not experience long delays. Although this deployment tolerates the failure of an entire region (i.e., a catastrophic failure), it imposes longer delays in the termination of both local and global transactions.

In both deployments, a global transaction that executes at P_1 (respectively, P_2) will read items from P_2 (P_1) within 2δ. Servers in remote regions speed up the execution of global transactions that execute in these regions and read data items in p.

Deployments WAN 1 and WAN 2 tolerate the failure of servers in a partition as long as a majority of the servers is available in the partition. The first deployment, however, does not tolerate the failure of all servers in a region, since such an event would prevent atomic broadcast from terminating in some partitions. For example, in Figure 6.1, first deployment, if all servers in Region 1 fail, then transactions local to Partition 1 and global transactions that modify Partition 1 will not terminate.

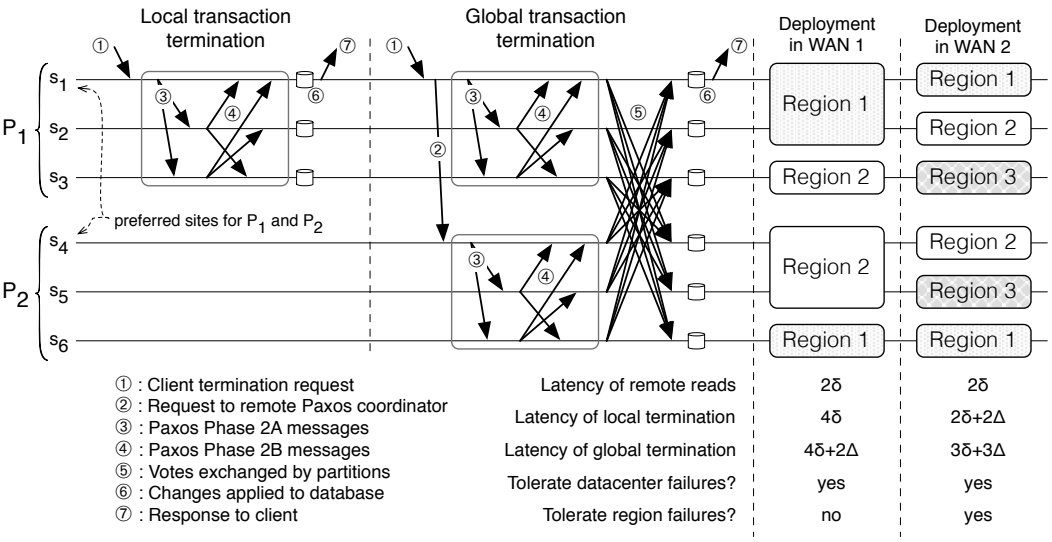

Figure 6.1. Scalable Deferred Update Replication deployments in a geographically distributed environment, where δ is the maximum communication delay between servers in the same region and Δ is the maximum communication delay across regions; typically $\Delta \gg \delta$. The database contains two partitions, P_1 and P_2, and clients are deployed in the same datacenter as server s_1.

6.4 Performance considerations

S-DUR provides in theory good latency in the geo-replicated settings described above. In practice, we identified the following problem that prevents S-DUR from achieving good performance in wide-area networks. A partition decides the order of a global transaction at delivery time. As a consequence global transactions potentially delay local transaction by up to two inter-region delays. For example, if t_i is delivered before t_j in partition p, t_i will be certified before t_j. If t_i and t_j pass certification (in all concerned partitions), t_i's updates

will be applied to the database before t_j's. While this mechanism guarantees deterministic transaction termination, it has the undesirable effect that t_j may have its termination delayed by t_i. This is particularly problematic in S-DUR if t_i is a global transaction and t_j is a local transaction since global transactions may take much longer to terminate than local transactions.

The consequences of global transactions on the latency of local transactions depend on the difference between the expected latency of local and global transactions. For example, in WAN 1 local transactions are expected to terminate much more quickly than global transactions, which is not the case in WAN 2. Thus, global transactions can have a more negative impact on local transactions in WAN 1 than in WAN 2. We have assessed this phenomenon experimentally (details in Section 6.8) and found that in WAN 1, global transactions can increase the latency of local transactions by up to 18 times. We next discuss three techniques that reduce the effects of global transactions on the latency of local transactions.

6.5 Delaying transactions

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

We can reduce t_i's effects on t_j as follows. When s receives t_i's termination request (message 1 in Figure 6.1), 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 6.1), then delaying the broadcast of t_i at p by Δ will not increase t_i's overall latency.

6.5.1 The algorithm in detail

Algorithm 7 shows the delaying of transactions for a server s in partition p. To delay a transaction, the only part that is affected is when server s receives a request to commit transactions t. In which case s broadcasts t to each one of the partitions involved by t, possibly delaying the broadcast at partition p. In Algorithm 7, function *delay*(x,p) of line 5 returns the estimated latency between partitions x and p.

Algorithm 7 Transaction delaying, server s in partition p

1: **when** receive($commit, t$)
2: let P be $partitions(t) \setminus \{p\}$ *{broadcast t to each...}*
3: **for all** $x \in P : abcast(x,t)$ *{...remote partition}*
4: $\Delta \leftarrow max(\{delay(x,p) \mid x \in P\})$ *{determine maximum delay}*
5: $abcast(p,t)$ after Δ time units *{delay local broadcast}*

6.6 Reordering with fixed threshold

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

We ensure that at partition p local transaction t_j can be reordered with previously delivered pending transactions $t_{i_0}, ..., t_{i_M}$ using a reordering condition similar to the one presented in [47], originally devised to reduce the abort rate of concurrent transactions. In our context, we define that t_j can be serialized at position l if the following holds:

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

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

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

Consider now an execution where t_i is pending at server s when t_j is delivered and let t_i read and write item x and t_j read and write item y. Thus, s can reorder t_j before t_i in order to speed up t_j's termination. At server s', before t_j is delivered s' receives all votes for t_i and commits t_i. The result is that

when s' delivers t_j, it will not reorder t_j before t_i since t_i is no longer a pending transaction at s'. Although t_i and t_j modify different data items, servers must commit them in the same order to avoid non-serializable executions. For example, a transaction that reads x and y at s could observe that t_j commits before t_i and another transaction that reads x and y at s' could observe that t_i commits before t_j.

To guarantee deterministic reordering of transactions, we introduce a *reordering threshold* of size k per pending global transaction t_i. Transaction t_i's reordering threshold determines that (a) only local transactions among the next k transactions delivered after t_i can be reordered before t_i; and (b) s can complete t_i only after s receives all votes for t_i and s has delivered k transactions after t_i. In the previous example, if we set $k = 1$, then server s' would not complete t_i after receiving t_i's votes from other partitions, but would wait for the delivery of t_j and, similarly to server s, s' would reorder t_j and t_i.

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

6.6.1 The algorithm in detail

Algorithm 8 shows the mechanism in detail. The algorithm overrides functions *readyToCommit*() and *certify*(). In addition, the algorithm keeps *DC*, the number of delivered transactions, which is updated whenever a transactions is certified (line 6)).

Function *certify*() certifies t against transactions that committed after t started (lines 7–8), using the usual certification test performed by function *ctest*(). This check distinguishes between local and global transactions: while a local transaction has its readset compared against the writeset of committed transactions, a global transaction has both its readset and writeset compared against committed transactions (see Section 5.4 for a description of why this is needed). If some conflict is found, t must abort (line 8); otherwise the check continues.

