Low-Overhead Distributed Transaction Coordination

by

James Cowling

S.M., Massachusetts Institute of Technology (2007)
B.C.S.T. (Adv.) H1M, The University of Sydney (2004)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2012

Certified by . . . .
Barbara H. Liskov
Institute Professor
Thesis Supervisor

Accepted by . . . .
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Low-Overhead Distributed Transaction Coordination

by

James Cowling

Submitted to the Department of
Electrical Engineering and Computer Science
on May 23, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

ABSTRACT

This thesis presents *Granola*, a transaction coordination infrastructure for building reliable distributed storage applications. Granola provides a strong consistency model, while significantly reducing transaction coordination overhead. Granola supports general atomic operations, enabling it to be used as a platform on which to build various storage systems, e.g., databases or object stores.

We introduce specific support for *independent transactions*, a new type of distributed transaction, that we can serialize with no locking overhead and no aborts due to write conflicts. Granola uses a novel timestamp-based coordination mechanism to serialize distributed transactions, offering lower latency and higher throughput than previous systems that offer strong consistency.

Our experiments show that Granola has low overhead, is scalable and has high throughput. We used Granola to deploy an existing single-node database application, creating a distributed database application with minimal code modifications. We run the TPC-C benchmark on this platform, and achieve 3× the throughput of existing lock-based approaches.

Thesis Supervisor: Barbara H. Liskov
Title: Institute Professor

ACKNOWLEDGMENTS

I'd like to thank Evan Jones for the many engaging technical discussions, and for providing the sourcecode that was used as the basis of our TPC-C implementation. I'd like to also thank the original H-Store authors, including Dan Abadi, for their contributions to this codebase.

I'd like to thank my thesis committee members, Sam Madden and Robert Morris, for their feedback on my work. I'd also like to thank the many anonymous reviewers along the way, who invested their time in giving constructive feedback on my thesis work, and also on my other projects throughout grad school.

I've greatly enjoyed working with a number of co-authors on various research projects, and have benefited significantly from their technical perspective. I'd also like to thank the members of the Programming Methodology Group at MIT, for technical discussions, feedback on my work, and their presence in the lab.

I've been fortunate to make many really wonderful friends while at MIT and I feel that the people I've met here have been at least as important as the work that I've done. I'd like to thank them for enriching my life, and am looking forward to many good times together after grad school.

I'd like to thank my fellow members of the EECS REFS program, and the many members of the MIT, EECS and CSAIL administrations that have taken an active interest in student support. I owe a special thanks to Dan Ports, for providing a great deal of support on both a technical and personal level. Dan always knew the right time to share his enthusiasm for my work, or to convince me that whatever "fundamental problem" I was struggling with wasn't so fundamental after all. It was extremely valuable to have someone there to offer continual encouragement.

My family have been tremendously supportive throughout my time at MIT, and I owe them a great deal of thanks. It hasn't always been easy living on the other side of the world, but they've always endeavored to bridge the gap between these two continents. I'd also like to thank my friends back in Australia, who've made a continual effort to keep in touch and remain an

active part of my life. I'd especially like to thank those who have flown all the way to America on a number of occasions.

I'd like to give a special thanks to Kim for the huge amount of support she's given me, and for being a tremendous positive influence in my life. I'd especially like to thank her for putting up with me being in the lab almost constantly. Having her spend time in the office with me, and be there during my downtime, has been invaluable.

I'd finally like to thank my advisor, Barbara Liskov, for her support, technical discussions, feedback, editing and hard work throughout my time at MIT. Barbara has helped shape the way I think about problems, and I'd like to thank her for instilling in her students a principled approach to systems design and research in general.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Distributed storage systems spread data and computation across multiple servers, and provide these resources to client applications. These systems typically partition their state among many nodes to provide fast access to data and to provide sufficient storage space for large numbers of clients.

There is a long history of research in distributed storage, but the past five years has been marked by explosive growth in the popularity of large-scale distributed storage systems [2–5, 7, 10, 12, 19, 20, 23, 26, 41, 49]. This has been motivated by the growing demands of cloud computing and modern large-scale web services.

Cloud computing platforms are typically designed to support large numbers of clients with diverse application requirements. Ideally these systems would satisfy the following base requirements:

Large Scale: An immense amount of information is being stored online, and demand for online storage systems is growing. Supporting these levels of storage requires a distributed architecture, which must be able to scale across large numbers of clients and a large number of storage nodes.

Fault Tolerance: Information entrusted to the system should not be lost, and should be accessible when needed. These reliability guarantees must be provided with very high probability.

High Performance: Reads and writes should execute with low latency and per-server throughput should be high.

Consistency: Clients should be provided the abstraction of one-copy serializability [14] so that client applications are not complicated by ill-defined consistency semantics. Operations on the storage state should execute atomically, and reads should always be guaranteed to observe a state at least as recent as the previous operation from the client.

The consistency requirement is of particular interest, since it is the requirement most often neglected in modern distributed storage systems. It is highly desirable to run operations as *atomic transactions*, since this greatly simplifies the reasoning that application developers must do. Transactions allow users to ignore concurrency, since all operations appear to run sequentially in some serial order. Users also do not need to worry that failures will leave operations incomplete.

Strong consistency is often considered prohibitively expensive, however, particularly for operations that involve data on multiple servers. Serializable distributed transactions are traditionally provided through the use of two-phase commit [28, 36] and strong two-phase locking [25]. This approach has several downsides, however. Latency is increased due to multiple message delays in the protocol, along with multiple forced-stable log writes required to allow recoverability from failures. Throughput can also suffer due to the overhead of locking, which can be computationally expensive and limit concurrency; we discuss the costs of locking in more detail in Section 2.2.

Most distributed storage systems do not provide serializable transactions, because of concerns about performance and partition tolerance. Instead, they provide weaker semantics, e.g., eventual consistency [23] or causality [41]. These weaker consistency models pose a challenge to application development, and limit the types of applications that can be built using the storage system.

1.1 GRANOLA

This thesis presents *Granola*,[1] an infrastructure for building distributed storage applications where data resides at multiple storage nodes. Granola provides strong consistency for distributed transactions, without the overhead of lock-based concurrency control. Granola provides scalability, fault tolerance and high performance, *without* sacrificing consistency.

Granola is a *transaction coordination* mechanism: it provides transaction ordering, atomicity and reliability on behalf of storage applications that run on the platform. Applications specify their own operations, and Granola does not interpret transaction contents. Granola can thus be used to support a wide variety of storage applications, such as file systems, databases, and object stores.

Granola implements atomic *one-round* transactions. These execute in one round of communication between a user and the storage system, and are used extensively in online transaction processing workloads to avoid the cost of user stalls [10, 32, 50]; we discuss one-round transactions in more detail in Section 2.1. Granola supports three different classes of one-round transactions: *single-repository* transactions, *coordinated* distributed transactions, and *independent* distributed transactions.

Single-Repository Transactions: These transactions execute to completion at a single storage node, and do not involve distributed transaction coordination. We expect the majority of transactions to be in this class, since data is likely to be well-partitioned.

Coordinated Transactions: These transactions execute atomically across multiple storage nodes, but require transaction participants to *vote* on whether to commit or abort the transaction, and will commit only if all participants vote to commit. These transactions are equivalent to what is provided by traditional two-phase commit.

[1] Granola derives its name from its role in guaranteeing a consistent *serial* (cereal) order for distributed transactions. The author apologizes for the bad pun.

Independent Transactions: This is a new class of transaction that we introduce in Granola to support lock-free distributed transactions. Independent transactions commit atomically across a set of transaction participants, but do not require agreement, since each participant will independently come to the same commit decision. Examples include an operation to give everyone a 10% raise, an atomic update of a replicated table, or a read-only query that obtains a snapshot of distributed tables.

There is evidence that a large number of typical workloads are comprised primarily of independent transactions [50, 53], such as the industry-standard TPC-C benchmark [8]. We discuss the use of independent transactions in Section 2.2.

One of Granola's key contributions is its use of a timestamp-based coordination mechanism to provide serializability for independent transactions *without locking*. Granola is able to achieve this by assigning each transaction a timestamp, and executing transactions in timestamp order. Each distributed transaction is assigned the same timestamp at all transaction participants, ensuring consistency between storage nodes. Granola adopts a distributed timestamp voting protocol to determine the timestamp for a given transaction, and utilizes clients to propagate timestamp ordering constraints between storage nodes.

Granola's timestamp-based coordination mechanism provides a substantial reduction in overhead from locking, log management and aborts, along with a corresponding increase in throughput. Granola provides this lock-free coordination protocol while handling single-repository transactions and independent transactions, and adopts a lock-based protocol when also handling coordinated transactions. Granola's throughput is similar to existing state-of-the-art approaches when operating under the locking protocol, but significantly higher when it is not.

Granola also uses transaction coordination protocols that provide lower latency than previous work that uses dedicated transaction coordinators, for all transaction classes. Unlike systems that use a single centralized

Transaction Class	Message Delays	Forced Log Writes	Locking
Single-Repository	2	1	False
Independent	3	1	False
Coordinated	3	1	True

Table 1.1: Properties of each transaction class. The number message delays includes communication with the client, but does not include delays required for using replication to provide forced log writes.

coordinator [31, 53], or a dedicated set of coordinator nodes [10, 54], Granola uses a fully-distributed transaction coordination protocol, with no need for centralized coordination. Granola's use of a decentralized coordination protocol reduces latency for distributed transactions, which no longer incur a round-trip to the coordinator.

Granola runs single-repository transactions with two one-way message delays plus a stable log write, and both types of distributed transactions usually run with only three one-way message delays plus a stable log write. This is a significant improvement on traditional two-phase commit mechanisms, which require at least two stable log writes, and improves even on modern systems such as Sinfonia [10], which requires at least four one-way message delays (for a remote client) and one stable log write. The performance characteristics of Granola's three transaction classes are summarized in Table 1.1.

Since Granola provides strong consistency, there may be cases where servers have to stall due to the failure of a transaction participant [16]; this is an unavoidable reality in distributed systems design [27]. Granola uses replication to minimize the likelihood that a participant will be unavailable. In the case of a correlated failure that leads to unavailability of an entire replicated node, Granola provides recovery techniques to enable progress for transactions that do not depend on the failed participant.

1.2 CONTRIBUTIONS

The main contributions of this thesis are as follows:

- Identifying an *independent transaction* model for distributed transactions that can be executed atomically without communicating commit or abort decisions between nodes. These transactions are applicable to a variety of workloads and allow a significant reduction in transaction coordination overhead.
- Developing a transaction coordination platform that handles the details of reliability and consistency on behalf of distributed storage applications built on the platform. This platform provides support for atomic one-round transactions and supports general operations without interpreting the contents of individual transactions.
- Designing and implementing a timestamp voting and propagation protocol to serialize independent transactions without locking and with no centralized transaction coordinator. This system is able to provide lower latency, higher throughput and lower overhead than existing lock-based approaches.

We implemented and tested Granola on a set of synthetic benchmarks, as well as the common TPC-C transaction processing benchmark [8]. These experiments exhibit a significant benefit from running with independent transactions, and lead to a 3× increase in throughput on TPC-C, as compared with existing lock-based approaches.

1.3 OUTLINE

This thesis discusses the design and implementation of the Granola platform, including the core protocol for distributed transaction coordination, along with the infrastructure for building distributed storage applications.

Chapter 2 describes our one-round transaction model in more detail, and provides additional motivation for the use of independent distributed transactions. Chapter 3 then describes Granola's architecture, application interface, and assumptions.

We describe the core Granola protocol in Chapter 4 and discuss its consistency properties. Chapter 5 then discusses the protocols for handling failures, including network partitions and long-term failure of transaction participants.

We present an experimental evaluation of Granola's performance in Chapter 6 on a set of microbenchmarks, along with an evaluation of Granola's performance on a more real-world workload, using the TPC-C transaction processing benchmark. We also present a theoretical analysis of some key aspects of the protocol.

Chapter 7 presents additional extensions and optimizations to the Granola protocol, both to assist in building practical applications on the platform and for improving performance.

We present a discussion of related work in Chapter 8, along with an overview of future work and our conclusions in Chapter 9.

We follow these chapters with a number of appendices: Appendix A provides a listing of the message formats used in Granola; Appendix B analyzes different methods for switching between transaction processing modes; and Appendix C discusses our original protocol for running Granola with centralized coordinator nodes.

2 TRANSACTION MODEL

This chapter discusses Granola's transaction model in more detail. We first describe our one-round transaction model. We follow this with a description of the types of applications for which independent transactions are applicable and a discussion of the motivation for supporting independent transactions, in terms of reducing locking overhead.

2.1 ONE-ROUND TRANSACTIONS

Granola adopts a particular style of transactions, which we call *one-round transactions*. These transactions are expressed in a single round of communication between the client and a set of storage nodes. We refer to these storage nodes throughout this thesis as storage *repositories*, and refer to the set of repositories involved in a given transaction as the transaction *participants*. We also use the term *transaction* throughout this thesis to refer to the one-round transactions used in Granola.

Each transaction is comprised of an application-specific operation to be run at the server application at each participant, which we refer to as an *operation* or *op*. The client specifies an operation to run at each participant as part of the transaction, and Granola ensures that these operations are executed atomically at all participants. Operations are described in more detail in Section 3.2.

One-round transactions are distinct from more general database transactions in two key ways:

1. One-round transactions do not allow for interaction with the client, where the client can issue multiple sub-statements before issuing a transaction commit. Granola instead requires transactions to be specified by the client application as a single set of operations.

2. One-round transactions execute to completion at each participant, with no communication with other repositories, apart from a possible commit/abort vote. Granola does not support remote nested transactions being issued by a transaction participant, or sub-queries being sent to other nodes.

Despite these restrictions, one-round transactions are still a powerful primitive. They are used extensively in online transaction processing workloads to avoid the cost of user stalls [10, 32, 50], and map closely to the use of stored procedures in a relational DBMS. One-round transactions in Granola are similar to *minitransactions* in Sinfonia [10], but can can be used to execute general operations at each repository, whereas Sinfonia enforces a more restrictive read/write interface with pre-defined locksets. One-round transactions are also similar to *one-shot transactions* in H-Store [50], although one-round transactions allow commit decisions to be communicated between participants.

Granola uses one-round transactions as its basic transaction primitive, and we expect that applications that are built on the platform will implement the majority of their transactions using this one-round model. Granola can also be used to implement interactive transactions, however, where each transaction consists of multiple communication phases between the client and repositories; we discuss how interactive transactions can be supported in Section 7.5.

2.2 INDEPENDENT TRANSACTIONS

Granola's protocol design and our support for independent transactions was motivated by the desire to provide distributed transactions *without locking*, while minimizing communication and message delays in the coordination protocol. Locking is typically used to ensure that once a server sends a "yes" vote for a transaction, it is guaranteed to be able to commit that transaction if all other participants vote to commit. Locking thus ensures that no other transaction can modify state that was used to determine a vote.

A key insight in this thesis is that, for many transactions, locking is not actually required in determining whether to commit or abort a transaction. In particular, if all participants are guaranteed to commit the transaction, then two-phase commit or two-phase locking are *not* required. We must instead just ensure that all participants execute their portion of the transaction in the same global serial order, guaranteeing serializability without the cost of locking or the possibility of conflicts.

Granola exploits lock-free transactions through the use of the independent transaction class. Instead of pre-executing the transaction to determine the commit vote, and then releasing locks at the commit point, independent transactions are executed atomically in a single shot once the transaction has committed. This new model of transactions allows us not only to avoid locking, but also to rethink the transaction coordination protocol, reducing the number of message delays and forced log writes.

Our approach was influenced by the H-Store project [50], which identified a large class of operations in real-world applications, deemed as *one-shot* and *strongly two-phase*, that fit our independent transaction model. However, that work did not provide a functional protocol for coordinating independent transactions in a distributed setting [31]. To our knowledge, no previous system provides explicit support for independent transactions.

We first describe the scenarios under which independent transactions can be used. We follow this with a discussion of our motivations for using independent transactions and the costs of using a lock-based alternative.

2.2.1 APPLICABILITY OF INDEPENDENT MODEL

The independent transaction class can be used for any one-round distributed transaction where all participants will make the same local decision about whether to commit or abort. The relevance of this model for typical online transaction processing workloads is argued in the H-Store project, for transactions that fit within H-Store's *one-shot* and *strongly two-phase* transaction classes [50]. Recent work on deterministic transaction ordering also argues for the relevance of transactions that can execute independently at each storage node [53, 54].

Any read-only one-round distributed transaction can be represented as an independent transaction, since each repository can independently return its portion of the transaction result, without affecting the state at other participants. These can comprise a significant fraction of transactions in typical read-heavy workloads. Examples of distributed read-only transactions include a transaction to read the total balance for a set of bank accounts distributed across multiple repositories, or any distributed snapshot read in general.

Any read-write one-round distributed transaction can also be expressed as an independent transaction if the commit decision is a deterministic function of state available at all participants. We discuss this case in more detail as follows.

In systems composed of multiple storage nodes, data is typically spread across nodes in two ways:

Replicated Data: Individual data items are stored on multiple servers, to provide higher transaction throughput for multiple simultaneous clients, or to allow local access to the data at multiple repositories, or to locate data geographically close to clients.[1]

Partitioned Data: Data is split among multiple servers to provide increased storage capacity and greater aggregate transaction throughput.

[1] Data is also typically replicated for the purposes of reliability; Granola internally replicates system state to provide durability and availability.

Updates to replicated data must be executed atomically, and can be performed using an independent transaction at all partitions that store a copy of the data. Application developers also commonly replicate tables when partitioning data, to ensure that the majority of transactions are issued to a single partition [32, 50]. Any transaction where the commit decision depends only on replicated data can be represented as an independent transaction, since each participant will have the same replicated data available to it, and hence will come to the same commit decision. One example of a transaction predicated on replicated data is a delivery order that will only commit if the client provides a valid zip code, where each participant has a local copy of the zip code table.

Transactions that modify partitioned data, but always commit, can be implemented using independent transactions. Many typical distributed transactions fit within this model, such as a transaction to atomically update a set of employee records to change their assigned supervisor, or to atomically give each employee a 10% raise. A transaction that records the winner of an online auction can be issued as an independent transaction that atomically updates the auction page on one partition and the user's bid history on another partition.

Transactions that update partitioned data do not all fall within the independent transaction class. In particular, a distributed transaction where the commit decision depends on data that is available only to a single repository may need to be implemented with a coordinated transaction, using lock-based concurrency control. However, a careful partitioning of application state can be used to convert many such transactions into independent transactions.

Escrow transactions [44] can be used to express many coordinated transactions as a single-repository transaction that first puts a quantity into *escrow*, followed by an independent transaction used to complete the transaction. One example of an escrow transaction is a bank transfer that first authorizes and reserves sufficient funds for the transfer in an escrow account, followed by an independent transaction which transfers the funds from the escrow account to the destination.

Demarcation [13] can be used to express some other coordinated transactions as single-repository transactions. Demarcation allows an invariant to be maintained across multiple partitions by setting demarcation limits at each partition, and only allowing a single-repository transaction to proceed if it does not violate these limits. One example is a hospital that is split across two branches on two repositories, and needs to have a minimum of 10 doctors in total on-call at all times. A demarcation limit of 5 doctors can initially be set at each branch. A doctor may be taken off-call at one branch without checking with the other branch, provided the number of doctors at the branch does not fall below this demarcation limit. If there are only 5 doctors left on call, a doctor cannot be taken off-call until a coordinated transaction is run to update the demarcation limits to 4 doctors at one branch and 6 at the other.

We discuss escrow transactions and demarcation in Section 9.1.2.

Case Study: TPC-C

The industry-standard TPC-C benchmark [8], which is designed to be representative of typical online transaction processing applications and represents a rather sophisticated database workload, can be implemented entirely using independent transactions. The H-Store position paper [50] describes a partitioning strategy used to express each TPC-C transaction as a one-round transaction.

There are five different transactions in the TPC-C benchmark, but only two of these are distributed transactions, given the H-Store partitioning: the `payment` transaction and the `new_order` transaction.

The `payment` transaction updates a customer's balance and the warehouse/district sales fields. Since this transaction never needs to abort, it can be implemented as an independent transaction where each repository updates the customer balance and sales fields independently.

The `new_order` transaction may need to abort if the transaction request contains an invalid item number. The Item table is read-only in the TPC-C workload, and can be replicated at each repository. Since each repository has local access to the Item table, each participant can independently determine whether to commit or abort a `new_order` transaction, and can thus execute it as an independent transaction.

If we extended the TPC-C benchmark such that the Item table was not read-only and occasionally needed to be updated, these updates could also be implemented using independent transactions.

2.2.2 MOTIVATION FOR AVOIDING LOCKING

Providing strong consistency for atomic transactions is a significant challenge in most distributed systems. Strong consistency, namely *serializability*, requires that the execution order of transactions at each server matches a single global serial order. This global order allows client applications to interact with the distributed system using the familiar abstraction of a single correct server, rather than a large collection of disparate nodes.

The challenge for providing serializability lies in respecting ordering dependencies between individual servers. For example, if a client executes an atomic transaction T_1 at servers S_1 and S_2, the system must guarantee that no client is able to observe T_1's effects at S_1 then execute a transaction at S_2 that modifies data used by T_1, before S_2 has executed T_1. An anomalous sequence of operations like this could violate the global serial ordering, and hence violate the abstraction of a single correct server.