A local transaction t is reordered among pending transactions (lines 15–24). The idea is to find a position for t in the pending list as close to the beginning of the list as possible, since that would allow t to leap over the maximum

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

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

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

6.7 Reordering with broadcasting of votes

In the reordering mechanism described earlier, each partition relies on an agreed upon reordering threshold to ensure a deterministic decision within a partition. We next describe an alternative mechanism to perform reordering. The idea is

Algorithm 8 Certification with fixed threshold, server s in partition p

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

to use the atomic broadcast primitive to agree on a deterministic ordering of global transactions within each partition. Upon certification, local transactions either commit or abort right away. A local transaction t commits right away if it passes the usual checks: t does not conflict with concurrent transactions that are already committed, and t does not invalidate votes of pending global transactions. Otherwise t aborts. The mechanism performs the same certification checks on global transactions. In addition, the protocol atomically broadcasts the final outcome of global transactions within partitions. Upon delivery, global transactions are committed or aborted according to the final outcome. This ensures a deterministic ordering of global transactions.

Differently from the previous reordering protocol, this mechanism aims at

always reordering local transactions at the beginning of the list of pending transactions, so that local transactions never wait for a global transaction to finish. The downside of this mechanism is that global transactions become more expensive, in that the termination protocol requires two invocations of the atomic broadcast primitive. With respect to the deployments described in Section 6.3, the termination of global transactions based on broadcasting of votes requires $6\delta + 2\Delta$ in the case of deployment WAN 1, and $3\delta + 5\Delta$ in the case of deployment WAN 2.

6.7.1 The algorithm in detail

Algorithm 9 shows the termination protocol in detail. With respect to Algorithm 5 of Chapter 5, we override function *certify*() and the handler for receiving votes from other partitions (lines 1–6).

Function *certify*() (lines 10–19) performs similarly as in Algorithm 5, the only significant change is that a committing local transaction is committed immediately, and is never included in the pending list.

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

6.8 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 in the chapter.

6.8.1 Setup and benchmarks

We ran the experiments using Amazon's EC2 infrastructure. We used r3.large instances equipped with two virtual cores and 15 GB of RAM. We deployed servers in three different regions: Ireland (EU), N. Virginia (US-EAST), and

Algorithm 9 Certification with broadcasting votes, server s in partition p

```
1:  when receive(tid, v) from partition q
2:     VOTES ← VOTES ∪ (tid, q, v)                    {one more vote for tid}
3:     let t be the transaction in PL such that t.id = tid
4:     if readyToCommit(t) then
5:        outcome ← (t.id, ⋆, abort) ∈ VOTES          {one abort vote and t will be aborted}
6:        abcast(t.id, outcome) to partition p
7:  when adeliver(tid, outcome)
8:     let t be transaction in PL such that t.id = tid
9:     complete(t, outcome)
10: function certify(t)
11:    if ∃t′ ∈ DB[t.st[p] ... SC] : ctest(t, t′) = false then
12:       return abort                                 {t aborts if it conflicts with committed t′}
13:    if ∃t′ ∈ PL : (t.rs ∩ t′.ws ≠ ∅) ∨ (t.ws ∩ t′.rs ≠ ∅) then
14:       return abort                                 {t aborts if it conflicts with pending t′}
15:    if t is global then
16:       PL ← PL ⊕ t                                  {append t to pending list if no conflicts}
17:    else
18:       complete(t, commit)                          {commit local transactions right away}
19:    return commit
```

Oregon (US-WEST). Using `ping`, we observed the following inter-region round-trip latencies: (a) ≈100 milliseconds between US-EAST and US-WEST, (b) ≈90 milliseconds between US-EAST and EU, and (c) ≈ 170 milliseconds between US-WEST and EU.

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.

Clients select keys randomly using a uniform distribution. We use one million data items per partition, where each data item is a 4-byte integer.

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

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

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

We report throughput and latency corresponding to 70% of the maximum performance, for both benchmarks. In all experiments, we generate one new snapshot every second, using the mechanism described in Section 5.6, unless stated otherwise. Each experiment lasts 80 seconds; we discard 10 seconds from the beginning and the end of the execution, leaving 60 seconds worth of traces per experiment.

6.8.2 Baseline

We deployed S-DUR in a geographically distributed environment following the two alternatives (WAN 1 and WAN 2) discussed in Section 6.3. Figure 6.2 shows the throughput and latency for both WAN 1 and WAN 2 deployments with workload mixes containing 0%, 1%, 10% and 50% of global transactions. Latency values correspond to their 99-th percentile and average. Figure 6.3 shows the cumulative distribution function (CDF) of latency for 0% and 1% of global transactions.

Global transactions have a clear impact on the system's throughput; as expected the phenomenon is more pronounced in WAN 1 than in WAN 2 (see

Section 6.4). In the absence of global transactions, local transactions can execute within 10 milliseconds in WAN 1. The latency of locals increases to 160 milliseconds with 1% of global transactions, a 16x increase. We observed that in workloads with 10% and 50% of global transactions, latency of locals increase to 173 milliseconds (17x increase to the 0% configuration) and 184 milliseconds (18x increase), respectively. This shows that the convoy phenomena deteriorates as we increase the fraction of global transactions.

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

The CDFs of Figure 6.3 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 6.4).

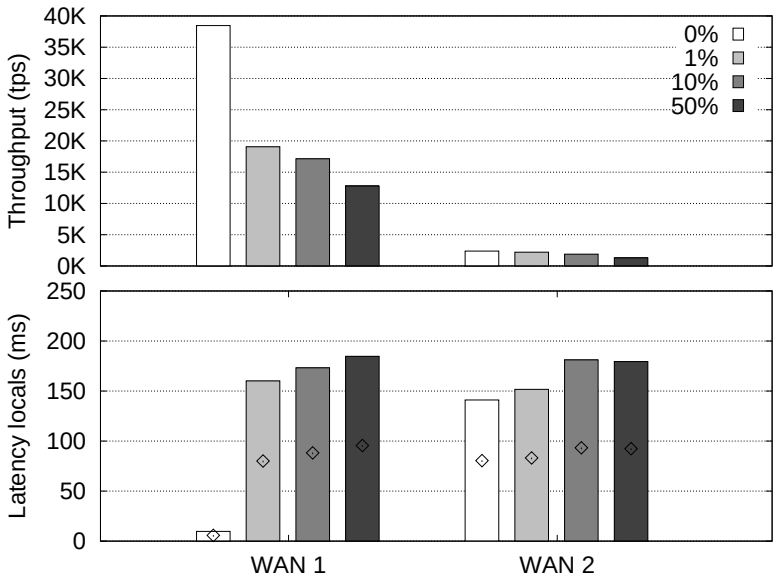

Figure 6.2. S-DUR's local and global transactions in two geographically distributed deployments (WAN 1 and WAN 2). Throughput in transactions per second (tps), latency 99-th percentile (bars) and average latency (diamonds in bars) in milliseconds (ms).

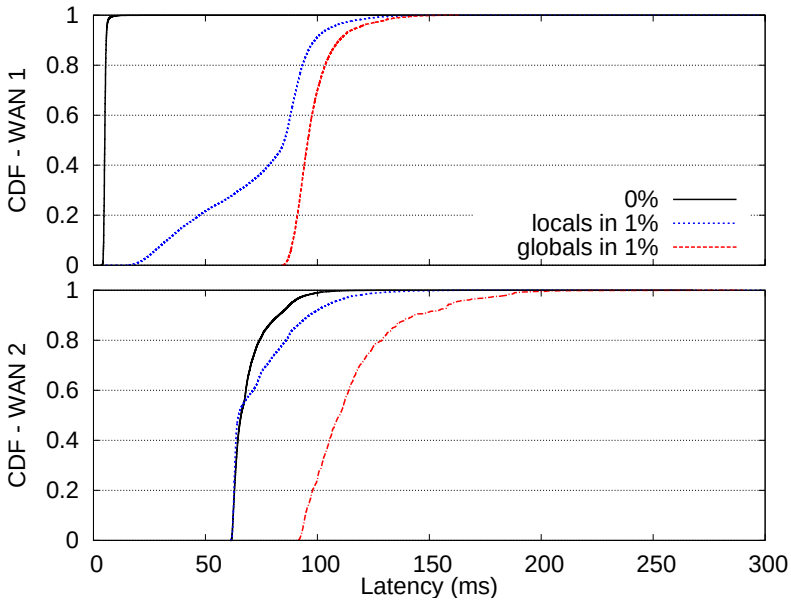

Figure 6.3. S-DUR's local and global transactions in two geographically distributed deployments (WAN 1 and WAN 2). Cumulative distribution functions (CDF) of latencies for 0% and 1% of global transactions.