Lock-based Concurrency Control

Serializable atomic distributed transactions are traditionally provided by the use of two-phase commit [28, 36], in conjunction with strict two-phase locking [25]. We illustrate this process in Figure 2.1, for a one-round transaction model. When a server receives a transaction it acquires any locks that may be required by the transaction, and records this information using a stable log write so that it will not be lost in case of a failure. The server then decides whether to commit or abort the transaction, and sends its vote to a coordinator. The coordinator collects votes from each participant, records this information using a stable log write, then sends out a commit

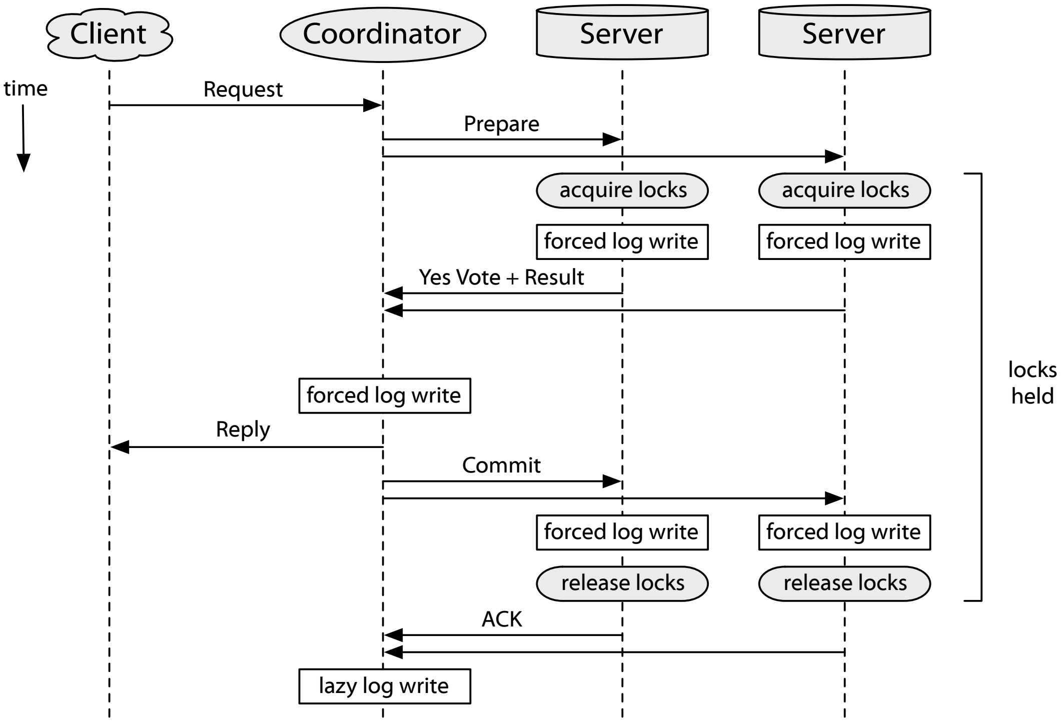

Figure 2.1: Traditional two-phase commit protocol, as applied to a one-round transaction model. Locks must be acquired for the duration of the transaction, and multiple forced log writes must be written on both the participants and the coordinator.

decision. Each server is only able to commit the transaction and release its locks once it receives this final commit decision. Of particular note here is the extended duration that locks must be held, which is exacerbated by the multiple forced log writes in the protocol.

Locking Costs

The use of two-phase commit with strict two-phase locking provides serializability [25], but has two significant downsides related to the cost of locking. The first downside is that locking is computationally expensive. Each locking step requires computing the lockset for the transaction, acquiring these locks, and recording an undo log in stable storage in case an abort decision is received. This locking overhead can comprise a significant fraction of CPU time, particularly for transactions that involve relatively low execution

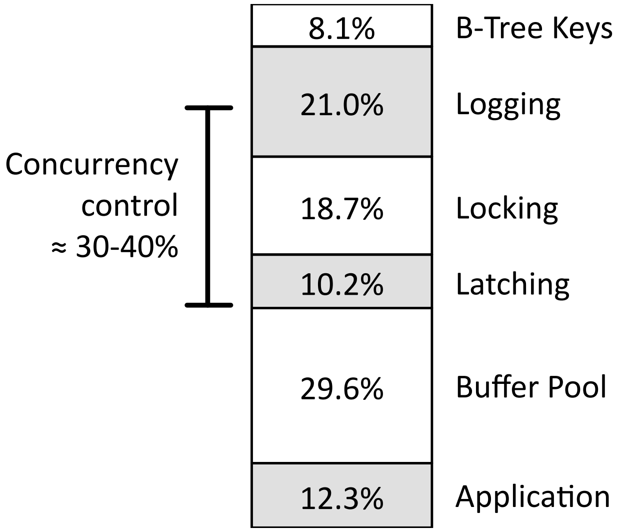

Figure 2.2: CPU cycle breakdown while executing TPC-C `new_order` transactions [8] on the Shore DBMS [6]. This figure appeared in *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases* [31] and is based on an earlier study [30].

cost, as is common in typical online transaction processing workloads [50]. Figure 2.2 shows the breakdown in execution time for transactions in the Shore database management system (DBMS) [6]. This figure shows a significant fraction of time devoted solely to locking, latching and logging. Studies of typical transaction processing workloads have estimated locking overhead to be 30–40% of total CPU load [30, 31], in addition to the logging required for durability.

The second downside is that locking can lead to *lock conflicts*, which limit concurrency and result in transactions being blocked or aborted. Locks must be held for the duration of a transaction, which can be a significant period of time for distributed transactions; as seen in Figure 2.1, the lock duration includes the time taken to execute the forced stable log writes, as well as the time taken to communicate with the coordinator. If a workload involves frequent access to commonly-used data, lock conflicts can significantly reduce system throughput.

These two downsides typically also apply to transactions that execute solely on a single server, since locking is required for all transactions if there are any concurrent distributed transactions.

3 ARCHITECTURE AND ASSUMPTIONS

This chapter describes Granola's architecture and system model, and defines the terminology used throughout this thesis. We also describe the Granola application interface and discuss the assumptions made in our design.

3.1 ARCHITECTURE

Granola contains two types of nodes: *clients* and *repositories*. Repositories are the server machines that store data and execute transactions, while clients interact with the repositories to issue transaction requests. Repositories communicate among themselves to coordinate transactions, whereas clients typically interact only with the repositories. This topology is illustrated in Figure 3.1.

Granola is a *general* platform for transaction coordination, and may be used to support a large variety of client and server applications. Applications link against the Granola library, which provides functionality for correctly ordering transactions and delivering them reliably to each server application.

Applications are layered atop the Granola client and repository code, as shown in Figure 3.2. The client application provides the desired application-specific interface to user code and interacts with the Granola *client proxy*

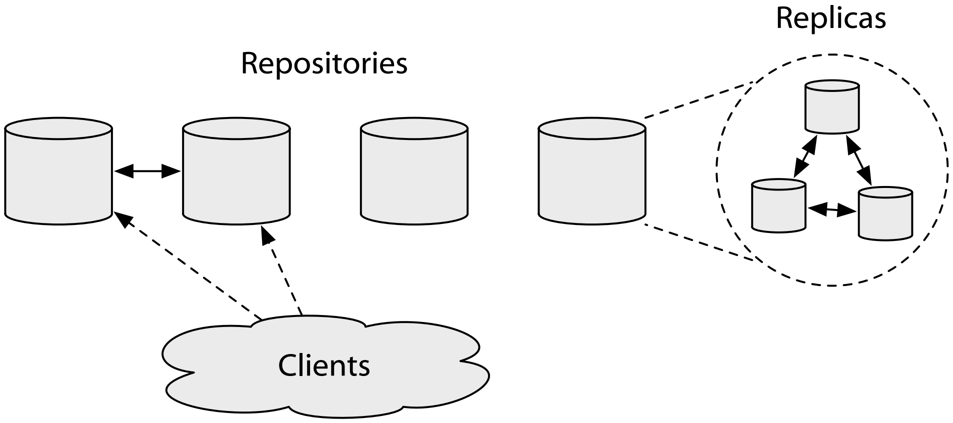

Figure 3.1: Granola system topology. This figure shows the clients and repositories, as well as the individual replicas that make up each repository.

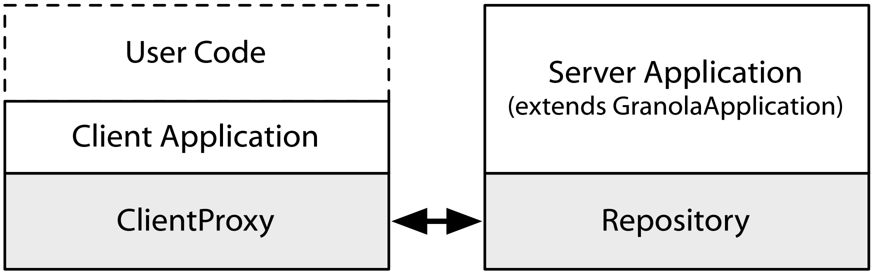

Figure 3.2: Logical structure of Granola client and server modules.

to issue requests. The client application is responsible for interpreting user operations, determining which repositories are involved in a given transaction, and choosing which transaction class to use.

The server application implements the GranolaApplication interface in order to execute transactions on behalf of each repository. The server application runs in isolation at each repository, and does not need to communicate with other repositories, since this functionality is provided by the Granola repository code. The client and server interfaces are described in more detail in Section 3.2.

Each repository is actually comprised of a number of replica nodes that are used to provide reliability. Replication is handled internally by the Granola protocol, on behalf of the application. This allows each server application to be soft-state, and eliminates the overhead for managing durability on the application level. Granola presents the abstraction of individual reliable repository nodes to the client application, and the client application does not need to be aware of the replicated repository structure. We discuss replication in more detail in Section 5.1.

3.2 SYSTEM INTERFACES

Granola is designed to support arbitrary storage applications built atop the platform, and communicates with applications using a well-defined interface. Granola does not interpret the contents of operations, and treats them as arbitrary byte strings. We discuss the client and server interfaces below.

3.2.1 CLIENT API

Client applications make function calls to the `ClientProxy` library to issue transaction requests. Figure 3.3 shows the interface provided by this library.

The client application must determine which transaction class to use for the request, and call the corresponding invocation function; the decision of which transaction class to use for a given transaction is thus the responsibility of the application developer. The client application is also responsible for specifying which repositories will receive the transaction. Each repository is identified by a unique repository identifier, known as an *RID*. The mapping from application-level names to RIDs is managed by a name service that stores this mapping on behalf of clients, as described in Section 7.1.3.

Transaction participants provide a result response if the transaction commits; the client proxy will write this response into the buffer provided by the client application. Granola also allows a result response to be returned for aborted transactions, when using the coordinated transaction interface, to allow for application specific abort information to be communicated to the client application, e.g., an abort code or status message. Granola only returns an aborted result from one repository to the client application, to avoid waiting to hear from all repositories.

```
/*
 * Independent Interface
 */

// issue trans to given repository with read-only flag
// writes result into provided buffer
void invokeSingle(int rid, ByteBuffer op, boolean ro,
                  ByteBuffer result);

// issue trans to set of repositories
void invokeIndep(List<Integer> rids, List<ByteBuffer> ops,
                 boolean ro, List<ByteBuffer> results);


/*
 * Coordinated Interface
 */

// issue trans to set of repositories
// returns application commit/abort status.
// only one result returned if abort received,
// the rest are set to length 0
boolean invokeCoord(List<Integer> rids, List<ByteBuffer> ops,
                    List<ByteBuffer> results);
```

Figure 3.3: Client API. Clients invoke transactions by calling this API. We provide additional specialized interfaces for distributed transactions where all participants receive the same request.

The client may specify whether a single-repository transaction or an independent transaction is read-only. Coordinated transactions are never read-only, since a read-only coordinated transaction can always be implemented using the independent transaction class.

The Granola `ClientProxy` interface provides blocking function calls for transaction invocation. The interface can easily be extended, however, to support non-blocking invocation. Our current implementation supports multiple concurrent invocation requests from individual client application threads. The final serial order of concurrent requests is determined by Granola.

3.2.2 SERVER API

Server applications extend the `GranolaApplication` interface, to process operations on behalf of the Granola repository code. This interface is shown in Figure 3.4.

We refer to function calls to the server application as *upcalls*, to signify that the code is being executed within the server application and not the Granola repository code. The repository adopts a single-threaded execution model, where one upcall is issued at a time to the server application. This allows the server application to avoid concurrency control overhead, and achieve significantly higher transaction throughput [50]. Our execution model achieves best performance when transaction execution does not involve significant stalls within the application, as discussed in Section 3.3.

Single-repository transactions and independent transactions are executed atomically using a single `run` upcall. The repository uses this upcall to pass the operation to the server application as a byte buffer, and receives a byte buffer result.

Upcalls for coordinated transaction follow a two-phase process: first the repository issues a `prepare` upcall to prepare the transaction, followed by a `commit` or `abort` upcall once the final commit decision has been determined. Each transaction is identified by a unique *transaction identifier (TID)*, which is provided by the client proxy in each request. The `prepare` upcall is used

```
/*
 * Independent Interface
 */

// executes transaction to completion
// returns false if blocked by coord trans, true otherwise
boolean run(ByteBuffer op, ByteBuffer result);


/*
 * Coordinated Interface
 */

// runs to commit point and acquires locks
// returns COMMIT if voting commit, CONFLICT if aborting
// due to a lock conflict, and ABORT if aborting due to
// application logic
// result is non-empty only if returning ABORT
Status prepare(ByteBuffer op, long tid, ByteBuffer result);

// commits trans with given TID, releases locks
void commit(long tid, ByteBuffer result);

// aborts trans with given TID, releases locks
void abort(long tid);


/*
 * Recovery Interface
 */

// acquires any locks that could be required if preparing
// the trans at any point in the serial order
// acquires additional handle on locks if there's a conflict
// returns true if no conflict, but acquires locks regardess
boolean forcePrepare(ByteBuffer request, long tid);
```

Figure 3.4: Server Interface. Applications extend this interface to handle upcalls from Granola.

to determine the *vote* for the transaction at the repository. Each participant decides whether to vote commit or abort, as in traditional two-phase commit. If the application votes to commit the transaction, the repository will issue a subsequent `commit` upcall if all other participants vote commit, or an `abort` upcall otherwise.

The contract with the server application is that if it votes to commit a transaction, then it must be able to commit or abort that transaction when the corresponding upcall is subsequently received, while providing isolation from any concurrent transactions. Typically this guarantee will be provided by using strict two-phase locking [25] within the application: when a `prepare` upcall is received, the application executes the transaction to completion, and acquires locks on any pages touched by the transaction. The application also records an undo record, to roll back execution if the transaction is aborted.

The application can abort a transaction in one of two different ways. If it encounters a lock conflict during the prepare phase, it rolls back execution and returns a `CONFLICT` response. The client proxy will retry this transaction after a random backoff period, to wait for any conflicting locks to be released. If the application decides to abort the transaction due to an application-specific predicate, however, such as there being insufficient funds in a bank account, then it rolls back execution and returns an `ABORT` response. If the client proxy receives an `ABORT` response it will return this directly to the client application, and will not retry the transaction.

The recovery interface is used to make progress when a transaction participant has failed. This interface is described in more detail in Section 5.3. The application is not required to maintain any stable state, since this functionality is provided by replication within the Granola protocol.

3.3 ASSUMPTIONS

Granola assumes a mostly-partitionable in-memory workload, as is common in most large-scale transaction processing applications [50]. A mostly-partitionable workload is required to allow the system to scale in the number of repositories; Granola is designed to provide high performance for distributed transactions, but no scalability benefit can be realized if every transaction is executed on a large fraction of the participants.

An in-memory workload is desirable as it avoids costly disk stalls and reduces the potential execution time for a transaction. We assume that transactions have relatively short duration, since our execution model issues upcalls one at a time; the assumption of short transaction duration is also typical in most large-scale online transaction processing workloads. Our model of serial transactions execution matches the execution model advocated in the H-Store project, which saves considerable overhead over the cost of application-level latching and concurrency control for short-lived transactions [50]. Note while each repository only executes one transaction at a time, it can coordinate the ordering of many transactions in parallel. Multiple repositories may be colocated on a single machine to take advantage of multiple compute cores, and Granola can also be extended to support multithreaded execution, as outlined in Section 9.1.1.

Granola tolerates crash failures. Our replication protocol depends on replicas having loosely synchronized clock *rates* [38]. We depend on repositories having loosely synchronized *clock values* for performance, but not for correctness.

Since we are providing strong consistency, there may be cases where we have to stall because of failure [16]. Our use of replication minimizes the likelihood that a repository will be unavailable. In the case of a correlated failure that leads to unavailability of an entire repository, some transactions may stall but others will continue to run. Section 5.3 describes the system behavior during periods of long-term failures or network partitions, and specifies protocols to allow progress during these periods.

4 GRANOLA PROTOCOL

This chapter describes the core Granola protocol. We first discuss the timestamps used to provide serializability in the absence of locking. We then discuss the two concurrency control modes that determine whether or not locking is used, and our protocols for the three transaction classes.

In the following sections we note where a *stable log write* is required. This step is analogous to a forced disk write in a traditional database, however we instead use replication to ensure that the log write is stable. Stable log writes involve the primary replica executing state machine replication, as described in Section 5.2.1. Replication is only required where explicitly denoted in the protocol as a stable log write. Communication between repositories occurs solely between the primary replicas at each repository; if a request is received by a non-primary replica, it forwards the request to the primary.

This chapter assumes there are no failures. We discuss failure and recovery in Chapter 5 and the protocols for retries and duplicate detection are discussed in Section 5.4. A full listing of the message formats used in the Granola protocol is provided in Appendix A.

4.1 TIMESTAMP OVERVIEW

Granola uses *timestamps* to define the commit order of transactions. Unlike systems based on traditional two-phase commit [1, 10, 12, 28, 36, 42], the use of timestamps allows Granola to order single-repository transactions and independent distributed transactions with no locking, conflicts or deadlock. Each transaction is assigned a timestamp that defines its position in the global serial order. We use a distributed timestamp voting mechanism to ensure that a distributed transaction is given the same timestamp at all participants, as described in Section 4.3.4.

A transaction is ordered before any other transaction with a higher timestamp. Two transactions may end up being assigned the same timestamp, since the final timestamps depend on votes from other participants. In this case the TID is used to break the tie when ordering transactions, i.e., the transaction with the lowest TID is ordered first. Since timestamps define a *total ordering* of transactions, and we execute transactions in this timestamp order, our protocol guarantees serializability.

Repositories choose timestamps by reading their local system clock. Transaction participants exchange timestamps before committing a given transaction, to ensure that they all assign it the same timestamp. Granola timestamps are similar to Lamport Clocks [34]: Each transaction result sent to the client contains the timestamp for that transaction, and each request from the client contains the latest timestamp observed by the client. Repositories postpone execution of a client request until they are up-to-date with respect to highest transaction observed by the client, ensuring that they are in sync.

Wall-clock time values are used as the basis for timestamps to ensure that repositories choose similar timestamps for a given distributed transaction, before voting to decide on the final timestamp. This is used purely as a performance optimization, since the correctness of the protocol is already guaranteed by clients propagating timestamps in subsequent requests. We explain the use of timestamps in the following sections.

4.2 CLIENT PROTOCOL

This section discusses the protocol used by the Granola client proxy, which is code provided by the Granola library. The client proxy receives transaction requests from the client application and issues them to the repositories, as shown in Figure 3.2. The client proxy exports the interface provided in Figure 3.3. We will refer to the client proxy in the following discussion as the "client."

4.2.1 CLIENT STATE

Each client maintains the following state:

`cid`: The globally-unique ID assigned to the client.

`seqno`: The sequence number for the current transaction. Each transaction has a unique sequence number at a given client.

`highTS`: The highest timestamp the client has observed in any previous reply from a repository.

The client identifies each transaction by a globally-unique *TID* (transaction identifier). The TID is generated by concatenating the globally-unique `cid` with the `seqno` for the transaction.

Persistence of this state across failures is discussed in Section 5.2.2.

4.2.2 TRANSACTION INVOCATION

The client application issues transaction invocation requests to the client proxy using the Granola client API, as discussed in Section 3.2.1. The interface for issuing single-repository transactions and independent distributed transactions is shown in Figure 4.1. The interface for coordinated distributed transactions is shown in Figure 4.2.

Each invocation upcall to the client proxy specifies the following parameters:

```
// issue trans to given repository with read-only flag
// writes result into provided buffer
void invokeSingle(int rid, ByteBuffer op, boolean ro,
                  ByteBuffer result);

// issue trans to set of repositories
void invokeIndep(List<Integer> rids, List<ByteBuffer> ops,
                 boolean ro, List<ByteBuffer> results);
```

Figure 4.1: Independent Transaction Client API. Clients invoke single-repository transactions and independent transactions by calling this API.

```
// issue trans to set of repositories
// returns application commit/abort status
//
// only one result returned if abort received,
// the rest are set to length 0
boolean invokeCoord(List<Integer> rids, List<ByteBuffer> ops,
                    List<ByteBuffer> results);
```

Figure 4.2: Coordinated Transaction Client API. Clients invoke coordinated transactions by calling this API.

rids: The repository IDs (RIDs) for the set of repositories involved in the transaction. Only one ID is provided for single-repository transactions.

ops: The transaction operation to execute at each repository in the rids set, expressed as an uninterpreted byte string.

ro: A flag specifying whether the request is read-only. This flag is implicitly set to false for coordinated transactions, since read-only transactions can always be implemented using independent transactions.

results: A set of buffers that the transaction results will be written into, one for each repository in rids.

The client first increments its seqno, and then sends a ⟨REQUEST, tid, highTS, rids, ops, ro, indep⟩ message to the set of repositories specified in rids. The indep flag specifies whether the transaction is independent, which is true for invokeSingle and invokeIndep upcalls and false for invokeCoord upcalls.

Each repository responds with a ⟨REPLY, tid, rid, ts, status, result⟩ message, including the TID for the transaction, the repository's RID, the timestamp (ts) assigned to the transaction, and the result from executing the op. A status value of COMMIT means that the transaction committed, which we refer to as a *committed response*. If status is CONFLICT or ABORT then the transaction was aborted, and we refer to it as *aborted response*. We discuss each case in the following sections.

Committed Response

If a client receives a committed REPLY message, it waits to receive a REPLY from all the repositories in rids. These will all be committed responses, since all repositories make a consistent decision about whether to commit a given transaction.

Once the client receives all responses, it first sets its highTS value to be the ts in the REPLY messages. It then aggregates the result from each repository and returns these to the client application in the provided results buffer.

If the transaction was invoked using a `invokeCoord` upcall, the return value will be set to true, otherwise the return value is void. The client application will always observe a single-repository transaction or independent transaction to commit. If the server application wishes to communicate an application-level abort for these transactions, it can encapsulate this in the transaction `result`.