6.8.3 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 6.4 shows that the technique has a positive effect in all percentages of global transactions we considered. Delaying globals by 40 milliseconds resulted in a reduction in the 99-th percentile latency of 30, 30.8, and 36 milliseconds for 1%, 10%, and 50% of global transactions respectively. Moreover, we noticed no impact on the overall latency of global transactions (Figure 6.5).

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

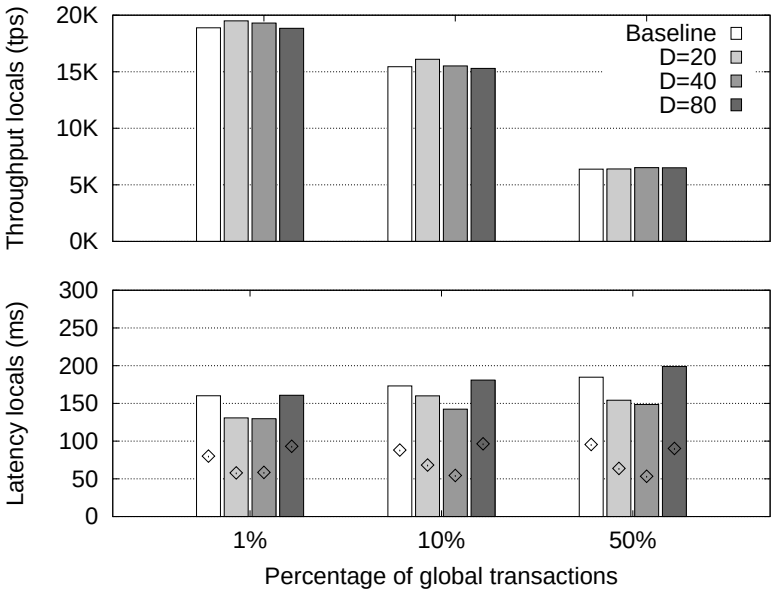

Figure 6.4. Local transactions with delayed transactions in WAN 1.

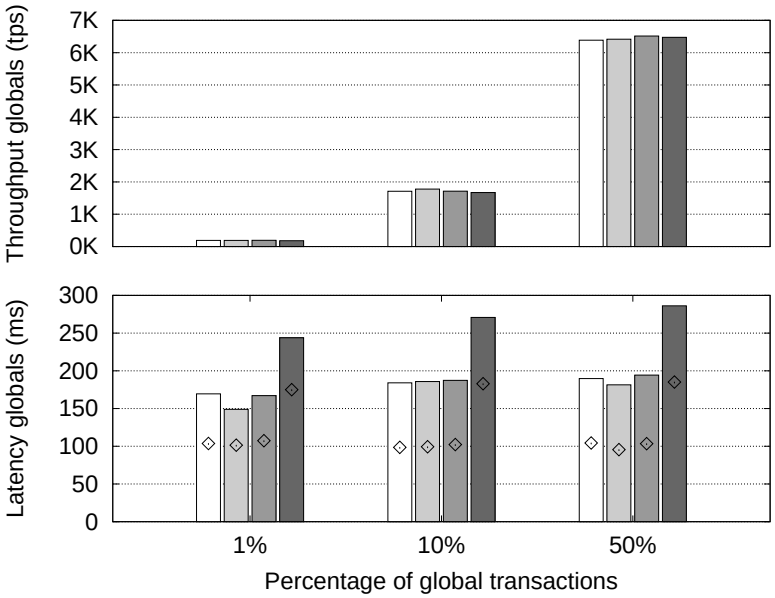

Figure 6.5. Global transactions with delayed transactions in WAN 1.

6.8.4 Reordering transactions

Figures 6.6 and 6.9 show the effects of reordering in the latency of local and global transactions under various workloads in WAN 1. We assess different reordering thresholds in configurations subject to a similar throughput. We compare the various reordering thresholds with reordering based on the broadcasting of votes.

In WAN 1, reordering has a positive impact on local transactions for all three workload mixes (Figure 6.6). For example, for 1% global transactions, a reordering threshold of 640 reduces the 99-th percentile latency of local transactions from 160 ms (in baseline) to 114 milliseconds, a 29% improvement. For mixes with 10% and 50% of global transactions the improvement is 24% and 30% respectively. The best results are achieved with reordering with broadcasting of votes, where the improvement is always greater than 90%, regardless of workload mix. We observed no significant impact on the 99-th percentile latency of the corresponding global transactions, for both reordering techniques (Figure 6.7).

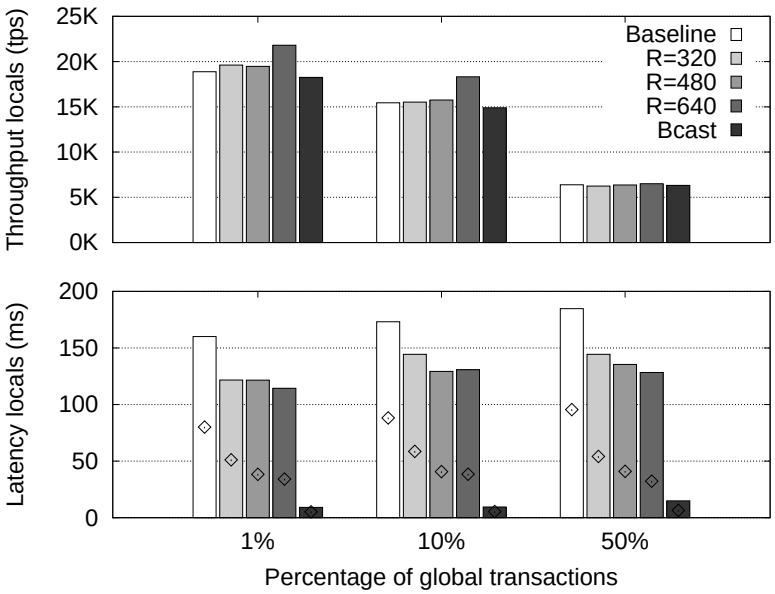

Figure 6.6. Local transactions with reordering in WAN 1.

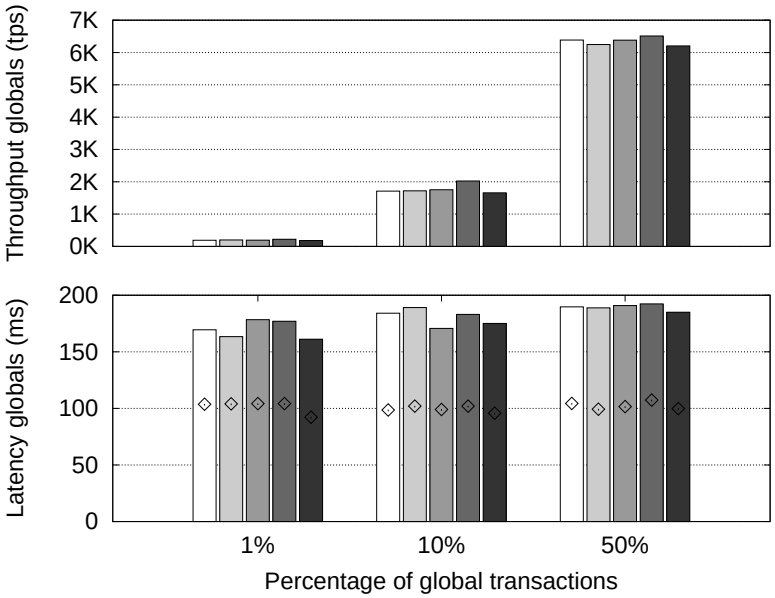

Figure 6.7. Global transactions with reordering in WAN 1.

Local transactions in WAN 2 (Figure 6.8) also benefit from reordering. Although, there is a tradeoff between the latency of locals and globals, an effect not seen for WAN 1. For example, in the workload with 10% of global transactions, a reordering threshold of 80 reduced the 99-th percentile latency of local transactions from 181 milliseconds (in baseline) to 130 milliseconds. The corresponding latency of global transactions increased from 211 milliseconds to 238 milliseconds (Figure 6.9). Reordering with broadcasting of votes achieves the best results for local transactions, though it also experiences the worst latency for global transactions. This is not surprising, as broadcasting the votes is expensive across geographical regions (atomic broadcast requires two more communication steps). Similar trends are seen for workloads with 1% and 50% of global transactions.

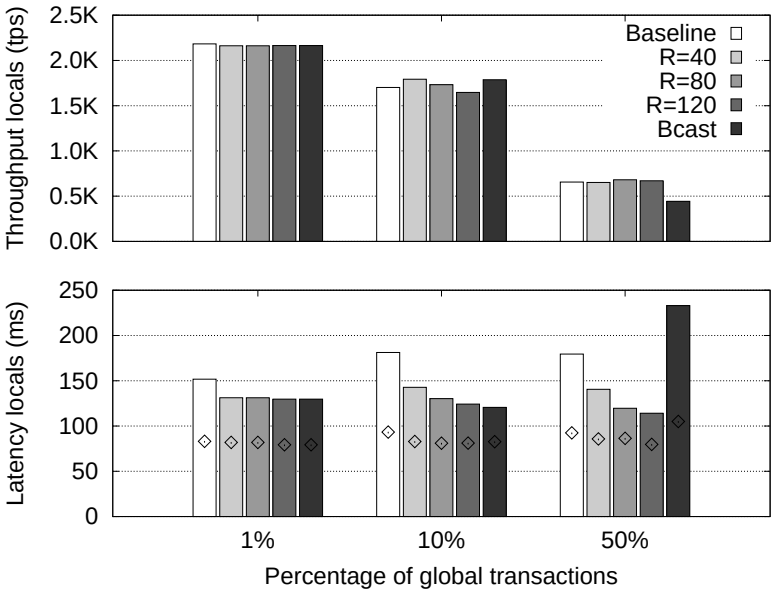

Figure 6.8. Local transactions with reordering in WAN 2.

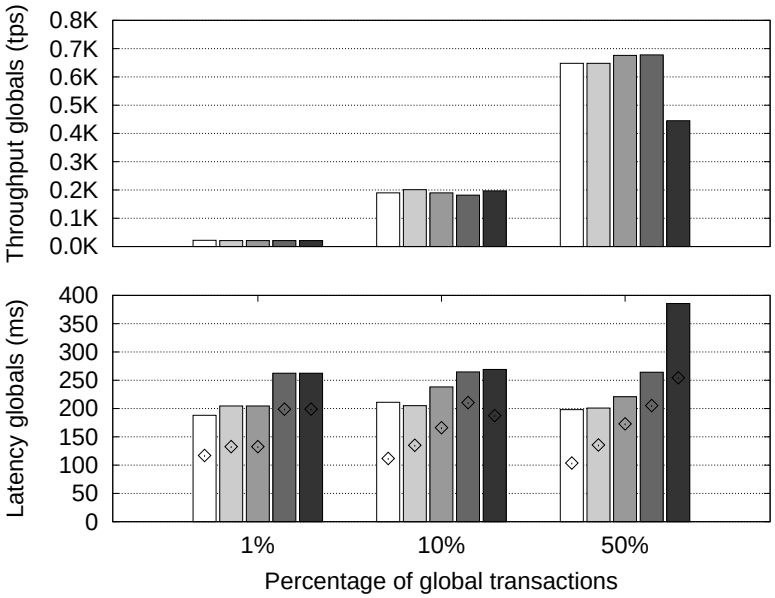

Figure 6.9. Global transactions with reordering in WAN 2.

6.8.5 Social network application

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

In WAN 1 (Figure 6.10), reordering with broadcasting of votes performs best. It achieves the best improvement on latency of local Post and Follow transactions (88% and 85% respectively), with minimal impact on the latency of global Follow transactions (less than 5%).

In WAN 2 (Figure 6.11), reordering with broadcasting of votes achieves the best latency for local Post and Follow transactions, although it is slower in committing global Follow transactions. As explained in the previous experiment, reordering with broadcasting of votes requires an additional invocation of atomic broadcast across wide-area links. In this deployment, reordering with a fixed threshold of size 60, achieves 40% improvement for local Post and Follow transactions, and only 7% increased latency on global Follow transactions.

In this experiment, Timeline transactions use the snapshot created with the snapshot mechanism described in Section 5.6. Timeline transactions perform equally well using both techniques: both reordering mechanisms have no impact on read-only transactions. The latency of Timeline transaction is low, even in WAN 2, because the snapshotting algorithm allows transactions to read data items from the closest replica.

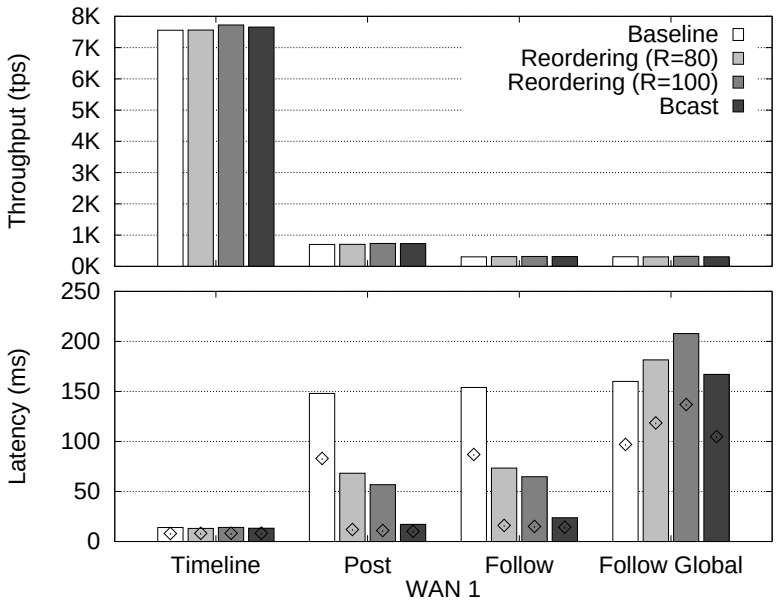

Figure 6.10. Social network application in WAN 1.