Aborted Response

If the transaction aborted, the `status` will be set to either `ABORT` or `CONFLICT`. These responses correspond to the two reasons a transaction may abort: either the server application ran the `op` and decided to abort it based on application logic; or the transaction aborted due to a lock conflict or other failure. We describe these two cases as follows:

Logical Abort An `ABORT` status signifies that at least one of the participants voted abort for the transaction, based on application-level predicates. This is considered the end of the transaction, and the client updates `highTS` as discussed for committed responses.

Logical aborts are returned only for coordinated transactions, since these are the only transactions that allow voting on the commit decision. The client writes the `result` from the first abort response to the buffer provided by the client application, and returns false in response to the `invokeCoord` upcall. The `result` response is included to allow the server application to communicate any additional information about the abort to the client application.

Conflict Abort A `CONFLICT` status signifies that there was a lock conflict or failure when attempting to execute the transaction, and that the transaction should be retried. Unlike logical aborts, the client does not update `highTS`, since the transaction has not yet completed. The client instead increments its `seqno` before retrying, to signify that it is a new transaction attempt. The client first waits a random binary exponential backoff period, then reissues the transaction with the new TID.

Mode	Supported Transactions
Timestamp	Single-Repository, Independent
Locking	Single-Repository, Independent, Coordinated

Table 4.3: Summary of concurrency control modes. A repository is in only one mode at a time.

The circumstances whereby transactions may abort are discussed in the following sections.

4.3 REPOSITORY PROTOCOL

This section describes the protocols for processing transactions at each repository. We first describe Granola's two concurrency control modes, which dictate which protocol is used for transaction coordination.

4.3.1 CONCURRENCY CONTROL MODES

Each repository in Granola operates in one of two distinct concurrency control *modes*. When the repository is currently handling only single-repository transactions or independent transactions, it operates in *timestamp mode*. This is the primary concurrency control mode in Granola, and the mode during which no locking is required. If a repository is currently handling a coordinated transaction, it dynamically switches to *locking mode*. Locking is required for all transactions at the repository when it is in locking mode; this avoids conflicts with active coordinated transactions. These modes are summarized in Table 4.3.

Each repository is in only one mode at a time. The decision to be in a particular mode is made locally at a given repository, and is not a global property. A repository in timestamp mode can safely be a participant in an independent transaction with other participants that are in timestamp mode. The application must keep track of whether it is in timestamp mode or locking mode, based on whether there are any currently-outstanding transactions that were issued a `prepare` upcall but not yet been issued a `commit` or `abort`.

Serializability is provided using timestamps when in timestamp mode, whereas locking is used in locking mode to provide compatibility with coordinated transactions. Sections 4.3.3 and 4.3.4 describe the protocols for single-repository and distributed transactions as they work in timestamp mode. Section 4.3.5 describes how the system runs in locking mode and how it transitions between modes.

4.3.2 REPOSITORY STATE

Each repository maintains the following state:

`rid`: The globally-unique ID assigned to the repository.

`mode`: The concurrency control mode the repository is currently in, either `TIMESTAMP` or `LOCKING`.

`lastExecTS`: The timestamp of the most recently executed transaction at the repository.

In addition to the state listed above, the repository must also maintain various internal buffers to keep track of individual transaction state. The management of this state is described in the following sections.

4.3.3 SINGLE-REPOSITORY TRANSACTIONS

The basic protocol for single-repository transactions is straightforward, and has much in common with how existing single-node storage systems work. The key difference in Granola is the use of timestamps to ensure serializability in the presence of independent transactions.

The server application at each repository executes transactions in response to `run` upcalls from the repository code, as described in Section 3.2.2. The application interface required to support execution of single-repository transactions is shown in Figure 4.4.

```
// executes transaction to completion
// returns false if blocked by coord trans, true otherwise
boolean run(ByteBuffer op, ByteBuffer result);
```

Figure 4.4: Independent Transaction Server Interface. Applications extend this interface to handle upcalls from Granola for single-repository transactions or independent distributed transactions.

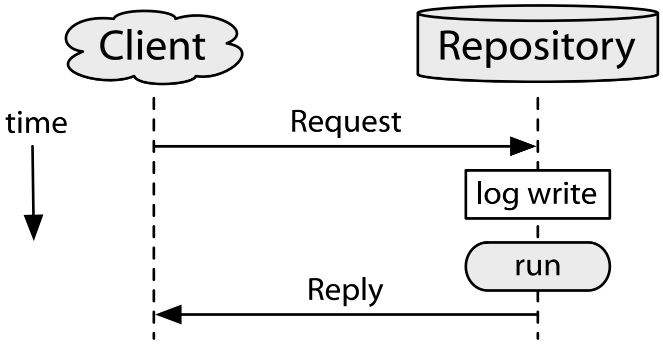

Figure 4.5: Protocol timeline for single-repository transactions.

Read-Write Transactions

We first describe the protocol for read-write transactions, which can arbitrarily read and modify the service state. The protocol for processing a ⟨Request, `tid`, `highTS`, `rids`, `ops`, `ro`, `indep`⟩ message, where the size of `rids` is 1 and `ro` is false, is described as follows. The protocol timeline for single-repository transactions is shown in Figure 4.5.

1. The repository first selects a timestamp `ts` for the transaction. `ts` must be greater than `lastExecTS` at the repository and the `highTS` provided in the Request.

2. The repository performs a stable log write to record both the Request and `ts`, so that this information will persist across failures.

3. The transaction is now ready to be executed. Transactions are executed in timestamp order, and are queued after any transactions with lower timestamps. The repository first sets its `lastExecTS` value to `ts`. The transaction is then executed by issuing a `run(op, result)` upcall to

the server application, where `result` is an empty byte buffer used to store the result.

4. Finally a ⟨Reply, `tid`, `rid`, `ts`, `status`, `result`⟩ message is sent to the client, where `status` is `COMMIT`.

Additional transactions can be processed while awaiting completion of a stable log write; these requests will be executed in timestamp order.

Read-Only Transactions

The protocol for read-only transactions is the same as for read-write transactions, except that a stable log write is not required in Step 2 of the protocol. Since read-only transactions do not modify the service state, they can be retried in the case of failure.

We present optimizations to run read-only transactions at non-primary replicas in Section 7.2.

4.3.4 INDEPENDENT TRANSACTIONS

Independent distributed transactions are ordered with respect to all other transactions, without any locking or conflicts. This is achieved by executing each independent transaction at a single timestamp at all transaction participants.

We determine the timestamp for independent transactions by using a distributed timestamp voting mechanism. Each participant nominates a proposed timestamp for the transaction, the participants exchange these nominations in Vote messages, and the transaction is executed at the highest timestamp from among these votes. This protocol does not require a dedicated coordinator, or the involvement of repositories that are not participants in the transaction.

The application interface required to support execution of independent transactions is the same as for single-repository transactions, as shown in Figure 4.4.

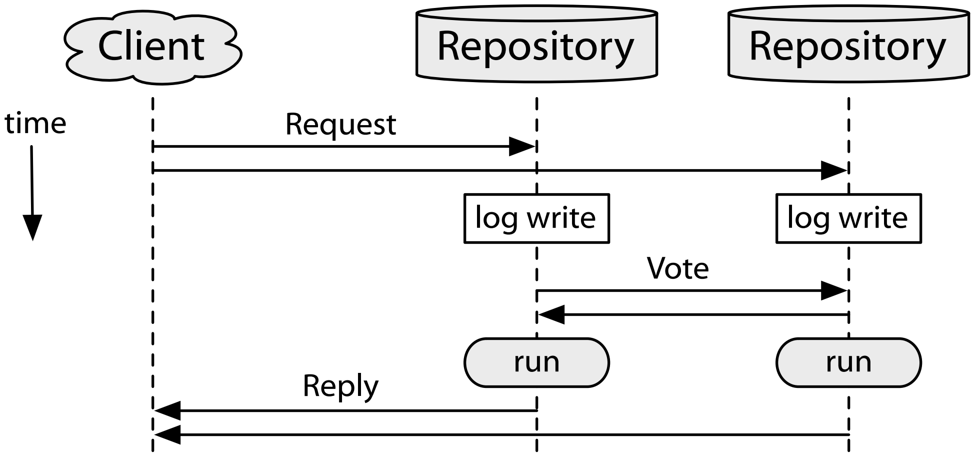

Figure 4.6: Protocol timeline for independent transactions.

This section discusses the protocol for handling independent transactions when in timestamp mode; the locking mode protocol is discussed in the subsequent section.

Read-Write Transactions

The protocol for processing a read-write ⟨Request, `tid`, `highTS`, `rids`, `ops`, `ro`, `indep`⟩ message, where `indep` is true and `ro` is false, is described as follows. The protocol timeline for independent transactions is shown in Figure 4.6.

1. The repository selects a *proposed* timestamp (`proposedTS`) for the transaction. `proposedTS` must be greater than the `lastExecTS` value at the repository and the `highTS` value provided in the Request.
2. The repository performs a stable log write to record both the Request and `proposedTS`, so that this information will persist across failures.
3. The repository sends a ⟨Vote, `tid`, `rid`, `proposedTS`, `status`, `retry`⟩ message to the other participants, where `status` is `COMMIT` and `retry` is false. The `retry` flag is only used during failure recovery, as discussed in Section 5.4. The repository can process other transactions after the vote has been sent.

4. The repository waits for Vote messages from the other participants listed in `rids`. If a Vote is received where `status` is `CONFLICT`, the repository ceases processing the transaction and immediately responds to the client with a ⟨Reply, `tid`, `rid`, `proposedTS`, `status`, `result`⟩ message, where `status` is `CONFLICT` and `result` is empty. `CONFLICT` votes can only be received from a participant that is in locking mode. It is not possible to receive a vote with an `ABORT` `status` for an independent transaction.

5. Once commit Vote messages have been received from all other participants, the transaction is assigned the highest timestamp (`finalTS`) from all `proposedTS`s in the votes. The same `finalTS` value will be chosen at all participants.

6. The transaction is now ready to be executed. Transactions are executed in timestamp order, and are queued after any transactions with lower timestamps. When it is time to execute the transaction, the repository first sets its `lastExecTS` value to `finalTS`. The transaction is then executed by issuing a `run(op, result)` upcall to the server application, where `result` is an empty byte buffer used to store the result.

7. Finally a ⟨Reply, `tid`, `rid`, `finalTS`, `status`, `result`⟩ message is sent to the client, where `status` is `COMMIT`.

As mentioned, the repository can process other transactions while waiting for votes. In all cases we guarantee serializability by executing in *timestamp* order. A transaction will not be executed until after any concurrent transaction with a lower timestamp at the repository. It is thus possible for the execution of a transaction to be delayed if a transaction with a lower timestamp has not yet received a full set of votes. The repository can continue to vote on other transactions and queue up transaction execution during this period. Longer-term delays can occur if a transaction participant has failed and the repository does not receive a full set of votes; recovery from a failed participant is discussed in Section 5.3.

Read-Only Transactions

The protocol for read-only independent transactions is the same as for read-write independent transactions, except that a stable log write is not required in Step 2 of the protocol.

If the repository recovers from failure, it is possible that the new primary will vote again for the read-only independent transaction with a different timestamp. This can result in the client receiving responses with mismatched timestamps. The client proxy will discard any such transaction, and retry it with a new TID. This protocol does not violate serializability, since even though different repositories may process a read-only transaction at different timestamps, the client proxy will never return a conflicting response to the client application; the repository states will not diverge since the transaction is read-only.

4.3.5 LOCKING MODE

We now discuss the protocols used at a repository when in locking mode. We first describe the protocol for coordinated transactions, and then discuss how the handling of single-repository transactions and independent transactions changes when in locking mode.

Coordinated Transactions

Coordinated transactions require participants to agree on whether to commit or abort a transaction. Each repository first determines whether to vote commit or abort for the transaction, and the transaction commits only if all participants send a commit vote.

Locking Coordinated transactions require locking to support concurrency. Otherwise a transaction might invalidate the state used to determine a commit vote for another transaction. A simple example is a bank transfer that can only proceed if an account has sufficient funds: locking is required to ensure that another transaction does not produce insufficient funds after the commit vote for the transfer has been sent.

```
// runs to commit point and acquires locks
// returns COMMIT if voting commit, CONFLICT if aborting
// due to a lock conflict, and ABORT if aborting due to
// application logic
// result is non-empty only if returning ABORT
Status prepare(ByteBuffer op, long tid, ByteBuffer result);

// commits trans with given TID, releases locks
void commit(long tid, ByteBuffer result);

// aborts trans with given TID, releases locks
void abort(long tid);
```

Figure 4.7: Coordinated Transaction Server Interface. Applications extend this interface to handle upcalls from Granola for coordinated distributed transactions.

Granola presents a layered architecture to applications and does not directly interpret transaction operations, as described in Section 3.1. Locking is thus the responsibility of the server application that sits atop the Granola repository code. The application interface required to support locking and execution of coordinated transactions is shown in Figure 4.7.

The repository passes an operation to the application using a `prepare` upcall, and the application returns its vote for the transaction. If the application returns a commit vote, it must guarantee that it is able to commit or abort the transaction at any point in the future. This can be achieved by using strict two-phase locking: pre-executing the transaction, locking any pages touched during execution, and recording an undo log in case the transaction aborts. The application is free, however, to use any equivalent concurrency control mechanism.

The final commit or abort decision is communicated to the application using a `commit` or `abort` upcall. Transactions are identified by passing the TID to the application.

Protocol The protocol for processing a read-write ⟨Request, `tid`, `highTS`, `rids`, `ops`, `ro`, `indep`⟩ message, where `indep` and `ro` are false is as follows.

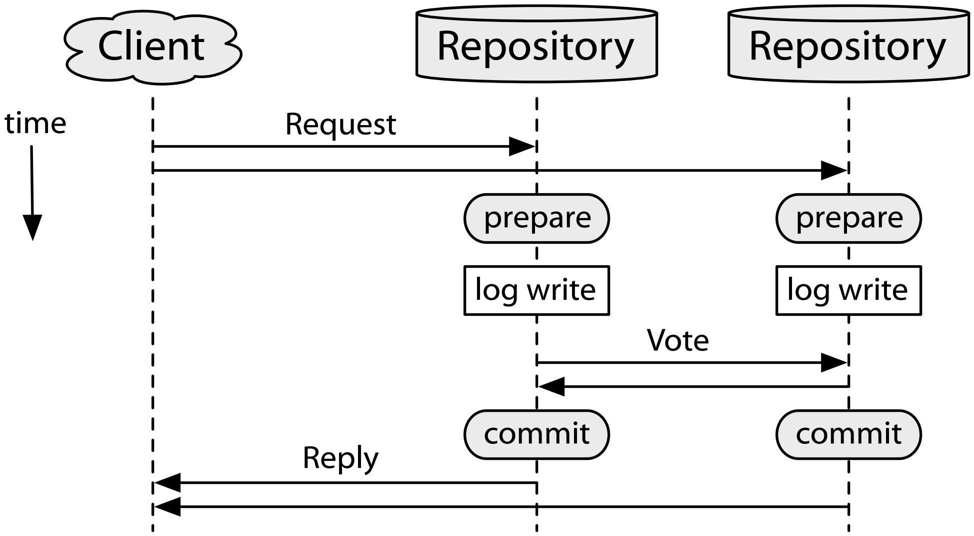

Figure 4.8: Protocol timeline for coordinated transactions.

Read-only transactions are not considered, since any read-only distributed transaction can be implemented as an independent transaction. The protocol timeline for coordination transactions is shown in Figure 4.8.

1. Coordinated transactions first undergo a prepare phase. The repository issues a `prepare(op, tid, result)` upcall to the application to determine the transaction vote, where `result` is an empty byte buffer. The application acquires any locks required by the transaction, and returns its vote, releasing its locks if it decides to abort. If the application is unable to acquire all locks due to a concurrently executing transaction, it returns a `CONFLICT` response. If the application is voting abort due to application logic, it returns an `ABORT` response, and can write any further abort information into the `result` buffer. Otherwise the return value is `COMMIT` and `result` is left empty.

2. The repository selects a *proposed* timestamp (`proposedTS`) for the transaction. `proposedTS` must be greater than the `lastExecTS` value at the repository and the `highTS` value provided in the Request.

3. The repository performs a stable log write to record the Request, `proposedTS`, and `status`, so that this information will persist across

failures.

4. The repository sends a ⟨Vote, tid, rid, proposedTS, status, retry⟩ message to the other participants, where status contains the response from the application. The retry flag is set to false as it is only used during retransmissions, as discussed in Section 5.4. If the repository is voting abort (i.e., status is CONFLICT or ABORT), the repository ceases processing the transaction, and responds to the client with a ⟨Reply, tid, rid, proposedTS, status, result⟩ message.

5. The repository waits for votes from the other participants. If it receives an abort vote (i.e., status is CONFLICT or ABORT), the repository responds immediately to the client with a ⟨Reply, tid, rid, proposedTS, status, result⟩ message. It then issues an abort(tid) upcall to the application, to revert any modifications and release locks acquired by the transaction. The repository then ceases processing the transaction.

6. Once commit Vote messages have been received from all other participants, the transaction is assigned the highest timestamp (finalTS) from all proposedTSs in the votes. The same finalTS value will be chosen at all participants. The transaction is then committed by issuing a commit(tid, result) upcall to the server application, where result is an empty byte buffer used to store the result. The application completes execution, releases any locks, and writes the return value into the result buffer.

7. Finally a ⟨Reply, tid, rid, finalTS, status, result⟩ message is sent to the client, where status is COMMIT.

Repositories are allowed to execute transactions out of timestamp order when in locking mode, since we use strict two-phase locking in this mode, which is sufficient to guarantee serializability in the absence of timestamps [25]. We discuss our consistency properties in more detail in Section 4.4.

Impact on Other Transactions

Granola uses different protocols for independent transactions and single-repository transactions during locking mode, to ensure that these transactions do not conflict with locks acquired for concurrent coordinated transactions.

Independent Transactions Independent transactions are processed using the coordinated transaction protocol when in locking mode, and issued to the application using `prepare`, `commit` and `abort` upcalls. A `prepare` upcall for an independent transaction will never receive an `ABORT` response. The client will retry the transaction with a new TID if it receives a `CONFLICT` response.

Single-Repository Transactions We avoid holding locks for single-repository transactions by attempting to execute them as soon as they have been assigned a timestamp, but before the transaction is logged. The repository attempts to execute the transaction by issuing a `run` upcall to the application; in this case the application must check whether the transaction conflicts with any currently-held locks. If there is a lock conflict, the application returns false in response to the upcall.

The application does not *acquire* locks for `run` upcalls; it only checks whether the transaction attempts to read or modify state that is currently locked. This is typically achieved by recording an undo record for the transaction, checking locks during execution and rolling back any changes if there is a conflict. The application tracks whether or not any locks are held at a given point in time, and only performs checks if there are active locks. No checks are performed when the repository is in timestamp mode.

If a `run` upcall returns false, the repository responds to the client with a `CONFLICT` response, and discards the transaction. No log write is required for the aborted transaction, and the client proxy will retry the transaction with a new TID.

If the `run` upcall returns true, the repository issues a stable log write to record the timestamp and commit status for the transaction. The repository does not respond to the client until after the stable log write is compete, but immediately updates its record of the `lastExecTS` timestamp so that future transactions are assigned higher timestamps than the single-repository transaction.

The repository must execute the transaction *before* logging, and cannot instead execute single-repository transactions after they have been logged as in the timestamp mode protocol. Since each repository is a replicated state machine, it is essential that the backup replicas replay the same sequence of operations as the primary replica. If the primary was to log the transaction before attempting to execute it, it would not yet know whether the transaction will commit or abort once the logging is complete. This commit or abort decision is non-deterministic, since it depends on the order in which concurrent distributed transactions commit, which is independent of timestamp order when in locking mode. By executing before logging, the primary is able to directly inform the backups of whether the transaction commits or aborts.

The protocol for a processing committed transaction guarantees that the single-repository transaction will be given a timestamp consistent with the global serial order. The single-repository transaction will have been given a timestamp higher than any previously-executed transaction, any subsequent transaction will be given a timestamp higher than the single-repository transaction, and any concurrent transaction is commutative with the single-repository transaction, since there are no lock conflicts.

The protocol for read-only transactions in locking mode is the only part of the protocol where the primary replica can execute a transaction before informing the backups. This has implications for the recovery protocol, which we discuss in Section 5.2.1.

4.3.6 TRANSITIONING BETWEEN MODES

So far we have discussed the protocols used when a repository is in timestamp mode and in locking mode, but have not described the protocol for switching *between* modes. Our protocols for switching attempt to do so as quickly as possible, to avoid stalling coordinated transactions when in timestamp mode, or using locking mode when not necessary. We perform a thorough evaluation of the cost of switching modes in Chapter 6.

Switching to Locking Mode

If there are independent transactions in progress when a coordinated transaction arrives, the application needs to acquire locks for these transactions before issuing a `prepare` upcall for the coordinated transaction.

The repository issues `prepare` upcalls, in timestamp order, for all independent transactions for which it has already sent a vote but has not yet executed. If all upcalls proceed successfully without lock conflicts, the repository transitions into locking mode and will complete execution of the transactions using the locking mode protocol.

If a `prepare` upcall is unsuccessful and conflicts on a lock, the repository must defer transitioning into locking mode until the upcall can be issued successfully for all concurrent transactions. This can result in a coordinated transaction being blocked waiting for independent transactions to complete. Recovery from an extended period of blocking, as in the case of a failure, is discussed in Section 5.3.

Switching to Timestamp Mode

If there are no current distributed transactions when the last coordinated transaction is complete, then the repository can immediately switch back into timestamp mode; the application only acquires locks for distributed transactions and hence will have no active locks. It may be the case, however, that there are still current active independent transactions for which locks were acquired using the locking mode protocol. This scenario is likely under our expected workloads of frequent independent transactions but infrequent coordinated transactions.