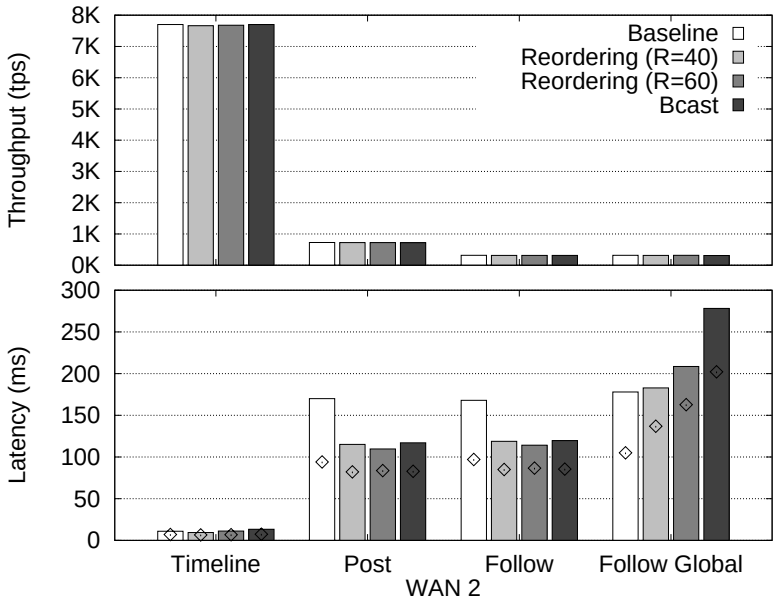

Figure 6.11. Social network application in WAN 2.

6.8.6 Impact of Snapshots

We next assess the impact of the snapshot mechanism described in Section 5.6. We consider average and 99-th percentile latency while varying the think time between invocations of the snapshot algorithm. Figures 6.12 and 6.13 shows the results for in WAN 1 and WAN 2, respectively.

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

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

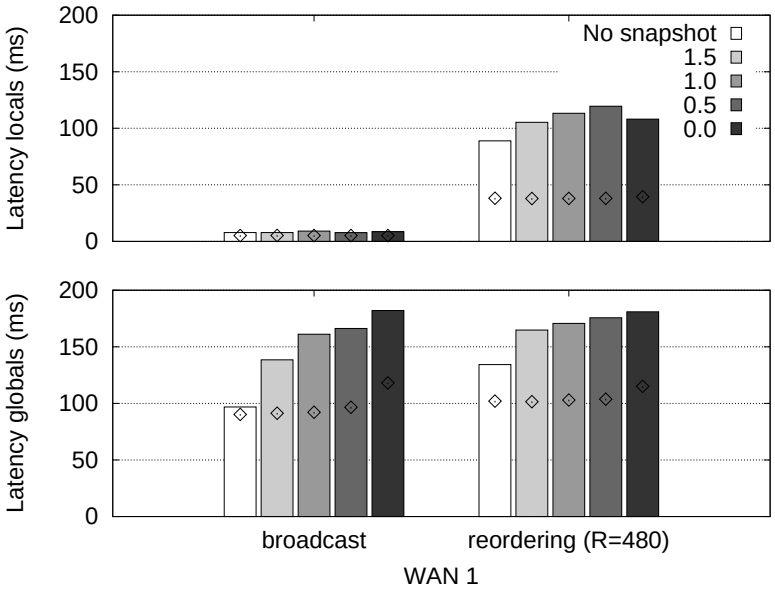

Figure 6.12. Impact of snapshots in WAN 1.

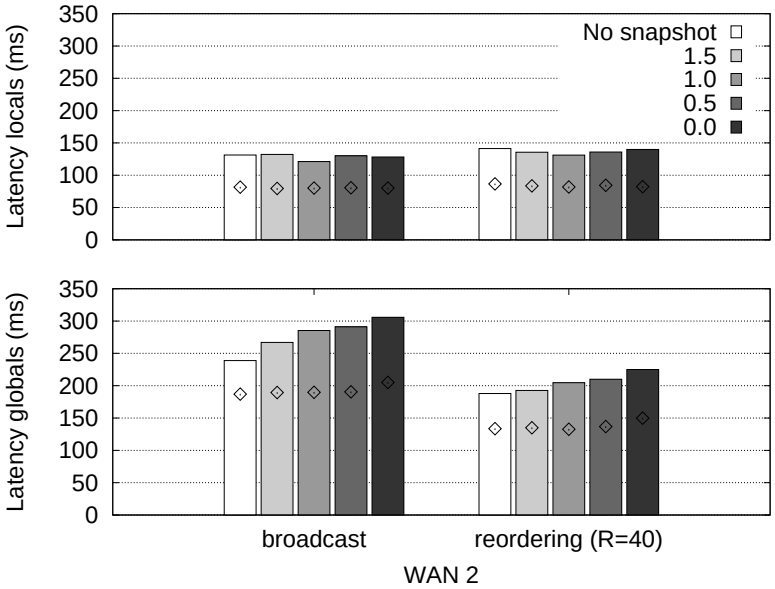

Figure 6.13. Impact of snapshots in WAN 2

6.9 Related Work

Many storage and transactional systems have been proposed recently. In the following we concentrate on systems that were designed to work well in geographically distributed environments.

Some of these systems (e.g., Cassandra [35], Dynamo [20], Voldemort[70]) 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.

COPS [39] is a 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. The Eiger system [40] improves over COPS in that it also provides atomic write-only transactions.

Walter [62] offers an isolation property called Parallel Snapshot Isolation (PSI) for databases replicated across multiple data centers. PSI guarantees snapshot isolation and total order of updates within a site, but only causal ordering across data centers.

PSI can be further weakened to a consistency criterion called Non-monotonic Snapshot Isolation (NMSI) [4]. Differently from PSI, NMSI allows a transaction to observe a snapshot that was committed after its start. While PSI assumes full replication, NMSI assumes partial replication [4].

Vivace [13] 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 [12] and Yahoo's PNUTS [14] 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 wide-area networks. We consider next systems that offer guarantees closer to S-DUR.

P-Store [55] is perhaps the closest to our work in that it implements de-

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

Spanner [15] is a distributed database for WANs. Like S-DUR the database is partitioned and replicated over several Paxos instances. Spanner uses a combination of two-phase commit and a 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.

Clock-SI [23] is similar to Spanner in that it also uses physical clocks to order transactions. Unlike Spanner, Clock-SI relies on loosely synchronized clocks, but does not provide replication.

MDDC [33] is a replicated transactional data store that also uses several instances of Paxos. MDCC optimizes for commutative transactions, and uses Generalized Paxos to relax the order of transaction delivery of commuting transactions.

Recently, a framework called G-DUR [5] has been proposed for developing and comparing protocols that use the deferred update replication technique. This work includes a comparison of S-DUR [57] (without the reordering techniques of Sections 6.6 and 6.7), P-Store [55], Serrano [58], Walter [62], NMSI [5], and GMU [48]. Their results confirm our considerations made in this thesis: protocols based on snapshot isolation (such as [58]) are not necessarily more efficient than protocols that provide serializability (such as S-DUR). However, weaker forms of Snapshot Isolation (such as PSI [62] and NMSI [4]) provide substantial gains. The use of atomic multicast in systems such as P-Store [55] is more expensive than the separate and independent atomic broadcasts of S-DUR. Slightly weakening read-only transactions with update serializability, as in GMU [48], also allows for substantial gains.

Our reordering technique with fixed threshold (Section 6.6) is based on the algorithm described in [47], originally designed for reducing the abort rate. In our context, we extend the idea to avoid the delay imposed by global communication on local transactions.