To enable rapid transition to timestamp mode, the repository issues an `abort` upcall for all independent transactions that underwent a prepare phase but have not yet been committed. This releases any locks for these transactions, and allows the repository to transition into timestamp mode. The transactions that were issued an `abort` upcall will be executed in timestamp order using `run` upcalls once their `finalTS` is known.

Our experimental analysis indicates that the best protocol to use for switching to timestamp mode is to immediately issue `abort` upcalls for any active independent transaction, despite the overhead of undoing and re-executing the transactions. We investigate other techniques for transitioning into timestamp mode in Appendix B.

4.4 CONSISTENCY

This section discusses the consistency properties of the Granola protocol, along with the mechanisms used to provide consistency.

4.4.1 LOCKING SUPPORT

Granola requires application support to maintain serializability in locking mode, by providing isolation between concurrent transactions. Typically this will be achieved by using strict two-phase locking in the application [25]. Locking is not required when in timestamp mode, since each transaction executes atomically with a single application upcall at each participant.

4.4.2 SERIALIZABILITY

Timestamps in Granola define a consistent global serial ordering across all transactions. This serial order is defined through the use of timestamp voting and the use of clients to propagate timestamps between repositories. When in timestamp mode, repositories always commit transactions in this global serial order. This protocol thus guarantees *global serializability*. The use of timestamps allows Granola to maintain serializability even without locking.

Our protocol for independent distributed transactions allows each repository to execute its part of the transaction without knowing what is happening at the other participants: it only knows that they will ultimately select the same timestamp. This means that it is possible for a client to observe the effects of a distributed transaction T at one repository before another participant has executed it. Since a subsequent request from the client will carry a `highTS` value at least as high as T, we can guarantee that this request will execute after T at any participant.

Strict two-phase locking is used to provide transaction isolation during locking mode. Strict two-phase locking is sufficient alone to provide serializability in locking mode [25], and repositories may thus execute transactions out of timestamp order when in this mode.

The local state at the repository in locking mode is state-equivalent to an execution where each transaction is executed serially in timestamp order. A given transaction T in locking mode will be assigned a timestamp higher than any transaction that executed before T was prepared, and the locks ensure that any concurrent transaction is commutative with T. Timestamps may thus not represent the *commit* order of transactions in locking mode, but nevertheless represent a valid *serial* order, since locking ensures that any transactions that execute out of timestamp order are guaranteed not to conflict.

The use of timestamps in locking mode ensures compatibility with participants that are operating in timestamp mode, since all participants will execute the transactions in an order equivalent to the global serial order. Timestamps also allow a repository to transition in and out of locking mode while maintaining serializability, since timestamps are used to define the serial order in both transaction modes.

Timestamp Example

We describe an example execution trace which shows the use of timestamps to ensure serializability, as illustrated in Figure 4.9. Each step in the sample execution is described as follows:

Step 1: Client 1 issues an independent transaction to Repositories 1 and 2, with TID 1 and `highTS` 0. Repository 1 chooses `proposedTS` 9 for the

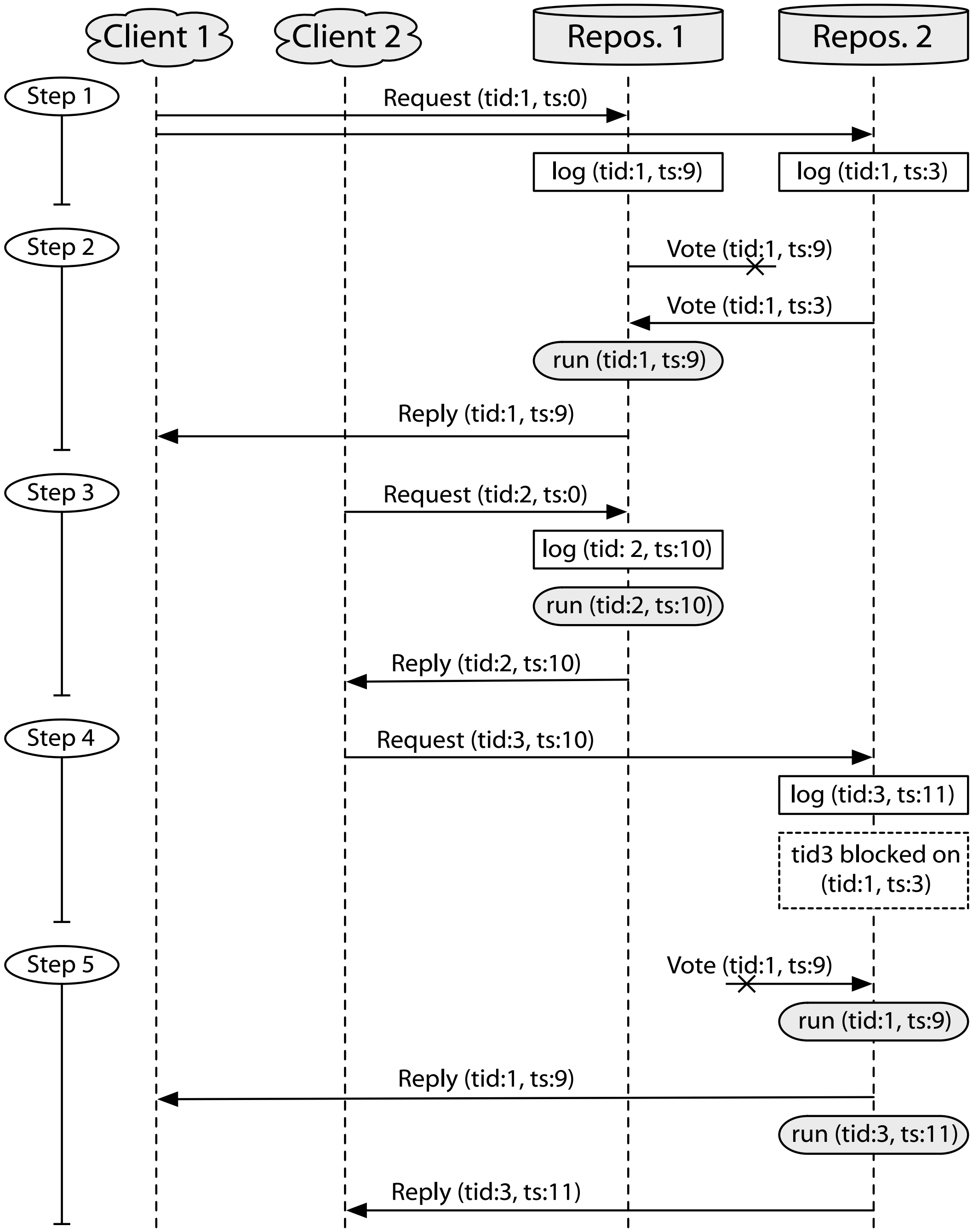

Figure 4.9: Example of a serializable execution, involving timestamp voting and the propagation of timestamps by clients.

transaction, while Repository 2 chooses `proposedTS` 3. Both repositories log the transaction record.

Step 2: Both repositories vote on TID 1, sending their proposed timestamp. Repository 1 receives Repository 2's vote, executes the transaction at `finalTS` 9, sets `lastExecTS` to 9, and responds to the client. Repository 2 does not yet receive Repository 1's vote, and is not yet able to execute the transaction.

Step 3: Client 2 issues a single-repository transaction to Repository 1, with TID 2 and `highTS` 0. Repository 2 assigns this transaction `ts` 10, which is higher than `lastExecTS`. The repository logs, executes and responds to the client with `ts` 10, and sets `lastExecTS` to 10.

Step 4: Client 2 issues a single-repository transaction to Repository 2, with TID 3 and `highTS` 10. Repository 2 assigns this transaction `ts` 11, but cannot yet execute it, since TID 1 has not yet been executed and currently has `proposedTS` 3. The repository thus blocks execution of TID 3 until there are no other transactions with lower timestamps.

Step 5: Repository 2 finally receives Repository 1's vote for TID 1 (retry mechanisms are employed in the case of a lost message). Repository 2 assigns TID 1 a `finalTS` of 9, which is the highest timestamp from the votes. Repository 2 then executes the transactions in timestamp order, updates `lastExecutedTS` and responds to the clients.

This protocol ensures that serializability is maintained, with a global serial history of $\{\text{TID 1}, \text{TID 2}, \text{TID 3}\}$.

Serializability in Granola relies heavily on the inclusion of `highTS` values in client Requests. If Client 2 didn't include its `highTS` in Step 4, Repository 2 may have executed TID 3 ahead of TID 1. This would violate serializability, since the execution history $\{\text{TID 3}, \text{TID 1}\}$ at Repository 2 conflicts with the histories $\{\text{TID 1}, \text{TID 2}\}$ at Repository 1 and $\{\text{TID 2}, \text{TID 3}\}$ at Client 2. The propagation of timestamps on client requests prevents these anomalies from occurring.

4.4.3 EXTERNAL CONSISTENCY

Granola does not provide *external consistency* [37], meaning that consistency is not guaranteed when communication occurs outside the system. Consistency in Granola relies on clients to propagate timestamp dependencies, as seen in Step 4 of Figure 4.9. Out-of-band messages might not include these timestamps, and thus violate our serializability protocol.

External consistency *could* be provided if a new transaction was delayed until the completion of all independent transactions that were voted on before the transaction arrived. We aim to minimize latency for single-repository transactions in Granola, however, which precludes delaying transactions to this extent.

Granola was designed as a client-server protocol, where clients typically do not interact directly, except via the server state. This model matches that of a typical web service or distributed database. In these scenarios, external consistency is not required. If clients *are* expected to communicate directly, and external consistency is required, then client-to-client communication must include the `highTS` timestamp value of the sender.

In the absence of timestamps on out-of-band communication, external consistency can only be violated within a small window of vulnerability equal to the maximum clock skew between repositories. Since the delay between clients and repositories is typically significantly longer than the clock skew, violations of external consistency are unlikely in practice.

Proof of Window of Vulnerability

We present a proof here that external consistency can only be violated within a window of time equal to the maximum clock skew between repositories.

Consider a transaction T_1 issued to a set of participants $\{P_0, \cdots, P_n\}$. Let W be the wall-clock time by which point all participants have received T_1 and sent their votes.

We assume that one participant receives a full set of votes and executes T_1. A client observes the result of T_1 from this participant, communicates this result using an out-of-band mechanism to another client, and that client issues a subsequent transaction T_2 to participant P_0, without including an appropriate highTS value. We let B represent the time taken for this sequence of events to take place, hence P_0 will receive T_2 at wall-clock time $W + B$.

A violation of external consistency is only possible if P_0 assigns a lower timestamp to T_2 than to T_1, i.e.:

$$ts_2 \quad < \quad ts_1 \tag{4.1}$$

where ts_1 and ts_2 are the timestamps assigned to T_1 and T_2 respectively.

We can compute a lower-bound on ts_2, as assigned by repository P_0, as follows:

$$\begin{aligned} ts_2 &= \max(\text{clock value}, \texttt{highTS}, \texttt{lastExecTS}) \\ &= \max(W + B + C_0, \texttt{highTS}, \texttt{lastExecTS}) \\ &\geq W + B + C_0 \\ &\geq W + B + C_{min} \end{aligned} \tag{4.2}$$

where C_0 is the clock offset at repository P_0 and C_{min} is lowest (possibly negative) clock offset of all repositories.

T_1 will receive a timestamp equal to the highest vote from amongst the participants, hence:

$$\begin{aligned} ts_1 &= \max(\text{clock value}, \texttt{highTS}, \texttt{lastExecTS}) \\ &= \max(W_i + C_i, H_1, L_i) \end{aligned}$$

where P_i is the participant that voted the highest proposedTS for transaction T_i, W_i is the wall-clock time at which repository P_i received transaction T_1, and C_i is the clock offset at P_i. H_1 is the highTS value provided for T_1, and L_i is the lastExecTS value at P_i before time W_i.

$W_i \leq W$, by definition of W, and $C_i \leq C_{max}$, where C_{max} is the maximum clock offset of all repositories, hence:

$$\begin{aligned} ts_1 &= \max(W_i + C_i, H_1, L_i) \\ &\leq \max(W + C_{max}, H_1, L_i) \end{aligned}$$

In addition, $H_1 < W + C_{max}$, since the most recent transaction the client observed would have been received at timestamp $W' < W$, and timestamp $H_1 <= W' + C_{max}$. Similarly, $L_i < W + C_{max}$. We thus have:

$$ts_1 \leq W + C_{max} \tag{4.3}$$

Combining equations (4.1), (4.2) and (4.3), we have:

$$\begin{aligned} W + B + C_{min} \leq ts_2 &< ts_1 \leq W + C_{max} \\ W + B + C_{min} &< W + C_{max} \\ B &< C_{max} - C_{min} \\ B &< \text{maximum clock skew} \end{aligned}$$

External consistency can thus only be violated if the sequence of out-of-band communication occurs within a time period B, which is less than the maximum clock skew.

4.5 CONCURRENCY

We have described the Granola protocol in a sequential fashion to aid in clarity. Granola allows for significant concurrency however. This section describes the impact of concurrency on clients and repositories.

4.5.1 CLIENTS

The client API in Section 3.2.1 presents a blocking invocation interface to client applications. Clients can issue multiple concurrent transactions, however, by using multiple client application threads that issue invocations to a single client proxy, or by extending the client API to a non-blocking interface.[1]

Granola provides serializability for all requests, but the relative ordering of concurrent requests is determined by the protocol rather than by invocation order. The client proxy buffers responses to concurrent client requests, and returns them to the client application in timestamp order.

The client proxy may not always receive responses from different repositories in timestamp order. It thus sets `highTS` to `max(highTS, ts)` when processing a Reply messages, to ensure it always has a record of the *highest* timestamp it has observed.

4.5.2 REPOSITORIES

The message processing protocol in Granola is single-threaded, to minimize the overhead of locking and synchronization that would be required to support multiple concurrent handlers.[2] We also issue `run`, `prepare`, `commit` and `abort` upcalls one-at-a-time, to minimize concurrency control at the application. The protocol is largely asynchronous and non-blocking, however, and is able to handle many transactions in parallel. Other transactions are processed when waiting for timestamp constraints to be met, when waiting for votes to arrive, or when waiting for stable log writes to complete. Our replication protocol also allows for many log writes to be executed in parallel, as discussed in Section 5.1.

Granola's serializability constraint requires that transactions are executed in timestamp order when in timestamp mode, which can lead to blocking if a transaction has a final timestamp but is waiting for the execution of transactions with lower timestamps. Significant blocking is rare in practice, since timestamp constraints are usually met: the `highTS` value provided by clients is typically lower than the clock value at the repository, since clock skew between repositories is typically lower than the time between

[1] Our implementation of the client proxy is thread-safe and supports multiple application threads issuing concurrent transaction invocations.

[2] While our repository implementation uses a single handler thread, we also use additional threads to handle messaging and replication, as described in Section 6.1

subsequent client requests. Voting also completes quickly, in a single one-way message delay.

As in all systems that offer strong consistency, there may be a period of unavailability if failures occur within a specific *window of vulnerability*. In some rare circumstances a repository may stall if it is unable to communicate with a participant for a given transaction. This reflects a trade-off made to provide consistency and performance, at the expense of a potential loss of availability. We define our window of vulnerability, and describe techniques to mitigate failures, in Chapter 5.

5 FAILURES AND RECOVERY

This chapter discusses the mechanisms used in Granola to tolerate failures. We first discuss the state machine replication protocol used to enable repository fail-over, followed by a discussion of the protocols used within Granola for handling the failure of individual nodes. We then discuss protocols for dealing with the correlated failure of the nodes that comprise a replicated repository, and for dealing with network partitions that cause a repository to become unavailable. Finally we discuss the mechanisms for retrying communication in response to network loss or failures, and the use of duplicate detection to identify retransmissions.

5.1 REPLICATION

Repositories implement state machine replication [48] to provide durability and fail-over in response to individual node failures. State machine replication is used to replicate each stable log write, and to replace failed nodes. Disk writes are commonly used in databases to provide durability for log writes, but they do not provide fast recovery from failure.

Our replication protocol is based on Viewstamped Replication [38, 43], which implements a consensus protocol similar to Paxos [35]. Repositories are replicated, using a set of $2f + 1$ replicas to survive f crash failures.

One replica is designated the *primary*, and carries out the Granola protocol. *Backup* replicas initiate a *view change* to select a new primary if the old one appears to have failed. All messages from clients and other repositories are sent directly to the primary replica at the repository. If a backup replica receives a Granola message, it forwards it to the primary. The only communication with backup replicas is via the replication protocol.

We made a number of improvements to the traditional Viewstamped Replication protocol to improve performance and manageability, including:

- Modifications to the *view change* protocol so that disk writes are not required.
- Batching to improve throughput under high load.
- Application state diffs perform state-transfer more efficiently.
- Fewer message delays for read/write operations.
- A reconfiguration protocol to allow moving the replica group to a different set of nodes, or to allow changing the size of the replica group.

A comprehensive description of our replication protocol, including the improvements above, is available in our paper *Viewstamped Replication Revisited* [39].

Stable log writes involve the primary sending the log message to the backups as a PREPARE message and waiting for f PREPAREOK replies, at which point the log is stable. This process takes only two one-way message delays, as shown in Figure 5.1. The primary does not stall while waiting for replies, and may continue processing incoming requests in the meantime. The replication protocol uses batching of groups of log messages [18], to reduce the number of messages sent to backups; this is analogous to group commit, where a number of log messages may be committed simultaneously.

Our protocol does not require disk writes as part of the stable log write, since we assume that replicas are failure-independent. Information

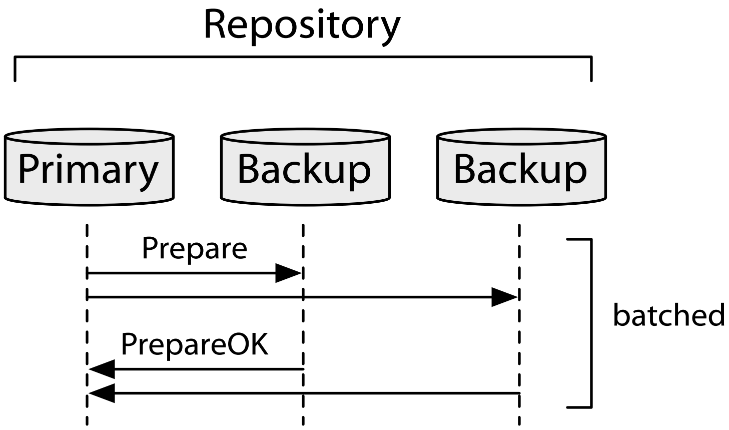

Figure 5.1: The basic Viewstamped Replication protocol as used in Granola to provide stable log writes. More complex communication is required when issuing a view change to recover from node failures.

is written to disk in the background, e.g., as required by main memory limitations. Failure-independence can be achieved by locating replicas in disjoint locations, storing log entries in battery-backed RAM, or equipping replicas with a UPS and flushing to disk when a power failure occurs.

5.2 INDIVIDUAL FAILURES

Granola is designed to seamlessly tolerate the failure of individual nodes. This section describes the protocols for recovering from repository and client failures.

5.2.1 REPOSITORY RECOVERY

We now discuss how the replication protocol is used to support recovery at each repository.

Logging

The primary replica at each repository follows the Granola protocol as described in the previous chapters. Backup replicas receive and log all stable log writes. The primary also piggybacks, on subsequent log messages, information about the commit status and final timestamps assigned to each transaction that has been executed, which is also logged by the backups.

Each log message is of the form ⟨`msg`, `proposedTS`, `status`, [`finalList`]⟩ where `msg` is the client's Request message, `proposedTS` is the currently-assigned timestamp for the transaction, `status` is the current commit or abort status (`COMMIT`, `CONFLICT` or `ABORT`), and `finalList` is an optional list of piggybacked commit information. `finalList` takes the form ⟨TID, `finalTS`, `status`⟩, and includes the TID, final assigned timestamp, and final status for transactions that have recently completed.

Each backup replica stores a log as shown in Figure 5.2.[1] Each entry in the log corresponds to a transaction, sorted by timestamp, and records the logged transaction information, along with a `final` flag indicating whether the transaction has a `finalTS` or `proposedTS` timestamp. Timestamps are only recorded as final once a piggybacked message is received indicating the final timestamp for that transaction. Non-final timestamps for distributed transactions that are still awaiting votes can be modified in subsequent log messages. Non-final timestamps for single-repository transactions can also be modified later if the transaction is reassigned a timestamp due to a lock conflict.

The log contains a prefix (indicated in the figure) of entries with a final timestamp value. Since the log is stored in timestamp order, the transactions corresponding to these final entries can be executed and removed from the log.

The replica maintains the same data structures for duplicate detection and retransmission as the primary, so that this information is available if the replica becomes the primary in a future view. These structures are described in Section 5.4.

View Changes

The replication protocol triggers a view change if the primary is faulty, which promotes one of the backups to be the new primary. The view

[1] This log should not be confused with the internal logging in the Viewstamped Replication protocol, which is treated as a black box by Granola.

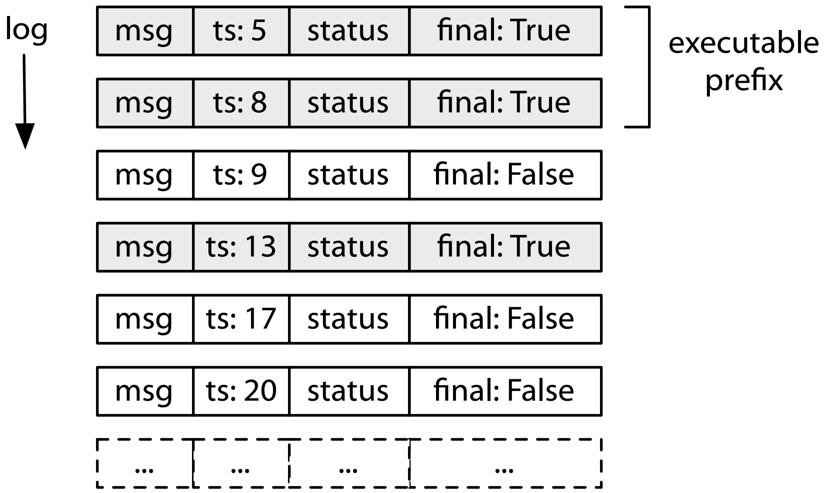
log
msg | ts: 5 | status | final: True
msg | ts: 8 | status | final: True
msg | ts: 9 | status | final: False
msg | ts: 13 | status | final: True
msg | ts: 17 | status | final: False
msg | ts: 20 | status | final: False
... | ... | ... | ...
executable prefix

Figure 5.2: Log recorded at each non-primary replica. The log is stored in timestamp order, and records each current transaction. Each shaded log entry represents a transaction that has been assigned a final timestamp. The indicated prefix of shaded log entries may be executed and pruned from the log.

change protocol ensures that the new primary has a local state that includes log entries for all transactions that completed a stable log write at the old primary. This information will include any stable log write that has occurred, but may not contain information about operations that happened at the old primary after this point, such as votes being sent or received, or subsequent transaction execution. This information can be recovered, however, by re-running the voting protocol for these transactions, based on information in the log.

The new primary temporarily halts receiving new transaction requests, and responds with a `CONFLICT` vote for these transactions to avoid blocking other participants while the view change proceeds. The new primary then iterates through the stable log for all entries that do not yet have a final timestamp (i.e., those where `final` is false), and sends a ⟨Vote, `tid`, `rid`, `proposedTS`, `status`, `retry`⟩ message to the other participants for these transactions, with the `proposedTS` and `status` from the log. The TID is obtained from each `msg`, and the same RID is shared by all replicas for the repository. The `retry` flag is set to true in these votes, to prompt the recipients to resend their previous votes for the transactions.

The new primary can then resume normal operation and start accepting new transactions, while handling the responses to the Vote messages using the standard request processing protocol. This protocol requires that other repositories keep a log of old Vote messages to facilitate resending votes for a repository that just underwent recovery. We discuss maintenance and garbage collection for this state in Section 5.4.

Efficiency

Granola imposes significantly less load on the backups than the primary, since the backups do not need to manage locks, send votes and responses, or attempt to execute transactions that were later aborted. The backups also do not need to execute read-only transactions, since these do not modify the state on the replicas; this is of significant benefit for typical read-heavy workloads.

Load on the backups can be reduced even further by designating f of the nodes in each replica group as *witnesses*, which participate in the protocol only when other replicas are unresponsive or have failed [38]. Since witnesses have no computational load under normal circumstances, they can be colocated on nodes used as replicas for other groups. Since any backup or witness may be required to act as a primary under some failure conditions, they must be capable of supporting the full request load.

In wide-area deployments, repositories that share related data can also have their primaries colocated in the same data-center where possible. This minimizes voting latency in the absence of failures. Protocols for dynamically reconfiguring replica groups to relocate replicas and witnesses are left as future work, as described in Section 9.1.3. We describe how to migrate *data* between repositories in Section 7.1.4.

Ensuring a Single Primary

State machine replication may result in two replicas concurrently believing they are the primary. This can be the case, for example, when there is message loss and the old primary does not hear about a view change to a new primary. Some clients may also not immediately learn of the view change, and continue issuing requests to this old primary.

Read-write transactions do not pose a problem here, since the primary must perform a stable log write before processing each transaction, and will discover that a view change has occurred. Read-only transactions execute solely at the primary replica however. If this replica is no longer the current primary, it may issue responses to reads that do not include recent writes, and hence violate serializability.

We avoid this problem by using the leases mechanism introduced in Harp [38], which guarantees that there will not be two primaries running simultaneously. Each primary can accept operations only if it holds an active lease from f other replicas. Replicas wait at least as long as this lease time before initiating a view change, ensuring that the old primary can no longer accept requests. The lease protocol depends on loosely synchronized clock rates for correctness.

We introduce a protocol for executing read-only transactions at backup replicas in Section 7.2. This protocol not only reduces load on the primary replica, but also avoids the requirement of synchronized clock rates for correctness.

Discarding Unlogged State

Section 4.3.5 described how single-repository transactions can be run without locking even when a repository is in locking mode, by executing the transaction before issuing the stable log write. If the primary replica fails or a view change occurs before the stable log write is complete, however, any such transaction will not persist into the new view. This protocol ensures correctness by not externalizing the effects of the single-repository transaction until after the stable log write. The client will eventually retry the transaction in the new view.

Under ordinary failure conditions, the old primary will have crashed and will have lost its local state. When the old primary recovers, it will rebuild its state using the Viewstamped Replication state transfer protocol, before rejoining the replica group. If the old primary did *not* lose its state, however, it may have executed some single-repository transactions that didn't persist into the new view. When the old primary becomes aware of the new view, it must roll back its state so as not to include any unlogged single-repository transactions. One solution to this problem is to maintain undo records for any single-repository transactions that have not yet been logged. Since this is a rare occurrence however, the most simple solution is to require the old primary to roll back its state to the most recent Viewstamped Replication checkpoint, and replay any new transactions from the Viewstamped Replication log [39].

5.2.2 CLIENT RECOVERY

Clients can fail without affecting the operation of the system as a whole. Repositories use retry mechanisms to ensure that a distributed transaction will complete even if the client fails before sending it to all repositories, as described in Section 5.4. We need to ensure, however, that a client chooses an appropriate sequence number and timestamp before issuing a subsequent request after recovering.

Sequence Numbers

Sequence numbers are used to generate TIDs, which uniquely identify each transaction. A client must thus ensure that it does not reuse a sequence number from a transaction that was active before it failed.

We ensure uniqueness of sequence numbers by using sequence number leases. Each client records a sequence number range on disk (e.g., 10000–20000) and may freely use sequence numbers within this range. The client records a subsequent range once the current sequence number space has been exhausted. When a client recovers from failure, it reads the previous range from disk, and records a lease for a subsequent range before resuming operation. Any sequence number chosen from this new range is guaranteed to be higher than that used before the failure occurred.

Timestamps

Clients are required to include their highest-observed timestamp, `highTS`, in each request, to ensure that each request executes after any transaction that the client previously observed. This value is redundant, however, once all repositories have a local clock value higher than the `highTS` value at the client; at this point any subsequent transaction will always be issued a timestamp higher than the highest timestamp observed by the client.

A client determines a safe `highTS` value after failure by first synchronizing its clock, and then ensuring that a period of time equal to the maximum expected clock skew has elapsed. After this point the client adopts its local clock value as its new `highTS` value, which will be at least as high as the timestamp it knew before it failed. Typically a client will have failed for longer than the maximum expected clock skew, and hence no waiting is required.

Note that here we are depending on loosely synchronized clock values for correctness, where otherwise we have not needed this assumption.[2] If it is infeasible to depend on loosely synchronized clocks, clients may adopt timestamp leases, similar to sequence number leases, where they will only accept a response if it falls within a timestamp range it has written to disk. If a response has a timestamp higher than this range, the client will write a new lease to disk before processing the response.

In some deployment scenarios there are no explicit consistency constraints between client sessions, e.g., when the client keeps no stable local record of transaction state. In these deployments the client recovers without any knowledge of previous transactions, and can safely set its latest-observed timestamp to 0.

[2] Our view change protocol relies on loosely synchronized clock *rates* for correctness, which is an easier guarantee to provide than loosely synchronized clock *values*.

5.3 CORRELATED FAILURE

Replication masks the failure of individual replica nodes, but cannot provide availability if an entire replicated repository becomes unavailable. Given that a set of $2f+1$ replicas can tolerate at most f simultaneous node failures within the group, the repository will become unavailable if more than half of the replicas fail simultaneously. If replicas are located in failure-independent partitions, this level of correlated failure will typically only be the result of a network partition that results in a significant fraction of replicas being unable to communicate with the rest of the system.

Permanent loss of an entire replicated repository may result in data loss and will likely require human intervention. This section instead addresses the situation where a good repository is unable to obtain votes from an unresponsive participant. When this happens the repository may have some *incomplete* distributed transactions that are awaiting votes from the participant. The repository is unable to unilaterally abort an incomplete transaction, since the unresponsive participant may have already received a full set of votes and executed the transaction before failing.

Read-only distributed transactions are not included in the set of incomplete transactions, since they do not modify the application state at any of the participants, and hence the repository can issue a `CONFLICT` abort to the client and discard the transaction; the client will retry the transaction with a new TID if it receives a `CONFLICT` response. Since we expect typical workloads to consist of a large fraction of read-only transactions, this can significantly reduce the number of incomplete transactions.

A repository first makes multiple attempts to establish communication with a failed participant. The repository attempts to resolve incomplete transactions with other participants by resending each VOTE message with the `retry` flag set to true. If any participant has already committed or aborted the transaction, it will respond with a REPLY message indicating the transaction result. The repository can then adopt this outcome and resume normal operation.

If the repository is in locking mode, it will have already acquired locks for the incomplete transactions. The repository must thus continue holding locks for these transactions until the participant recovers and sends its vote. Any subsequent transaction received in the meantime will be aborted with a `CONFLICT` response if it conflicts with a lock held for an incomplete transaction. This protocol allows the repository to make progress provided that an incomplete transaction does not lock a significant fraction of the application state. If a participant fails while a transaction is holding locks on the entire application state, then the repository will be stalled until the participant recovers; this fate-sharing between transaction participants is also encountered in two-phase commit.

Recovery from timestamp mode is more complex, since the repository does not already have locks on the independent transactions. We discuss the recovery protocol for timestamp mode in the following section.

5.3.1 RECOVERY FROM TIMESTAMP MODE

If the repository is running in timestamp mode, it will not have acquired locks on the incomplete transactions, and must thus execute transactions in timestamp order. In addition to the set of incomplete transactions, this results in a set of *blocked* distributed transactions, for which the repository has a full set of votes and has assigned a final timestamp, but can't yet execute because they are queued behind an incomplete transaction with a lower timestamp.

An example of a queue of incomplete and blocked transactions at a given repository is shown in Figure 5.3. The repository has sent votes for transactions 31 and 12, but has not yet received votes from a failed participant. Transactions 2, 16, 54 and 8 have a full set of votes and a final timestamp, but can not yet be executed because they do not have the lowest timestamp at the repository.

Neither single-repository transactions nor read-only independent transactions are included in the set of blocked transactions. Since the repository has not yet responded to the client for the read-only independent transactions, it can safely discard the transaction and respond to the client with a

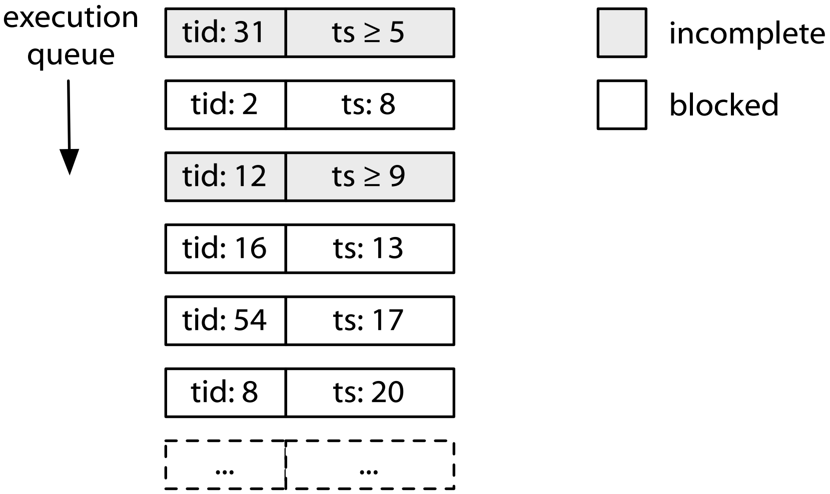

Figure 5.3: Queue of logged transactions at a stalled repository, in timestamp order. The shaded transactions are *incomplete* awaiting votes from a failed participant and do not yet have a final timestamp. The non-shaded transactions have a final timestamp, but are *blocked* behind incomplete transactions, waiting to be executed in timestamp order.

`CONFLICT` abort response. Single-repository transactions can also be aborted, by issuing a stable log write to record the abort status at the backups, then responding to the client with a `CONFLICT` abort response. The client proxy will retry these transactions with a new TID, but the repository will not process these retries until the blocking has been resolved.

Similarly, distributed transactions that the repository has not yet voted for can also be safely aborted, and are not included in these sets. As an optimization, the repository may process single-repository transactions that have a `highTS` value lower than the lowest incomplete transaction, since these transactions can be assigned a timestamp ahead of the incomplete transactions in the serial order, and be executed without blocking.

The relative ordering of independent transactions is important in timestamp mode, since the transactions might conflict. Transactions cannot conflict in locking mode, since the repository checks locks before sending a vote, and aborts the transaction if there is a lock conflict. Concurrent independent transactions may modify the same data items, and thus their results may depend on the order in which they are run. This order depends on the final timestamp order, which is not yet known for independent transactions.

The objective of recovery in timestamp mode is to acquire locks on the entire set of incomplete and blocked transactions, so that the repository can process future transactions that do not conflict with these locks. The repository can also execute blocked transactions that do not conflict with the remaining transactions.

If none of the blocked and incomplete transactions conflict, the repository can transition into locking mode immediately and follow the protocol discussed above. If there is a lock conflict however, the repository must acquire these locks in an order-independent way. We discuss these protocols in the following sections.

Transitioning into Locking Mode

The repository first attempts to transition into locking mode, using the same protocol for switching into locking mode under normal operation, by acquiring locks on the incomplete and blocked transactions in the current timestamp order. This order is defined by the `proposedTS` for incomplete transactions and the `finalTS` for blocked transactions. Locks are acquired by iterating through the list of incomplete and blocked transactions, and executing a `prepare` upcall to the server application, stopping if there is a lock conflict.

If the repository receives a `COMMIT` response for a `prepare` upcall, then the application was able to acquire locks for the transaction in this order, without any conflicts. If the prepare is successfully able to acquire locks for an *incomplete* transaction, the repository will continue holding the locks. If the prepare is successful for a *blocked* transaction, then the repository executes it immediately by issuing a `commit` upcall, and responds to the client; the blocked transaction will not conflict with any incomplete transaction possibly ahead of it in the final serial order, since any incomplete transaction after it in the current timestamp order will eventually be assigned a `finalTS` timestamp at least as high as its current value.

If the repository receives a `COMMIT` response for every `prepare` upcall, then it will have executed every blocked transaction, and acquired locks on all incomplete transactions. It may thus transition into locking mode, and resume operation following the locking mode protocol in the previous section.

If one of the `prepare` upcalls returns with a lock conflict, however (i.e., a `CONFLICT` response), the repository must halt iterating over the transactions. In the absence of conflicts, the repository can acquire locks for transactions in any order, since none of the transactions interfere. If there is a lock conflict, however, the repository cannot attempt to prepare subsequent transactions in the queue, since they may depend on the locks that should have been acquired for the conflicting transaction.

In order to make progress, the repository must ensure that the application requires locks for all remaining transactions regardless of what order they are serialized in. We describe the relevance of lock ordering in the following section, followed by the interface and protocol for obtaining locks irrespective of ordering.

Relevance of Lock Ordering

Blocked transactions have a `finalTS` timestamp, and will maintain their position in the serial order. Incomplete transactions only have a `proposedTS` timestamp, however, and will end up being serialized at some `finalTS` $\geq$ `proposedTS`, once the remaining votes are eventually received.

If the repository is able to acquire locks on all transactions in the incomplete and blocked sets, then these transactions must not conflict and can be committed in any order. If there is a lock conflict, however, the order in which locks are acquired may affect transaction outcomes.

An example of a transaction where the lock ordering matters is shown in Figure 5.4. This transaction performs a bank transfer if there are sufficient funds, or otherwise does nothing. The application acquires locks on both account balances if there is sufficient funds to perform the transfer, but only one on one balance if there are insufficient funds.

Consider three accounts, `A`, `B` and `C`, which are replicated across a set of

```
/*
 * Simplified transaction operation to transfer money from
 * one account to another, if there are sufficient funds.
 *
 * Ignores details of undo logging, return status, etc.
 */
void transferFunds(Account from, Account to, int amount) {
    lock(from);
    if (from.balance() >= amount) {
        lock(to);
        from.withdraw(amount);
        to.deposit(amount);
    }
}
```

Figure 5.4: Simple funds-transfer operation where the lockset is a function of the system state.

repositories, and start off with balances of \$0, \$0 and \$10 respectively. The following two transactions are issued concurrently:

T_1: `transferFunds(A, B, 10)`

T_2: `transferFunds(C, A, 10)`

If T_1 is serialized before T_2, the first transfer will fail, and the cumulative lockset for the two transactions will be $\{a, c\}$. If T_2 is serialized before T_1, however, both transfers will succeed, and the cumulative lockset will be $\{a, b, c\}$. If one of T_1 or T_2 is a incomplete transaction, the repository does not know which order they will eventually be serialized in, and hence needs to acquire all locks that could be required in any possible ordering.

Order-Independent Locking

Our solution to the problem of transaction ordering is to require the application to acquire locks for transactions independently of the order in which they are prepared. In the case of the bank transfer, this would involve acquiring locks on the combined lockset $\{a, b, c\}$, since it encapsulates a superset of all possible orderings. We refer to the complete lockset for a transaction irrespective of ordering as the *lock-superset* for the transaction. The lock-superset is only required during recovery from failure, and is not used in normal-case processing.

Acquiring a lock-superset is closely related to lock granularity escalation in a DBMS. If a transaction updates different individual tuples depending on database state, then escalation from tuple-level locks to range locks or table-level locks may be used to cover all possible tuples.

Granola's name service uses coarse-grained IDs to map data partitions to repositories, as described in Section 7.1.2. In workloads that are predominately comprised of independent transactions, these IDs can be used as the unit of lock granularity, since locking will only be employed when recovering from correlated failure. In these scenarios the locksets are also independent of the transaction ordering.

Avoiding Order-dependent Transactions For many applications, transactions are already order-independent, and the lock-superset is the same as the set of locks acquired by a regular `prepare` upcall. This is particularly the case for transactions that update fields that are known ahead of time [10]. The Sinfonia protocol is applicable to a wide variety of transaction processing applications [10], yet requires that the locksets for each transaction are provided *a priori* by the client. Any application that can be implemented using Sinfonia could also be implemented using Granola, using the client-provided lockset as the lock-superset.

Thomson and Abadi argue that locksets can be computed in advance for a large fraction of transactions in online transaction processing workloads [53], and hence these locks are independent of transaction ordering. For transactions where locks *do* depend on transaction ordering, they propose decomposing a transaction into two sub-transactions: a read-only transaction that determines which tuples will be locked, followed by a read-write transaction that will abort if the set of tuples has changed in the meantime. The authors argue that the overhead for such an approach is "almost negligible" in typical workloads.

```
// acquires any locks that could be required if preparing
// the trans at any point in the serial order
// acquires additional handle on locks if there's a conflict
// returns true if no conflict, but acquires locks regardess
boolean forcePrepare(ByteBuffer request, long tid);
```

Figure 5.5: Recovery Interface. Applications must extend this interface to support recovery from correlated failure while in timestamp mode.

For example, the TPC-C `payment` transaction may have to look up a customer name to determine the primary key corresponding to the customer. This `payment` transaction can be broken down into a single-repository transaction that first looks up the key corresponding to the customer name, followed by an independent transaction that deterministically locks the record indexed by this key, and will abort if the mapping is no longer valid. The second transaction will only abort if the customer name is changed extremely frequently, which is an unlikely scenario. The authors make the general claim that "real-life OLTP workloads seldom involve dependencies on frequently updated data" [53], and hence it is reasonable to assume that a large fraction of transactions can be expressed such that the lockset is equivalent to the lock-superset.

Recovery Interface

Acquiring lock-supersets for conflicting transactions requires extensions to the server application interface. Figure 5.5 shows this application interface.

The `forcePrepare` upcall is similar to the `prepare` upcall, but it differs in two key ways:

1. The application acquires the *lock-superset* for the operation, rather than the order-dependent lockset.

2. The application must acquire locks on the full lock-superset, regardless of whether there are lock conflicts. The application must acquire multiple handles on the same lock in the case of conflict.

The second point is required since multiple transactions in the queue may require the same locks. If transaction `t1` has lock-superset $\{a, b\}$ and transaction `t2` has lock-superset $\{b, c\}$, then the application must acquire lock b for both transactions.

The application returns true for the `forcePrepare` upcall if there were no conflicts, but acquires the locks even if returning false. The locks acquired by the `forcePrepare` will be released when the repository commits or aborts the transaction, as described in the following section.

Recovery Algorithm

After attempting to transition into locking mode, as described previously, the repository will have a set of incomplete transactions for which locks were acquired, followed by a set of blocked and incomplete transactions that have not yet been locked. The repository first issues an `abort` upcall for the first set of transactions, to release their locks in preparation of adopting order-independent locking.

The repository then iterates through the incomplete and blocked transactions in current timestamp order, and issues a `forcePrepare` upcall for them. If the application returns true in response to a `forcePrepare` upcall for a *blocked* transaction, then the transaction must not conflict with any transaction possibly ahead of it in the final serial order, and can be committed and executed then removed from the queue.

Once the lock-supersets have been acquired for all transactions in the incomplete and blocked queue, the repository can switch into locking mode and resume accepting new transactions. The remaining incomplete and blocked transactions can be executed in timestamp order, once votes are eventually received for the incomplete transactions and their position in the serial order is known.

Fine-Grained Recovery

The protocol described in the previous section assumes that it is possible to determine a lock-superset for each transaction, which is independent of the order in which the transaction is run. This is possible for all transactions, but in the degenerate case it may involve acquiring a lock on the entire application state.

Our original protocol for recovery did not require the application to compute lock-supersets, and instead issued a series of upcalls to the application to acquire individual locksets for the incomplete and blocked transactions, in all possible execution orders.

There are many constraints on the ordering of these transactions, which reduces the number of orderings that need to be evaluated. The order of blocked transactions is known once they have been assigned their final timestamps. There are also ordering constraints between blocked and incomplete transactions, since a incomplete transaction will end up with a final timestamp at least as high as its current proposed timestamp. Moreover, if multiple transactions commute, then any permutation of their respective orderings is equivalent.