The transactions delaying technique of Section 6.5 increases the likelyhood that transactions are delivered approximately at the same time in every partition. A similar problem has been explored in the context of Optimistic Atomic Broadcast protocols. These protocols take advantage of the spontaneous or-

der in which networks deliver messages so that the computation is overlapped with the final total order delivery. While spontaneous ordering is highly probable in local area networks, it is not tipically the case in wide area networks. Spontaneous order can be achieved by injecting artificial delays [61; 50] or by timestamping messages (using loosely synchronized clocks) optimistically deliver in timestamp order [9].

6.10 Conclusion

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

Chapter 7

Conclusions

Strongly consistent transactions have been historically considered too expensive. Implementing such guarantees requires coordination across machines and has been considered impossible to be used at scale. However, we have recently witnessed a resurgence of interest, both industrial and academic, in strongly consistent transactional data stores. The main reason for this trend is that strong consistency leads to semantically simpler APIs. In other words, it is simpler to implement a large system on top of stronger guarantees. This thesis focuses on transactional replication protocols. We started from the baseline Deferred Update Replication protocol, and devised extensions that improve its performance, scalability and latency in both local, and wide-area networks.

7.1 Contributions

This thesis contributes three replication protocols motivated by current hardware and deployment trends: (i) RAM-DUR optimizes for reads and in-memory execution; (ii) S-DUR employs partial replication to offer scalability and strong consistency; and finally (iii) reordering termination in S-DUR provides substantial latency improvements in geographically distributed environments. The thesis describes each of these protocols, their implementation, and performance evaluation.

In-memory Deferred Update Replication. RAM-DUR extends the baseline Deferred Update Replication approach with volatile nodes, or *vnodes*. Vnodes implement a distributed caching layer which allows for fast in-memory transaction execution. The protocol behind vnodes is built around the assumption that retrieving a data item from a remote server is more efficient than accessing

a local disk. Vnodes store a subset of the dataset in memory and retrieve data items from remote servers during the execution phase of transactions. The resulting protocol guarantees the consistency level of the baseline approach.

Scalable Deferred Update Replication. S-DUR is a scalable Deferred Update Replication protocol. The assumption here is that the database can be partitioned such that most transactions are local to one partition, that is they access data items from one partition only. Local transaction achieve best performance, in that no coordination is required across partitions. The protocol behind S-DUR allows multi-partition transactions to be committed using only one roundtrip across the partitions involved. The resulting protocol guarantees the same consistency level of DUR. The resulting protocol is less expensive than previous protocols with similar guarantees [27; 55].

Geo-distributed Deferred Update Replication. Globally distributed applications impose challenging requirements on transactional systems. In particular, the high latency of wide-area networks is detrimental to coordinate strongly consistent systems. We investigated the performance of S-DUR when deployed in a wide-area network, and found that the commitment of a global transaction may delay the commitment of local transaction by up to two roundtrips. We introduced two reordering algorithms. The first is based on a threshold that fixes the number of local transactions that have to be committed before a global transaction can terminate. The second algorithm uses one extra invocation of the atomic broadcast primitive to order the vote messages of global transactions. Both techniques result in a considerable reduction of latency, especially in deployments where global transactions harm local transactions the most.

7.2 Future Directions

Several aspects of this thesis are worth further exploration.

Combining RAM-DUR and S-DUR. We introduced two techniques to improve the performance of deferred update replication. S-DUR improves scalability by partitioning the system into multiple smaller partitions, while RAM-DUR allows fast in-memory execution. We believe that the two approaches can be combined by simply applying RAM-DUR's techniques within S-DUR partitions. This approach would improve the execution phase of local transactions in S-DUR. In addition, RAM-DUR vnodes can be added and removed online, thus

making it possible to resize partitions based on the actual client load.

Dealing with non-uniform workloads. Database workloads are often non-uniform. That is, some data items may be accessed more frequently than others. Such workloads are generally problematic for partitioned databases like S-DUR. For instance, if the workload mostly accesses one of the partitions only the performance is bound to whatever a single partition can handle. A simple load balancing scheme could spread heavily accessed items to the available partitions. However this might not be ideal, because moving data items to a partition might increase the percentage of global transactions. This problem calls for a load balancing scheme that is aware of the possible consequences on transaction execution when moving items from one partition to another.

Reconfiguration of S-DUR partitions. In Chapters 5 and 6 we considered a fixed partitioning of the data. That is, the number of partitions, and the partitioning function is fixed. Changing the number of partitions in a S-DUR deployment while the system is running, is not trivial problem to solve efficiently. Ideally, a solution to this problem would be relatively quick in increasing or decreasing the number of partitions, have minimal impact on the performance of the running system, and allow ongoing transactions to terminate without interruption.

Appendix A

RAM-DUR: Proof of correctness

We show next that RAM-DUR only produces serializable schedules. In the proof, we argue that every history h produced by RAM-DUR has an acyclic multi-version serialization graph (MVSG) [8]; if $MVSG(h)$ is acyclic, then h is view equivalent to some serial execution of the transactions in h [8].

We first prove the following property about RAM-DUR.

Proposition 1 *(No holes.) If a lookup(k,st) request results in tuple $\langle k,v,x\rangle$, where $x \leq st$, then no committed transaction creates tuple $\langle k,u,y\rangle$ such that $x < y \leq st$.*

PROOF: Assume for a contradiction that at server s, a lookup(k,st) returns $\langle k,v,x\rangle$, $x \leq st$, and there is a transaction t that creates entry $\langle k,u,y\rangle$ and $x < y \leq st$.

From Algorithm 3, there are two cases to be considered:

Case (a): $\langle k,v,x\rangle$ is stored locally. When s issues a retrieve(k,st) request, it returns the most recent version smaller than or equal to st from the store; thus, entry $\langle k,u,y\rangle$ must not be in the store when s executes the retrieve operation. But when s executes retrieve(k,st), it has already certified every transaction that creates snapshot $y \leq st$.

We divide case (a) in three sub-cases (Rule 2):

- *Case (a.1)* s is the owner of k. Then it must store every update to k, including version y. Thus, we conclude that s is not k's owner.

- *Case (a.2)* s caches entry $\langle k,v,x\rangle$. Then either (a.2.1) s received from r, k's owner, cacheable entry $\langle k,v,x\rangle$ or (a.2.2) s cached an earlier version of k and delivered update $\langle k,v,x\rangle$. In case (a.2.1), by the algorithm, when r replied to the remote lookup request $(remote\text{-}lookup, k, SC_s, st)$ from s, v was the newest version stored locally and $SC_r \geq SC_s$ (Rule

3). It follows that there is no version y of k such that $x < y \leq SC_s$, a contradiction since $st \leq SC_s$. Case (a.2.2) is similar to (a.1).

- *Case (a.3)* s garbage collected $\langle k, u, y \rangle$. A contradiction to Rule 4, because there exists older version x of k which was not garbage collected.

Case (b): $\langle k, v, x \rangle$ is not stored locally. Therefore, s sends message $(remote\text{-}lookup, k, SC_s, st)$ to server r, the owner of k, and it must be that $st \leq SC_s$. Server r only executes s's request when $SC_r \geq st$ (Rule 1). Hence, r returns $\langle k, u, y \rangle$, and not version x, a contradiction that concludes the proof. □

Theorem 1 *RAM-DUR guarantees serializability.*
PROOF: MVSG is a directed graph, in which the nodes represent committed transactions. There are three types of directed edges in MVSG: (a) read-from edges, (b) version-order edges type I, and (c) version-order edges type II. These types are described below. We initially consider update transactions only, i.e., edges that connect two update transactions. Then we consider read-only transactions.

Update transactions. From the algorithm, the commit order of transactions induces a version order on every data item: if transactions t_i and t_j create entries $\langle k, v_i, ts_i \rangle$ and $\langle k, v_j, ts_j \rangle$, respectively, then $SC(t_i) < SC(t_j) \Leftrightarrow ts_i < ts_j$, where $SC(t)$ is the snapshot counter associated with transaction t and corresponds to its commit order.

To show that $MVSG(h)$ has no cycles, we prove that for every edge $t_i \rightarrow t_j$ in $MVSG(h)$, it follows that $SC(t_i) < SC(t_j)$. The proof continues by considering each edge type in $MVSG(h)$.

1- Read-from edge. If t_j reads $\langle k, v_i, ts_i \rangle$ from t_i, then $t_i \rightarrow t_j \in MVSG(h)$.

We have to show that $SC(t_i) < SC(t_j)$. From the algorithm, $ts_i = SC(t_i)$, which is the value of global counter SC at server s when t_i was certified at s. Since transactions only read committed data from other transactions, and t_j reads an entry from t_i, t_j is certified after t_i is certified. For each transaction that passes certification, s increments SC, and thus, it must be that $SC(t_i) < SC(t_j)$.

2- Version-order edge type I. If t_i and t_j create entries $\langle k, v_i, ts_i \rangle$ and $\langle k, v_j, ts_j \rangle$, respectively, such that $ts_i < ts_j$, then $t_i \rightarrow t_j \in MVSG(h)$.

Since the commit order induces the version order, we have that $ts_i < ts_j \Leftrightarrow SC(t_i) < SC(t_j)$.

3- Version-order edge type II. If t_i reads $\langle k, v_k, ts_k \rangle$ from t_k and t_j creates entry $\langle k, v_j, ts_j \rangle$ such that $ts_k < ts_j$, then $t_i \rightarrow t_j \in MVSG(h)$.

Since no two transactions have the same commit timestamp, either $SC(t_i) < SC(t_j)$ (i.e., what we must show), or $SC(t_j) < SC(t_i)$. For a contradiction, assume the latter, which together with $ts_k < ts_j$ leads to (a) $SC(t_k) < SC(t_j) < SC(t_i)$. Since t_i reads from t_k, it follows that (b) $SC(t_k) \leq ST(t_i)$, where $ST(t)$ is the database snapshot version seen by t.

From (a), (b), and the fact that $ST(t_i) < SC(t_i)$ (i.e., the database snapshot version seen by t_i must precede the version t_i creates), we have to show that the cases that follow cannot happen (see Figure A.1):

Case (i): $SC(t_k) \leq ST(t_i) < SC(t_j) < SC(t_i)$. In this case, t_j's writeset must be considered when certifying t_i since t_j commits before t_i is certified. t_i can only commit if its readset does not intersect t_j's writeset, but since $t_i.rs \cap t_j.ws = \{k\}$, t_i fails certification, a contradiction.

Case (ii): $SC(t_k) < SC(t_j) < ST(t_i) < SC(t_i)$. When t_i reads key k, it receives $\langle k, v_k, SC(t_k)\rangle$ from the storage, and not the value created by t_j. The read operation is translated into a lookup$(k, ST(t_i))$ and returns $\langle k, v_k, SC(t_k)\rangle$, where $SC(t_k) < ST(t_i)$. Thus, from Proposition 1 (no holes), there is no transaction that creates entry $\langle k, -, SC(t_j)\rangle$, a contradiction.

Read-only transactions. Let t_q be a read-only transaction in h. Since t_q does not update any data item, any edge involving t_q in $MVSG(h)$ is of the type either (a) read-from edge: $t_i \rightarrow t_q$ or (b) version-order type II: $t_q \rightarrow t_j$, where t_i and t_j are update transactions. We show that the former implies $SC(t_i) < ST(t_q)$ and the latter implies $ST(t_q) < SC(t_j)$.

4- Read-from edge. Since $t_i \rightarrow t_q \in MVSG(h)$, t_q must read some data item written by t_i. Since only committed versions can be read, it follows that $SC(t_i) < ST(t_q)$.

5- Version-order edge type II. Since $t_q \rightarrow t_j \in MVSG(h)$, there must exist some transaction t_k such that both t_k and t_j create entries $\langle k, v_k, ts_k\rangle$ and $\langle k, v_j, ts_j\rangle$, where $ts_k < ts_j$, and t_q reads $\langle k, v_k, ts_k\rangle$. By an argument similar to case (3) above, it must be that $ST(t_q) < SC(t_j)$.

The proof continues by contradiction: assume t_q is involved in a cycle c in $MVSG(h)$. Then, for each edge $t_a \rightarrow t_b \in c$, $SC(t_a) < SC(t_b)$ if both t_a and t_b are update transactions (from the first part of the proof), $SC(t_a) < ST(t_b)$ if t_b is a read-only transaction (from case (4) above), and $ST(t_a) < SC(t_b)$ if t_a is a read-only transaction (from case (5) above). Thus if c exists, it follows that $ST(t_q) < ST(t_q)$, a contradiction that concludes the proof. □

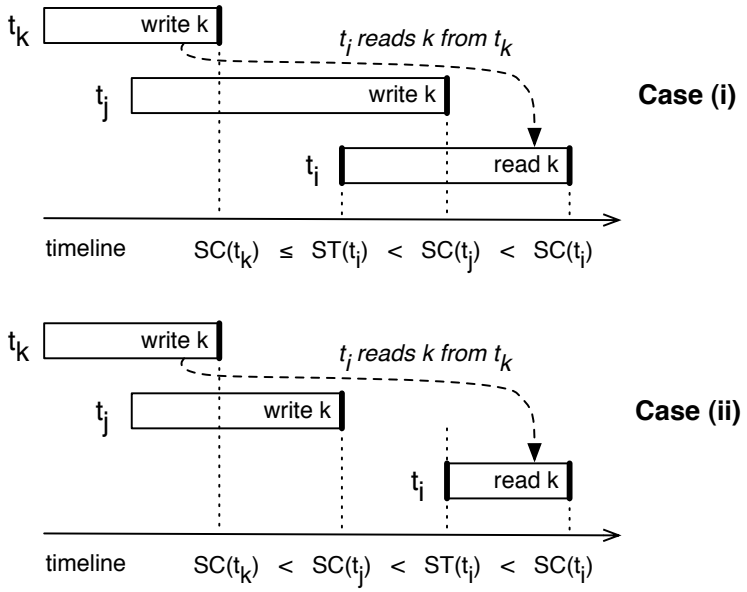

Figure A.1. Instances of cases (i) and (ii) in proof.

Appendix B

S-DUR: Proof of correctness

B.1 Scalable Deferred Update Replication

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 B.1).

- *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.

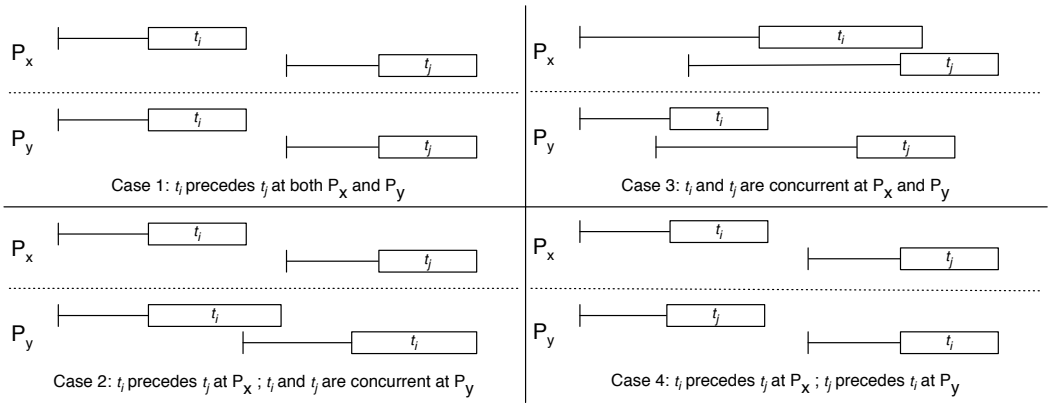

Figure B.1. Interleaved executions of transactions (Cases 1–3 are possible under S-DUR but not Case 4).