Given these constraints, we designed and implemented a search algorithm that explored all possible distinct transaction orderings, and determined the combined locksets for these orders. This search process is efficient if the majority of transactions are blocked and there are only a few incomplete transactions. The algorithm is unfortunately extremely inefficient if there are a large number of conflicting incomplete transactions: in the worst-case the search space is $O(n!)$ in the number of incomplete transactions, which is clearly prohibitive in many failure scenarios. We thus instead opted to require a concept of lock-supersets, and avoid thus this search process.

5.4 RETRIES AND DUPLICATE DETECTION

Each repository maintains information about recently executed transactions, to allow it to resend messages to clients or participants in case of message loss or a failure, and to identify duplicate messages. Granola uses fairly standard techniques for handling retries, duplicate detection and garbage collection, which we outline here for completeness.

Responding to Repositories

The repository stores a copy of each transaction REQUEST it receives, and can use this to forward the transaction to a participant that has not received the request. The repository can discard a REQUEST message once it has received a full set of votes for the transaction, since every participant must have a stable record of the transaction before it sends its vote.

The repository must also store a copy of VOTE messages it has sent, since a vote may be lost due to network failure or a view change that occurred at a participant. If a participant is missing a vote from the repository, it will resend its own VOTE message, including a `retry` flag set to true. The retry flag signals to the repository to resend its vote.

The repository is able to discard a VOTE message once all participants have executed the transaction and piggybacked the final transaction status into their log. Participants communicate this information to each other in acknowledgment messages or in subsequent votes.

Responding to Clients

The repository must keep a copy of recent REPLY messages, to identify duplicate client requests and resend previous replies in case of message loss. The repository cannot rely on acknowledgments from clients, however, since a client may become disconnected or fail without notice. Any unacknowledged REPLY messages are thus stored in a circular buffer, which facilitates resending replies to clients within a certain time window. If a client is disconnected for a period longer than this window, the reply may be lost, and the client will have to issue a subsequent transaction to observe the current server application state.

6 EVALUATION

This chapter presents an evaluation of Granola's performance characteristics. We use a set of microbenchmarks along with theoretical models to examine the throughput, latency and scaling characteristics of the system under a variety of configurations. We also perform a detailed analysis of how often a repository operates in timestamp mode, where it is able to exploit the main advantages of the Granola protocol. We follow this analysis with a set of real-world macro-benchmarks, using the TPC-C transaction processing benchmark.

6.1 IMPLEMENTATION

We implemented Granola in Java, with full support for the API presented in Figures 3.3 and 3.4. This implementation provides reliability and recoverability, including message ordering, duplicate detection, retries, and recovery from client and repository failures. The implementation does not include a full distributed replication protocol; replication is implemented locally at each repository using a model of a batched replication protocol, using delay measurements from our full implementation of Viewstamped Replication [39] on our testbed.

We use TCP for communication between nodes, but implement our own retransmissions and do not rely on TCP guarantees for correctness. Message serialization is performed using Google's Protocol Buffers library [9]. Network communication is handled by a non-blocking network library we built using Java NIO and a single selector thread.[1]

Repositories are implemented using an event-based model which uses a single handler thread to process transaction requests, since the Granola protocol does not contain disk stalls or user stalls. Application upcalls are blocking and execute in this same handler thread in our implementation; the Granola protocol allows for running the application in a separate thread, but this was not deemed necessary in our benchmarks.

6.1.1 WORKLOADS

Our micro-benchmarks examine a counter service built on the Granola platform. Each update modifies either a counter on a single repository, or counters distributed across multiple repositories. We vary the conflict rate for coordinated transactions by adjusting the ratio of conflicting updates issued by a client. Since the protocol for read-only operations is similar to many other distributed storage systems, these benchmarks focus exclusively on transactions that mutate (and observe) data.

We also examine an implementation of the TPC-C transaction processing benchmark, built atop the Granola platform. This benchmark is discussed in more detail in Section 6.7.

6.1.2 SINFONIA

We compare the performance of Granola against the transaction coordination protocol from Sinfonia [10], which uses somewhat a more traditional lock-based version of two-phase commit. This protocol requires the involvement of dedicated master nodes to coordinate distributed transactions, and

[1] We also developed a multi-threaded thread-per-connection network library with blocking network IO, while experimenting with system performance. This library achieved similar performance to the non-blocking version, but offered lower throughput when under congestion collapse.

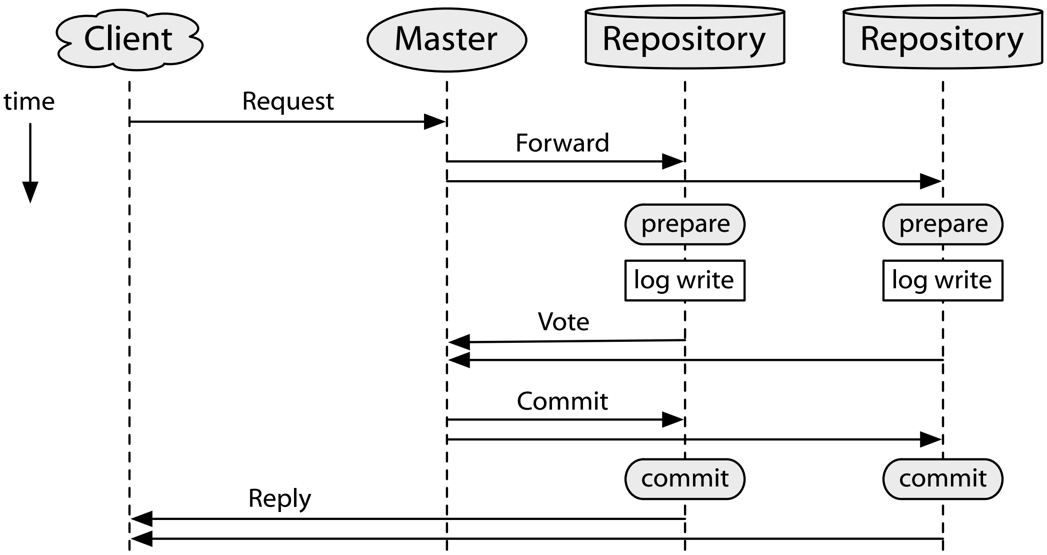

Figure 6.1: Protocol timeline for distributed transactions in the Sinfonia/Two-Phase Commit implementation.

involves an additional phase of communication compared to Granola when transactions are issued by a remote client. Unlike many implementations of two-phase commit, however, the Sinfonia protocol only requires a single stable log write, and supports one-round transactions, as in Granola. The Sinfonia protocol for distributed transactions is illustrated in Figure 6.1. The protocol for single-repository transactions is similar to the protocol used in Granola, except that locking is used instead of timestamps.

Sinfonia does not support general operations, and instead requires clients to explicitly specify the lock sets in each transaction request.[2] We extended the Sinfonia protocol by allowing the use of `prepare` application upcalls to determine transaction lock sets on the server side. This provides a fair comparison for Sinfonia against Granola, and allows support for complex applications such as TPC-C.

Our implementation of Sinfonia uses both read locks and write locks to avoid conflicts for read-only transactions, and avoids deadlock by aborting any transaction that encounters a lock conflict. The client retries aborted

[2] The Sinfonia authors present an outline of how the protocol could be used to support general operations, but do not implement or fully-develop these extensions [10].

transactions after a binary exponential backoff. Locking is required for all distributed transactions, since Sinfonia does not support a timestamp-based concurrency control mode. We avoid locking for single-repository transactions when there are no concurrent distributed transactions, although locks must be acquired when there are active distributed transactions.

We used a single master node in our experiments, and hence our Sinfonia results do not suffer from aborts due to contention from multiple masters [10].

6.1.3 EXPERIMENTAL SETUP

We deployed our implementation on a cluster of ten 2005-vintage dual-core 3.2 GHz Xeon servers with 2 GB RAM to serve as repository nodes, along with ten quad-core 2.5 GHz Core2 Quad desktop machines with 4 GB RAM to serve as client nodes. The Core2 Quad desktop machines offer significantly higher performance than the older Xeon nodes, and are used to support multiple concurrent clients on a given machine, to fully load the repositories. We use a high-performance 16-core 1.6 GHz Xeon server with 8 GB RAM to serve as the master node when comparing against Sinfonia, since Sinfonia experiences significant load on the master. These machines were connected by a gigabit LAN with a network latency of under 0.2 ms. Wide-area network delay is emulated by blocking outgoing packets in our network library.

The results in this chapter present the maximum *practical* system throughput observed on each configuration. As is typical when increasing request load, Granola eventually reaches a point of maximal capacity, at which point latency rapidly increases and throughput plateaus. When operating with our non-blocking network library and a very large number of clients, we were often able to push the system beyond this point, achieving even higher throughput at a cost of excessive client delay due to queuing in the protocol. In our experiments in this chapter we instead set client load appropriately to maximize throughput while keeping latency within reasonable bounds, on the order of a few milliseconds.

Our figures show 95% confidence intervals for all data-points.

6.2 BASE PERFORMANCE

We first investigate the base throughput and latency of the Granola protocol, for single-repository transactions on a single repository, and for distributed transactions on two repositories. Each repository runs on a single compute core on a single machine. Replication is simulated by running the replication protocol locally at the repository. We insert appropriate network delay to emulate local-area and wide-area replication configurations; 0.1 ms one-way network delay is used in the local area, and 10 ms one-way delay in the wide area.

6.2.1 SINGLE-REPOSITORY TRANSACTIONS

We varied the number of concurrent client threads and measured the throughput and latency for single-repository transactions. Each client thread issues one request at a time, with no delay between requests. Figure 6.2 plots the throughput of a single repository as a function of the number of concurrent client threads, spread over a total of ten physical client machines.

Both configurations reach a maximum throughput of over 60,000 tps (transactions/second) on a single compute core. More clients are required to saturate the repository in the wide-area configuration, since the per-transaction latency is higher and hence per-client throughput is lower.

Throughput is CPU-bound in our experiments, and benefits significantly from higher-performance hardware. We also ran these experiments with the repository located on a more modern 2008-vintage 2.83 GHz Core2 Quad desktop machine, with 4 GB of RAM. We were able to consistently achieve over 100,000 tps on a single compute core on this machine, with throughput peaking at 140,000 tps in the local-area configuration.

We show the average per-request latency for these experiments on our original set of machines in Figure 6.3. Total latency overhead is within 10% of the underlying replication delay until the point where client load exceeds

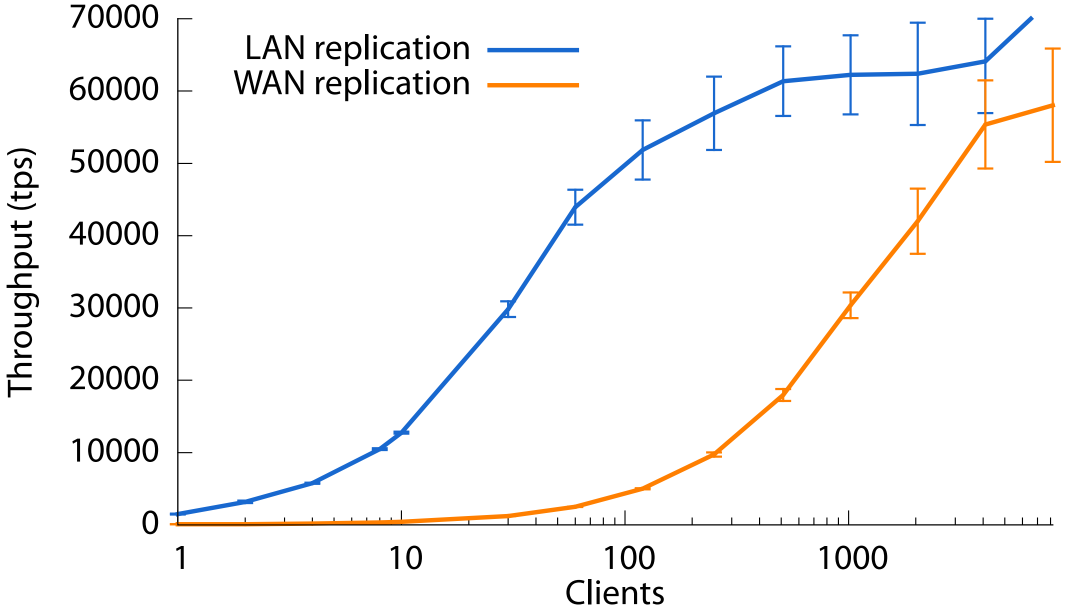

Figure 6.2: Single-repository throughput with increasing client load, for both local-area and wide-area replication configurations.

total system capacity. Wide-area replication does not impose a significant throughput or latency overhead, since our replication protocol can handle multiple transactions in parallel.

6.2.2 DISTRIBUTED TRANSACTIONS

Figure 6.4 shows the throughput for distributed transactions on a two-repository topology. This figure plots throughput for independent transactions; coordinated transactions also exhibit the same performance when there is no locking overhead and no conflicts.

We observe a maximum total throughput of over 20,000 transactions per second. This value is less than that observed for single-repository transactions, but a relatively small penalty for a workload comprised entirely of distributed transactions. We examine the impact of distributed transactions in more detail in the following sections.

We show average per-transaction latency for distributed transactions in Figure 6.5. Distributed transactions incur only a small latency overhead

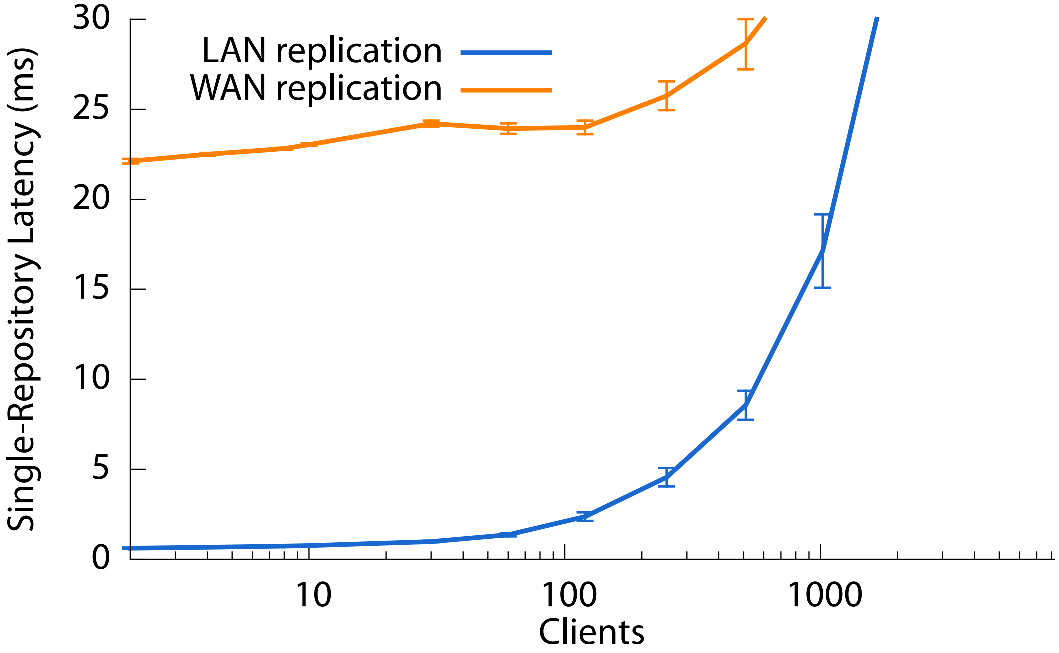

Figure 6.3: Per-transaction latency for single-repository transactions, with increasing client load, for both local-area and wide-area replication configurations.

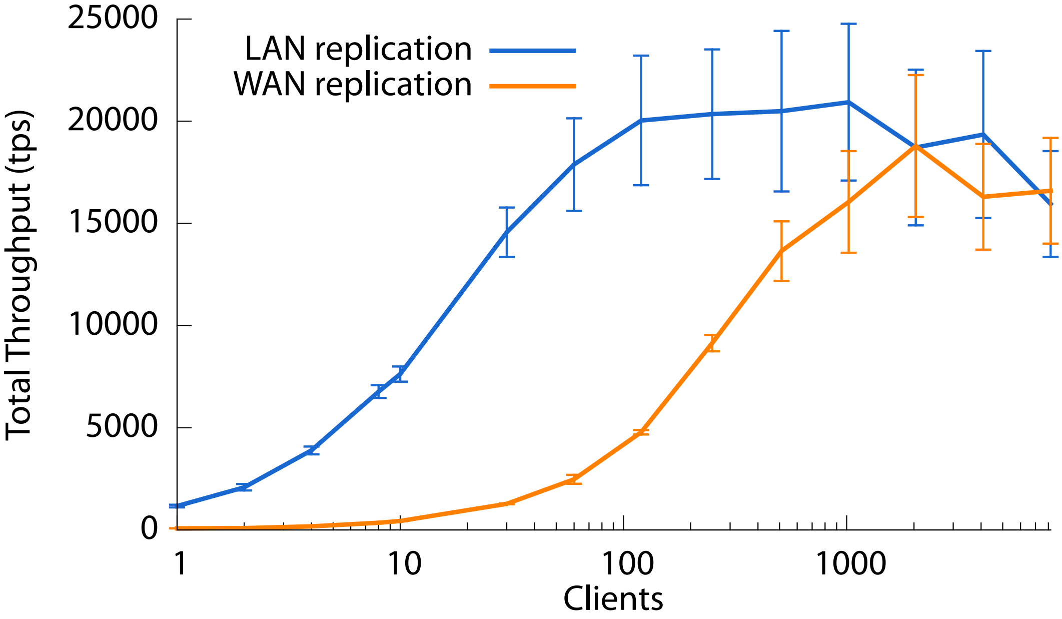

Figure 6.4: Total throughput with increasing client load for a two-repository system with 100% independent transactions, for both local-area and wide-area replication configurations.

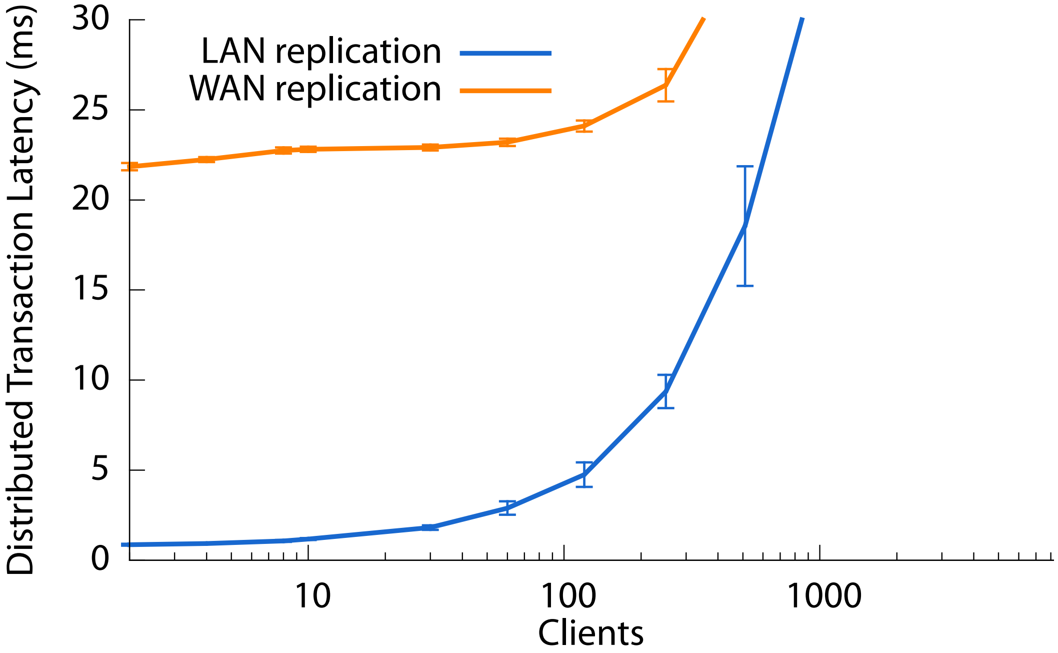

Figure 6.5: Per-transaction latency for a two-repository system with 100% independent transactions, with increasing client load, for both local-area and wide-area replication configurations.

compared to single-repository transactions, and this figure closely tracks Figure 6.3. Latency for the two types of transactions are so similar since distributed transactions require only a single additional one-way message delay, and the transaction participants are located on a single LAN. We examine latency for distributed transactions in more detail in the following sections.

6.3 SCALABILITY

We illustrate the scalability of Granola in Figure 6.6, which shows the total system throughput with respect to the number of repositories. Clients issue each request to a random repository, with between 0% and 10% of requests issued as distributed two-repository independent transactions. These figures show a local-area replication configuration. Configurations with 10 ms one-way delay between repositories and between replicas gave

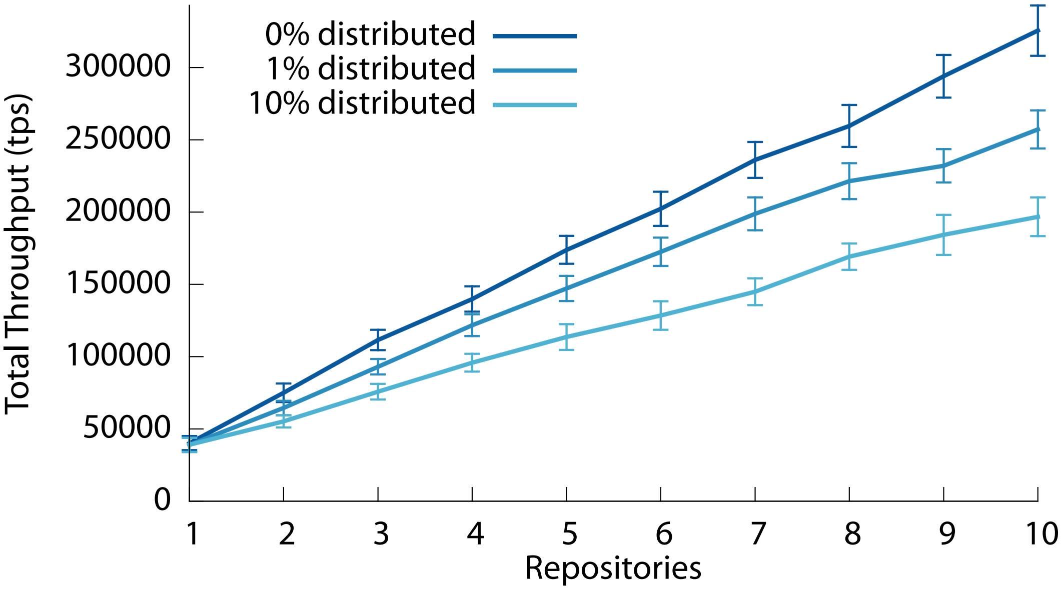

Figure 6.6: Total system throughput with a varying number of repositories, for workloads comprised of a varying fraction of independent transactions. Each distributed transaction is issued to two repositories, with a single master.

equivalent throughput but required significantly more clients to load the system, due to higher request latency.