B.2 Read-only snapshots

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

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

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

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

- (b.1) Processes in partition j delivered T after receiving the last snapshot marker that created S. Thus, processes in j send S's snapshot marker before sending T's vote, but from (a), processes in i received T's vote from j before the marker from j, which contradicts the property of FIFO channels.

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

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

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

Appendix C

Geo-DUR: Proof of correctness

C.1 Delaying transactions

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

C.2 Reordering with fixed threshold

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

For claim (a) above, from Algorithm 8, the reordering of a local transaction t (lines 5 – 24) is a deterministic procedure that depends on $DB[t.st[p]\ldots SC]$ (line 7), *PL* (lines 16 – 20), and *DC* (line 18). We show next that *DB*, *PL*, *SC* and *DC* are only modified based on delivered transactions, which suffices to substantiate claim (a) since every server in p delivers transactions in the same order, from the total order property of atomic broadcast.

For an argument by induction, assume that up to the first i delivered transactions, *DB*, *PL*, *SC* and *DC* are the same at every correct server in p (inductive hypothesis), and let t be the $(i+1)$-th delivered transaction (line 5). *PL* is possibly modified in function *certify*() (line 23) and from the discussion above depends on *DB*, *PL*, *SC* and *DC*, which together with the induction hypothesis

we conclude that it happens deterministically. *DB*, *PL* and *SC* are also possibly modified in the *complete*() procedure (Algorithm 5, lines 24 – 29), called (i) after t is delivered (Algorithm 5, line 17), (ii) when the head of *PL* is a local transaction (Algorithm 5, line 20), and (iii) when the head of *PL* is a global transaction u (Algorithm 5, line 20).

In cases (i) and (ii), since all modifications depend on t, *PL* and *SC*, from a similar reasoning as above we conclude that the changes are deterministic. In case (iii), the calling of the *complete*() procedure depends on receiving all votes for t and t having reached its reorder threshold (line 4). From the induction hypothesis, all servers agree on the value of *DC*. Different servers in p may receive u's votes at different times but we will show that any two servers s and s' will nevertheless reorder t in the same way. Assume that when s assesses u it already received all u's votes and proceeds to complete u before it tries to reorder t. Another server s' assesses u when it has not received all votes and does not call the complete procedure. Thus, s will not reorder t with respect to u. For a contradiction, assume that s' reorders t and u. From the reorder condition, it follows that u has not reached its reorder threshold at s', which leads to a contradiction since u has reached its threshold at s, from the algorithm (line 6) *DC* depends only on delivered messages and from atomic broadcast all servers deliver the same transactions in the same order.

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

C.3 Reordering with broadcasting of votes

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

For claim (a) above, from Algorithm 9, the reordering of a local transaction

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

For an argument by induction, assume that up to the first i delivered transactions, *DB*, *PL* and *SC* are the same at every correct server in p (inductive hypothesis), and let t be the $(i+1)$-th delivered transaction (line 10). *PL* is possibly modified in function *certify*() (line 16) and from the discussion above depends on *DB*, *PL*, *SC* and *DC*, which together with the induction hypothesis we conclude that it happens deterministically. *DB*, *PL* and *SC* are also possibly modified in function *complete*() (Algorithm 5, lines 24 – 29), called (i) after t is delivered (Algorithm 5, line 17), (ii) when t is committed in function *certify*() (line 10), and (iii) when s delivers a final vote for global transaction u (line 7).

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

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

Bibliography

[1] Atul Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

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

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

[4] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 163–172, Washington, DC, USA, 2013. IEEE Computer Society.

[5] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 13–24, New York, NY, USA, 2014. ACM.

[6] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, and F. D. Muñoz Escoí. SIPRe: A Partial Database Replication Protocol with SI Replicas. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 2181–2185, New York, NY, USA, 2008. ACM.

[7] Jason Baker, Chris Bond, James C. Corbett, J. Furman, Andrey Khorlin, Jamesa Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: providing scalable, highly available storage for

interactive services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

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

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

[10] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

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

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[13] Brian Cho and Marcos K Aguilera. Surviving Congestion in Geo-Distributed Storage Systems. In *USENIX Annual Technical Conference*, pages 439–451, 2012.

[14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

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

[16] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on*

Dependable Computing, PRDC '09, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[17] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[18] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, New York, NY, USA, 2010. ACM.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

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

[21] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[23] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society.

[24] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, April 1988.

[25] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware Load Balancing and Update Filtering in Replicated Databases. *SIGOPS Oper. Syst. Rev.*, 41(3):399–412, March 2007.

[26] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[27] Udo Fritzke Jr and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 284–291. IEEE, 2001.

[28] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[29] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems (TODS)*, 7(2):209–234, 1982.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[32] Bettina Kemme and Gustavo Alonso. Don'T Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 134–143, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

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

[34] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[35] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

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

[37] LibPaxos. https://bitbucket.org/sciascid/libpaxos.

[38] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 419–430, New York, NY, USA, 2005. ACM.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

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

[41] Parisa Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A Highly Efficient Atomic Broadcast Protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 527–536, 2010.

[42] Diego Ongaro, Stephen M. Rumble, Ryan. Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[43] Leandro Pacheco, Daniele Sciascia, and Fernando Pedone. Parallel Deferred Update Replication. In *Proceedings of the 2014 IEEE 13th International Symposium on Network Computing and Applications*, NCA '14, pages 205–212, Washington, DC, USA, 2014. IEEE Computer Society.

[44] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.*, 23(4):375–423, November 2005.

[45] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[46] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, pages 513–520, London, UK, UK, 1998. Springer-Verlag.

[47] Fernando Pedone, Rachid Guerraoui, and André Schiper. The Database State Machine Approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[48] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 455–465, Washington, DC, USA, 2012. IEEE Computer Society.

[49] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

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

[51] O. T. Satyanarayanan and D. Agrawal. Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases. *IEEE Trans. on Knowl. and Data Eng.*, 5(5):859–871, October 1993.

[52] André Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, 1996.

[53] Nicolas Schiper and Fernando Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *Distributed Computing and Networking*, pages 147–157. Springer, 2008.

[54] Nicolas Schiper, Rodrigo Schmidt, and Fernando Pedone. Optimistic Algorithms for Partial Database Replication. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, OPODIS'06, pages 81–93, Berlin, Heidelberg, 2006. Springer-Verlag.

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

[56] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

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

[58] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting Database Replication Scalability Through Partial Replication and 1-Copy-Snapshot-Isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.

[59] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013.

[60] António Sousa, Rui Oliveira, Francisco Moura, and Fernando Pedone. Partial Replication in the Database State Machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, NCA '01, pages 298–, Washington, DC, USA, 2001. IEEE Computer Society.

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

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

[63] Michael Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres. *IEEE Trans. Softw. Eng.*, 5(3):188–194, May 1979.

[64] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[65] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J ElmoreA, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[66] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[67] Alexander Tomic. MoSQL: A Relational Database Using NoSQL Technology. Master's thesis, University of Lugano, 2011.

[68] Alexander Tomic, Daniele Sciascia, and Fernando Pedone. MoSQL: An Elastic Storage Engine for MySQL. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 455–462, New York, NY, USA, 2013. ACM.

[69] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[70] Voldemort. http://www.project-voldemort.com/.