We also present these results in terms of per-repository throughput in Figure 6.7, which shows how many transactions each individual repository is processing for a given system size.

Throughput for single-repository transactions scales well from two repositories onwards. There is a slight drop in per-repository throughput when transitioning from one repository to multiple repositories, due to the additional scheduling overhead and CPU load from timestamp blocking. Blocking of a given single-repository transaction can occur if the transaction request arrives from a client with a `highTS` value higher than the clock value at the repository, in which case it may be scheduled behind subsequent transactions at the repository. Timestamp blocking of single-repository transactions does not ordinarily occur in practice, since the delay between subsequent requests from a given client is usually significantly higher than the clock skew between repositories. In these experiments we colocate clients

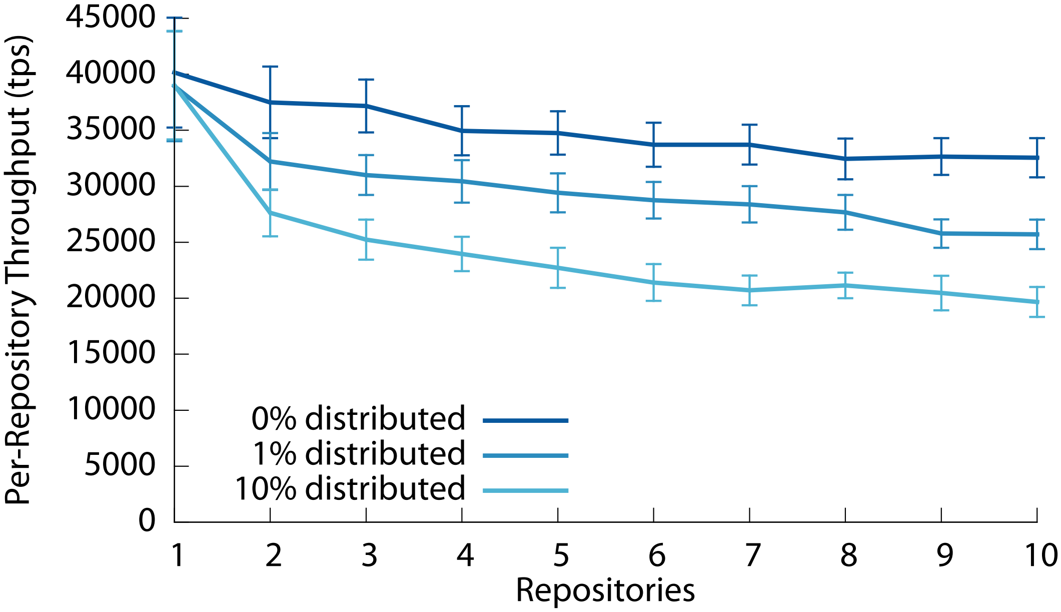

Figure 6.7: Per-repository throughput with a varying number of total repositories, for workloads comprised of a varying fraction of independent transactions. Each distributed transaction is issued to two repositories, with a single master.

on the same LAN as the repositories, however, hence network delay is low compared to clock skew between repositories. Per-repository throughput stabilizes as system size increases, as each individual machine has a lower impact on average clock skew.

Throughput for distributed transactions also scales, but is lower than for single-repository transactions; this is due to the additional overhead for coordinating distributed transactions, as well as the fact that each distributed transaction involves executing an operation on two repositories rather than just one, and also due to the potential for blocking when the `finalTS` value for a transaction significantly exceeds the `proposedTS` value. Distributed-transaction latency was found to be consistently twice the latency of single-repository transactions due to additional communication delay; this leads to correspondingly lower per-client throughput for distributed transactions. We examine distributed transactions in more detail in the subsequent section.

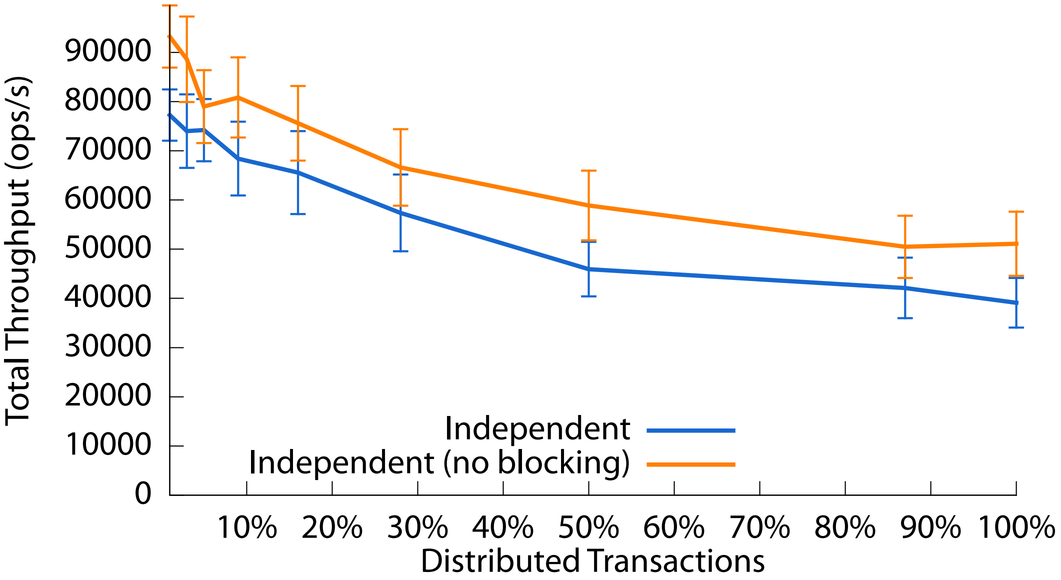

Figure 6.8: Two-repository throughput in *operations* per second, with an increasing proportion of distributed transactions. We examine topologies where the two repositories are located on the same LAN, and when they are separated by a 10 ms one-way network delay.

6.4 DISTRIBUTED TRANSACTION TOLERANCE

The previous section examined throughput for 1% and 10% rates of distributed transactions. It is also interesting, however, to examine performance with much higher rates of distributed transactions. Figure 6.8 shows the impact on throughput of a proportion of distributed transactions, varied between 0% and 100%. We use a topology with two repositories on a local-area configuration. This figure shows throughput for independent transactions in timestamp mode, along with a version of the protocol that was modified to have no timestamp blocking. This non-blocking version does not respect timestamp dependencies and executes transactions immediately once votes are received; the protocol thus does not provide consistency, but serves as a point of comparison in evaluating the overhead of a Granola's timestamp protocol.

The units for throughput in Figure 6.8 are in *operations* per second, where each distributed transaction consists of two operations, one on each repository. This captures the notion that each distributed transaction entails work on two separate repositories, and hence incurs twice the execution cost.

An optimal coordination scheme, one that involves no execution cost, would exhibit a constant throughput in Figure 6.8. Granola achieves lower throughput than this optimal level, due to the additional overhead involved in communication and processing for distributed transactions. We also observe a 10–20% reduction in throughput due to the presence of timestamp blocking. As discussed in the previous section, timestamp blocking is more prevalent in our microbenchmarks than in a deployment with wide-area delay between clients and repositories. The computational overhead of buffering and rescheduling transactions at the repository also contributes to a throughput reduction, since the cost of actually executing transactions in our microbenchmarks is comparatively low. We examine the throughput of independent transactions on a more realistic workload in Section 6.7.

6.5 LOCKING

One of the key contributions of the Granola protocol is the ability to run independent transactions without locking. This section evaluates the overhead of locking, and the performance benefit from avoiding this overhead in Granola.

Locking adds overhead in two key ways:

- The execution cost of acquiring locks and recording undo logs.
- Wasted work from having to retry transactions that abort due to lock conflicts.

We examine both these downsides separately in the following two sections.

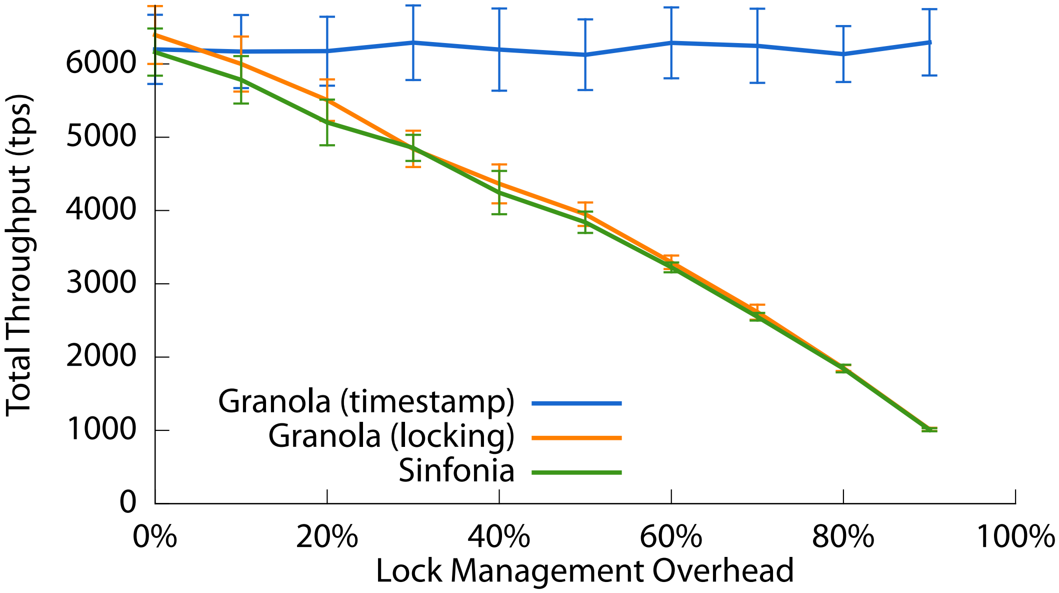

Figure 6.9: Two-repository throughput for a workload comprised of 100% distributed transactions with no lock contention, as a function of cost of managing locks. The cost of locking is expressed as a fraction of total CPU time to execute the transaction. We examine Granola in both timestamp mode and locking mode, as compared to the Sinfonia protocol.

6.5.1 LOCK MANAGEMENT

Figure 6.9 shows the throughput of Granola and Sinfonia on a two-repository topology as a function of lock management cost. We examine a workload composed entirely of distributed transactions, and run Granola in both timestamp mode and locking mode. This experiment measures lock overhead in the absence of lock contention; contention is examined in the subsequent section, and the combined effects of lock management and contention are examined in our macrobenchmarks.

Lock management overhead represents the cost to acquire and release locks, along with the cost of managing undo logs, and is expressed as a percentage of the CPU load at the server application. This CPU load does not include any processing within the repository, but rather just the cost of the work done within the server application in response to Granola upcalls. This experiment includes busy-work at the server application to simulate transaction execution; we set the amount of work to be approximately equal to the cost of executing a TPC-C `new_order` transaction, as seen in Figure 6.17

As expected, throughput for Granola in timestamp mode is independent of lock management overhead, since no locking is used in this mode. Throughput for both Sinfonia and Granola in locking mode drop as lock overhead increases. Typical values for lock management cost in online transaction processing workloads are in the vicinity of 30–40% of total CPU load [30, 31]. At this level of lock management overhead Granola gives 25–50% higher throughput than the lock-based protocols, even in the absence of lock contention.

Sinfonia has slightly lower performance in this benchmark, due to the overhead of communicating with a dedicated master to coordinate each transaction. We deployed the Sinfonia master on a powerful machine to avoid the single master being a CPU bottleneck; Sinfonia throughput was 20% lower when the master was run on the same class of machine as the repositories.

6.5.2 LOCK CONTENTION

We investigate the impact of lock *contention* in Figure 6.10, on a two-repository topology with 100% distributed transactions. We control the lock conflict rate by having transactions modify either a private counter, or a shared counter that conflicts with other transactions. This experiment examines lock contention in isolation, and we tailor our application such that negligible lock management is performed, and no undo logs are recorded.

Throughput for timestamp mode is unaffected by contention, since it does not involve locking. Throughput for both Sinfonia and Granola in locking mode deteriorate fairly rapidly at high lock conflict rates, due to fact that each transaction must be retried multiple times before being successfully committed (note the logarithmic x-axis in the figure). At 100% contention, where transactions must be processed one-at-a-time, Sinfonia and Granola in locking mode taper to under 1,000 tps; we observed a latency of approx-

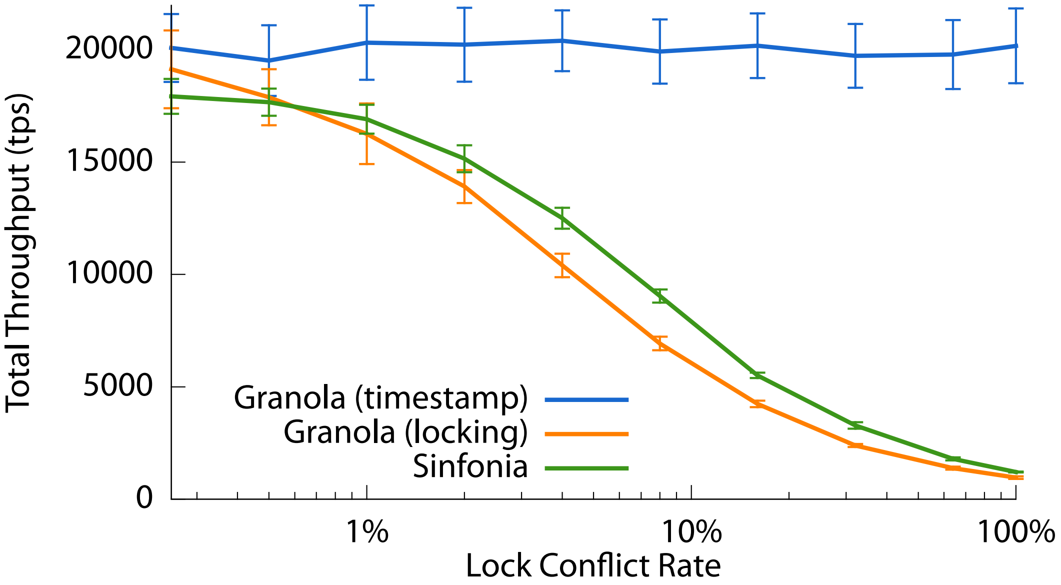

Figure 6.10: Two-repository throughput for a workload comprised of 100% distributed transactions, as a function of the lock contention rate. We examine Granola in both timestamp mode and locking mode, as compared to the Sinfonia protocol.

imately 1 ms for successfully-executed transactions in this configuration, which correlates with the overall throughput of approximately 1,000 tps when transactions are processed serially.

The use of a single master to coordinate transactions in our Sinfonia implementation ensures that each repository receives distributed transactions in the same order. This can result in slightly higher throughput in this benchmark, since each repository sees the same sequence of transactions, avoiding contention due to conflicting transaction orderings. We present an extension to use master nodes in Granola in Appendix C. In practice the performance difference is minimal, since the presence of single-repository transactions or transactions with other participants results in each repository seeing a different sequence of transactions.

Sinfonia only presents a consistent transaction ordering to the repositories when there is a single master node, which is not the case when scaling to larger system sizes [10]. Unlike in our other experiments, the single master is a bottleneck at high transaction throughput in Figure 6.10, leading to lowered maximum throughput at low conflict rates. Multiple master machines would thus be required to fully-utilize a larger topology at high rates of distributed transactions.

6.6 TRANSITIONING BETWEEN MODES

Granola provides low overhead for all transaction classes, but it is optimized specifically for independent distributed transactions. Independent transactions are given special support only in timestamp mode, however; when a repository is in locking mode, it processes all distributed transactions using the coordinated transaction protocol. This section evaluates how much time a repository spends in timestamp mode, and examines the cost of switching between modes.

6.6.1 TRANSITIONING TO LOCKING MODE

The transition from timestamp mode to locking mode involves acquiring locks on all currently-active independent distributed transactions, as described in Section 4.3.6. Locking mode cannot commence until locks have been acquired on all such transactions, and hence may be delayed if there is a lock conflict.

We examine the likelihood of the transition being blocked by using a theoretical model. We define the following notation for workload-specific parameters:

λ_{ind}: The average arrival rate of independent distributed transactions at a given repository.

W_{ind}: The average length of time an independent distributed transaction is active at a repository. For the transition to locking mode, the relevant period of time is from when a repository first sends a vote for the transaction till when all votes have been received.

C_{ind}: The probability that two given independent transactions will conflict on a lock, when processed using the locking mode protocol. We do not distinguish between shared read locks and exclusive write locks in this analysis.

From Little's Law [40], we know that the average number of active independent transactions at a repository at a given point of time, L_{ind} is given by:

$$L_{ind} = \lambda_{ind} W_{ind}$$

The probability that a pair transactions *don't* conflict is $1 - C_{ind}$, and the probability of no conflicts for a set of n transactions is $(1 - C_{ind})^n$. Hence the probability, B_{coord}, that there will be at least one lock conflict, and the repository will have to block when transitioning to locking mode, is given by:

$$B_{coord} = 1 - (1 - C_{ind})^{\lambda_{ind} W_{ind}}$$

When there are no lock conflicts the repository can enter locking mode immediately, otherwise it must wait until it can acquire locks on the remaining independent transactions, or until all independent transactions are complete. An upper-bound on the expected delay a repository will incur from blocking, D_{coord}, is thus given by:

$$D_{coord} \leq B_{coord} W_{ind}$$

since even the most recent active independent transaction will complete in less than W_{ind} time, in the absence of failures.

The probability C_{ind} is expected to be low in practice, since a large fraction of independent transactions are read-only, and thus only acquire shared read locks.

Figure 6.11 plots the expected locking mode transition delay, D_{coord}, as a function of lock conflict rate. We set the processing delay for independent transactions, W_{ind}, to 10 ms, based on our measurements on a local-area deployment, and vary the transaction load up to a maximal throughput of 10,000 tps.

Both axes on this figure are logarithmic. Transition delay is very low for low conflict rates or when the independent transaction request rate is

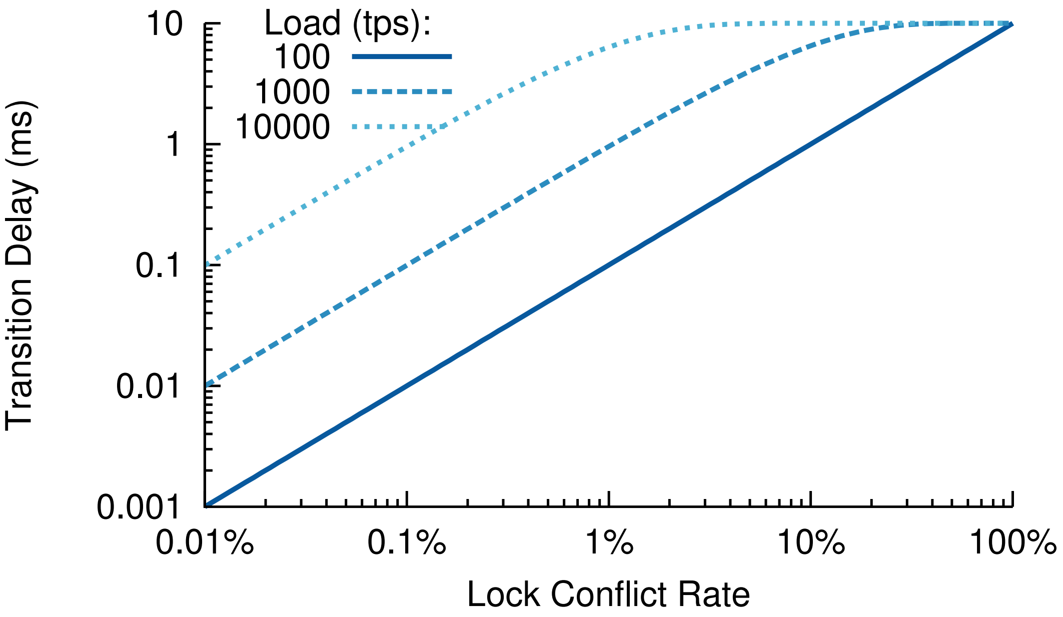

Figure 6.11: Expected delay when transitioning to locking mode (D_{coord}), for a given lock conflict rate (C_{ind}) and independent transaction load (λ_{ind}). Transaction processing time (W_{ind})is set to 10 ms.

low, but approaches a maximum delay of 10 ms. At this maximum value the repository must wait for all previous independent transactions to finish executing before transitioning.

We also examine these results for a varying network delay between transaction participants, in Figure 6.12. The protocol for transitioning to locking mode only includes transactions for which a vote has been sent but the final timestamp has not yet been assigned, hence W_{ind} is is approximated by the one-way network delay between participants.

An increase in network delay leads to an increase in both the probability of a lock conflict when transitioning, and also the time required to wait when blocking on a conflict. Transition delay is nonetheless low for low lock conflict rates, even at high network delays. At the limit all curves approach the line $D_{coord} = W_{ind}$, where the repository must always wait for the independent transactions to complete before transitioning.

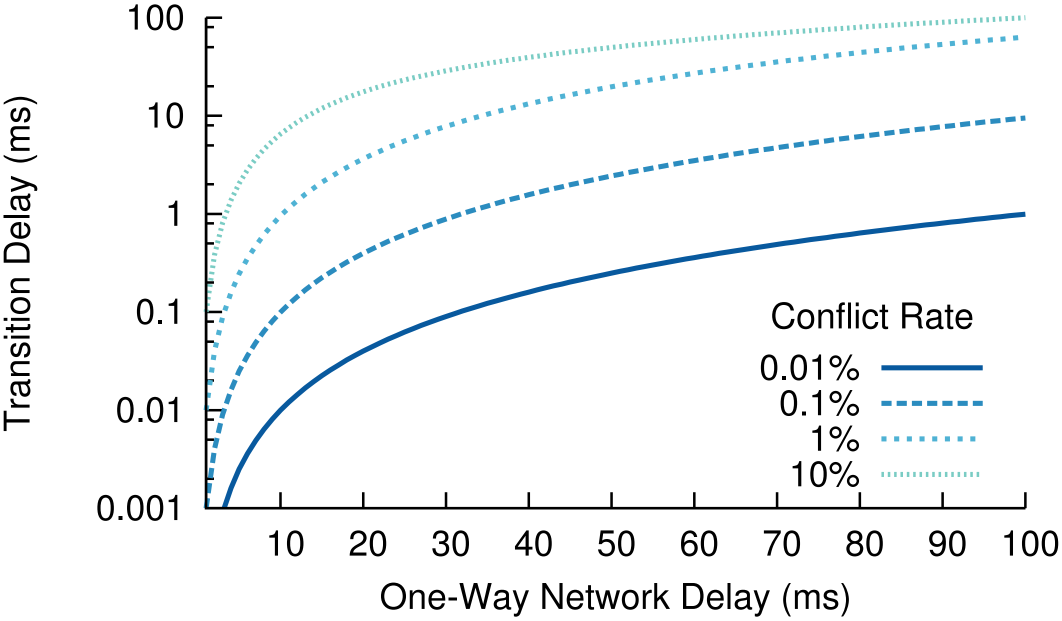

Figure 6.12: Expected delay when transitioning to locking mode (D_{coord}), for a given lock conflict rate (C_{ind}). The one-way network delay between transaction participants approximates the time an independent transaction is active at each repository (W_{ind}). Independent transaction load (λ_{ind}) is set at 1,000 tps.

6.6.2 TRANSITIONING TO TIMESTAMP MODE

There are two challenges involved in transitioning out of locking mode. The first is that a transition can only occur once there are no active coordinated transactions at the repository. The second is that even once all coordinated transactions have completed, there may still be active independent transactions that were processed using the locking mode protocol.

This section examines the likelihood that a repository is processing a coordinated transaction at a given point in time, both theoretically and experimentally. Our experiments assume that any active independent transactions are immediately aborted and retried when transitioning to timestamp mode, to facilitate a rapid transition, as described in Section 4.3.6. We examine alternative mechanisms for handling independent transactions that were processed using the locking mode protocol in Appendix B; we find that the simple protocol presented in Section 4.3.6 is the best option.

Theoretical Model

We define our model in terms of the following parameters:

$\boldsymbol{\lambda_{dist}}$**:** The average arrival rate of distributed transactions at the repository, for both independent and coordinated transactions.

$\boldsymbol{W_{dist}}$**:** The average length of time a distributed transaction is active at a repository, for both independent and coordinated transactions. For the transition to timestamp mode, the relevant period of time includes the duration for which a repository holds locks for the transaction, including the logging and voting phases.

$\boldsymbol{P_{coord}}$**:** The fraction of the workload that is comprised of coordinated transactions.

As in Section 6.6.1, the expected number of concurrent distributed transactions at a repository at a given point in time, L_{dist}, is given by Little's Law as:

$$L_{dist} = \lambda_{dist} W_{dist}$$

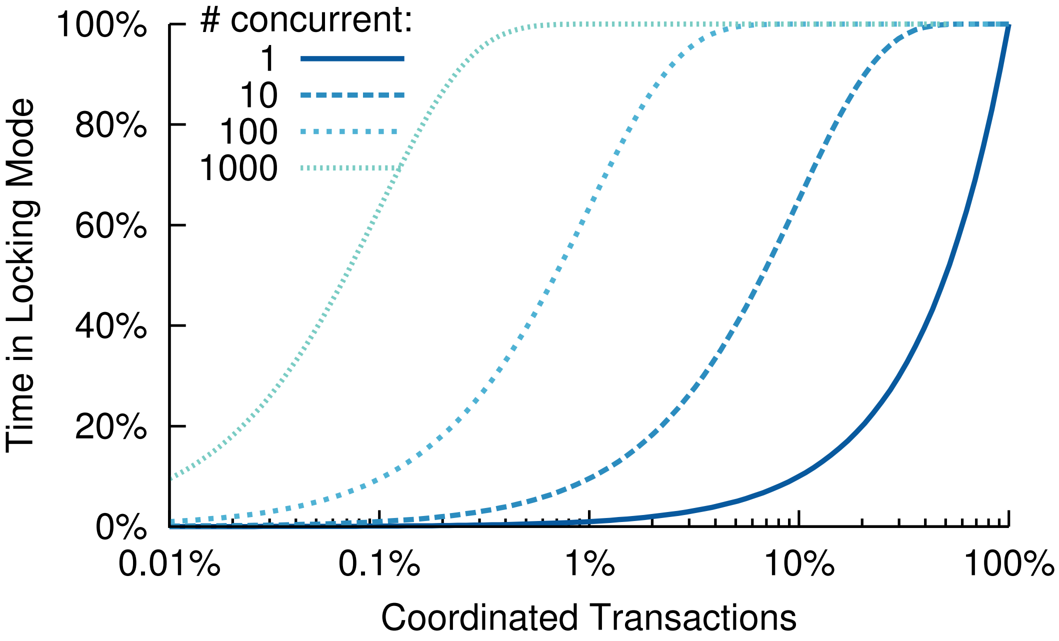

Figure 6.13: Expected time spent in locking mode at a repository (T_{coord}), as a function of the fraction of coordinated transactions (P_{coord}) and number of concurrent distributed transactions (L_{dist}).

The likelihood of a given transaction being independent is $1 - P_{coord}$. Assuming that transactions are *independent and identically distributed*, the expected likelihood of there being no active coordinated transactions is thus $(1 - P_{coord})^{L_{dist}}$, and the likelihood, T_{coord}, of there being at least one coordinated transaction at a repository is given by:

$$T_{coord} = 1 - (1 - P_{coord})^{L_{dist}}$$

We plot this function in Figure 6.13. We see that at high rates of concurrent distributed transactions, even a small fraction of coordinated transactions can lead to a repository remaining almost permanently in locking mode. At lower rates of concurrent transactions these results appear more reasonable; with 100 concurrent transactions, a repository can sustain a workload comprised of 1% coordinated transactions, while still benefiting from being in timestamp mode for nearly half the time.

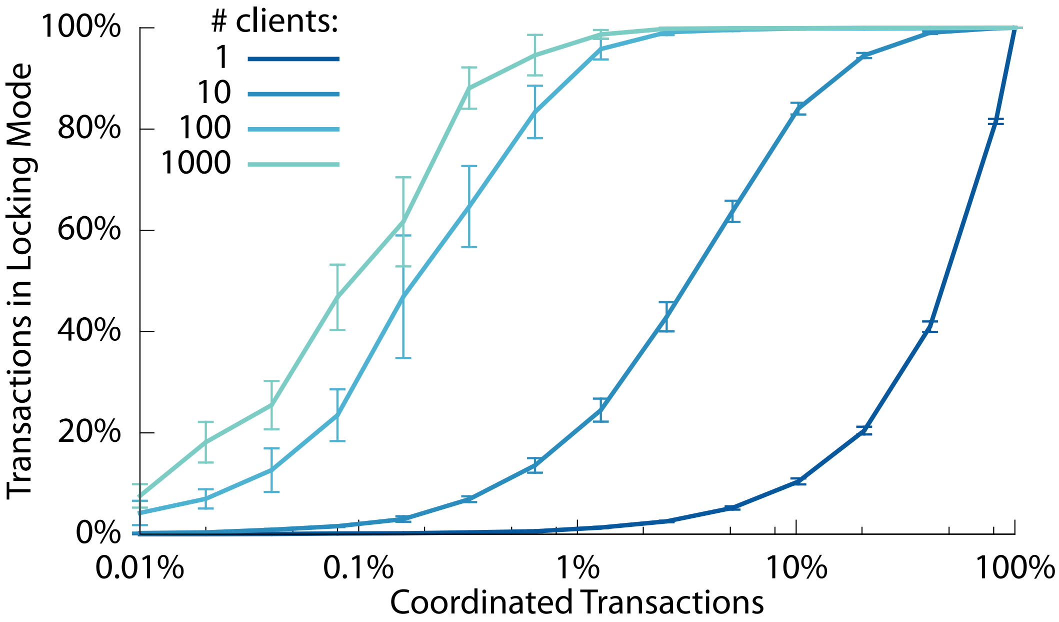

Figure 6.14: Fraction of transactions processed in locking mode, for workloads composed of a varying fraction coordinated transactions. We examine a varying number of concurrent client connections. 100 clients are sufficient to fully-load the repositories in this benchmark.

Experimental Results

We further examine the time spent in locking mode in Figure 6.14, which records a real workload on a local-area configuration. This workload is comprised entirely of single-repository transactions and coordinated distributed transactions. We vary the number of simultaneous clients to alter the load on the repositories, and hence the number of simultaneous transactions.

We also monitored throughput and latency in this benchmark. 100 clients were sufficient to fully-load the repositories, yielding a throughput of approximately 40,000 tps, at 2.5 ms latency for single-repository transactions and 10 ms latency for coordinated transactions. Increasing client load to 1,000 concurrent clients did not yield a significant increase in throughput, but triggered a 10× increase in latency as a result of the increase in queue size. The 100-client line in this figure thus represents the optimal value for maximum throughput at minimal latency.

Granola does not spend a significant fraction of time in timestamp mode in this experiment, when handling workloads comprised of more than 1% coordinated transactions. Granola hence provides best performance when operating on workloads with a particularly small proportion of coordinated transactions. We note that the time spent in locking mode is specific to each individual repository; if a repository never receives a coordinated transaction, it never needs to transition to locking mode under normal operation, even if it participates in transactions with repositories that are in locking mode.

6.7 TRANSACTION PROCESSING BENCHMARK

We evaluate performance on a large application using the TPC-C transaction processing benchmark [8]. This benchmark models a large order-processing workload, with complex queries distributed across multiple repositories. The TPC-C schema contains 9 tables with a total of 92 columns, 8 primary keys, and 9 foreign keys. The workload consists of 5 types of transactions that operate on this schema, including both single-repository transactions and distributed transactions, some of which are read-only and some of which modify the database. Our implementation stores the TPC-C dataset in-memory and executes transactions as single-round stored procedures.

We used the C++ implementation of TPC-C from the H-Store project [50] for our client and server application code. As is common in research use, this implementation does not strictly adhere to the TPC-C specification. In particular, each client submits a subsequent transaction as soon as it receives the final response for the previous transaction, instead of waiting for a specified “think time”. This modification is used to drive sufficient client load with a limited system size.

The particular TPC-C codebase that we used was designed for a single node deployment and had no explicit support for distributed transactions. By interposing the Granola platform between the TPC-C client and server, we were able to build a scalable distributed database with minimal code changes; code modifications were constrained to calling the Java client proxy implementation from the C++ client using JNI, responding to transaction requests from the repository code, and translating warehouse numbers to repository IDs.

We adopt the data partitioning strategy proposed in H-Store [50]. This partitioning ensures that all transactions can be expressed as either single-repository or independent transactions. We are thus able to disable locking and undo logging in the codebase when running the benchmark with independent transactions.

We examine TPC-C performance for Sinfonia and for Granola, which runs TPC-C exclusively using independent transactions. As a means of comparison we also run Granola manually set to stay in locking mode, to examine the performance impact of locking.

6.7.1 SCALABILITY

We examine scalability in Figure 6.15. This experiment uses a single TPC-C warehouse per repository, and increases the number of clients to maximize throughput. 10.7% of transactions in this benchmark are issued to multiple repositories.

All systems exhibit the same throughput in a single-repository configuration, since they all have similar overhead in the absence of locking. Throughput drops for the lock-based protocols on multiple nodes, however. The TPC-C implementation is highly optimized and executes transactions efficiently, hence the lock overhead imposes a significant relative penalty; the overhead of locking and allocating undo records in these experiments was approximately equal to the cost of executing each operation, in line with similar measurements on the same workload [32]. Throughput reduction is also heavily impacted by the cost of retrying transactions that conflict on locks. Sinfonia sees a slight performance hit compared to Granola in locking mode, due to the additional overhead of communicating with the master.

We show the latency for distributed transactions on this benchmark in Figure 6.16. This figure shows the average latency for transactions that complete successfully, and does not include aborted transactions. Latency is relatively constant with respect to system size, since it is primarily a

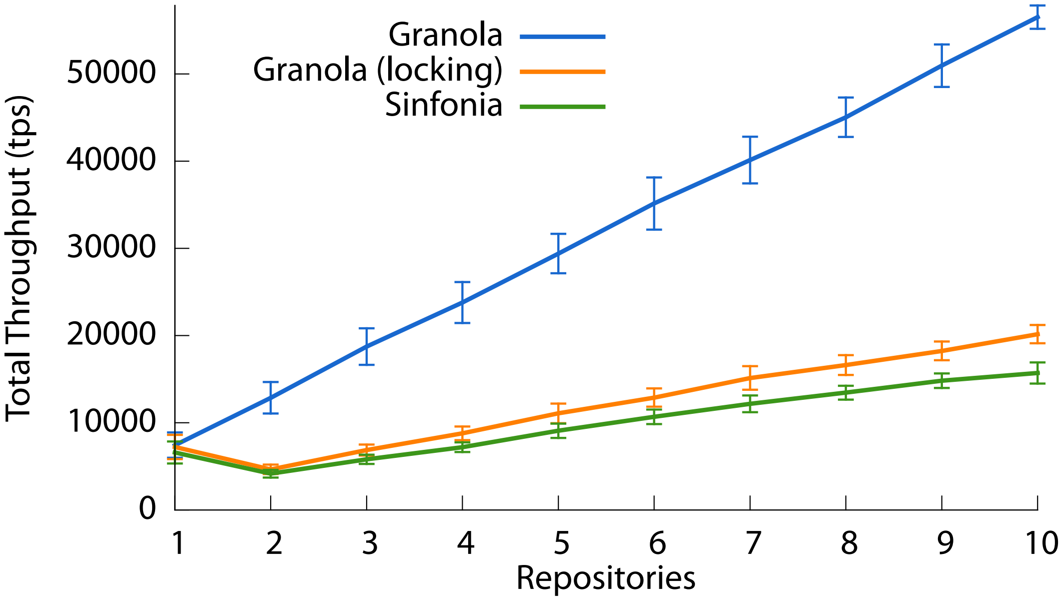

Figure 6.15: Total system throughput on the TPC-C benchmark with an increasing number of repositories. We examine both the Granola and Sinfonia protocols, along with a version of Granola that always runs in locking mode.

function of communication delay and execution time. Latency for Granola in locking mode is higher than for Granola in timestamp mode, since the repository must wait additional time for locking to be performed, and the transaction queue at the repository is longer due to the lower throughput. Sinfonia has approximately 60% higher latency than Granola, since the Sinfonia protocol involves 5 message delays rather than 3 in Granola. We don't include a data-point for a configuration with only one repository, since there are no distributed transactions in such a topology.

6.7.2 DISTRIBUTED TRANSACTION TOLERANCE

We further examine coordination overhead by modifying TPC-C to vary the proportion of distributed transactions [32]. We alter the workload to be composed entirely of `new_order` transactions, which are the most common source of distributed transactions in the TPC-C benchmark, and the source of most execution time. `new_order` transactions may be either single-

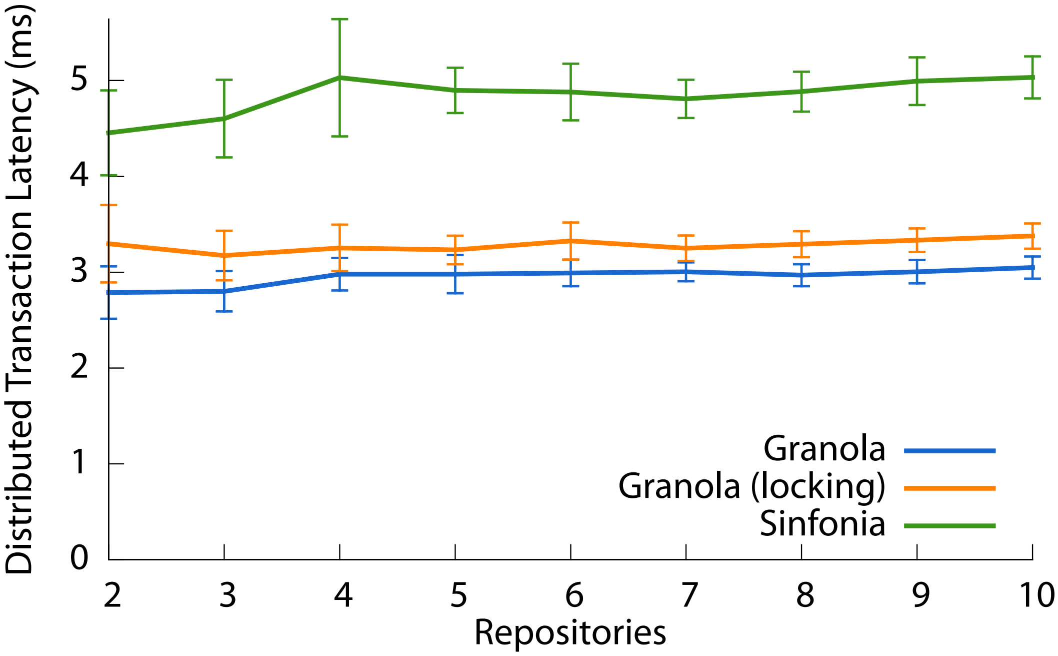

Figure 6.16: Per-transaction latency for distributed transaction on the TPC-C benchmark, with an increasing number of repositories. We examine both the Granola and Sinfonia protocols, along with a version of Granola that always runs in locking mode.

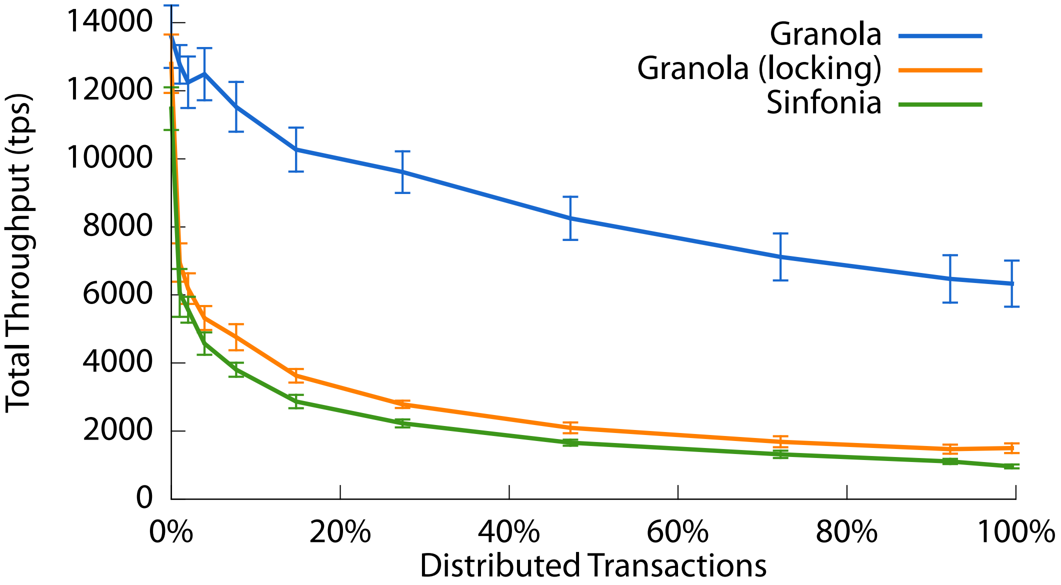

Figure 6.17: Two-repository throughput on a version of the TPC-C benchmark that issues only `new_order` requests, with a variable rate of distributed transactions. We examine both the Granola and Sinfonia protocols, along with a version of Granola that uses only coordinated transactions.

repository or distributed. We modify the workload parameters to manually adjust the likelihood that an item in the order will come from a remote warehouse, hence adjusting the likelihood that a `new_order` transaction will be a distributed transaction [32].

We show the throughput for our modified benchmark in Figure 6.17, for a two warehouse configuration on two repositories, with a varying fraction of distributed transactions.

Granola achieves better resilience to distributed transactions in this benchmark than in our microbenchmarks, since overhead in TPC-C is dominated by transaction execution costs rather than protocol effects. Throughput for Granola in timestamp mode decreases by approximately 50% when moving from 0% to 100% distributed transactions. This represents a very low performance penalty for distributed transactions, since each distributed transaction involves execution on two repositories instead of one, and hence there is a natural reduction in throughput even for optimal transaction coordination schemes.

Granola achieves lower throughput when in locking mode, and scales less optimally to a large proportion of distributed transactions. As in previous benchmarks, the lock-based protocols suffer a performance penalty due to both lock management overhead and aborts due to lock conflicts. Sinfonia offers lower absolute throughput than Granola in locking mode on this benchmark, due to the additional overhead of communicating with the master, but encounters the same relative performance penalty from locking and conflicts.

6.7.3 LATENCY TRADE-OFF

We finally examine the throughput/latency trade-offs for the three protocols, for a six-repository topology running the TPC-C benchmark. We increase client load for the three protocols, and measure the corresponding average total throughput and the latency for distributed transactions. These results are displayed in Figure 6.18, plotting distributed transaction latency as a function of throughput.

These results show that the three protocols offer relatively constant latency up until the point of maximum throughput, where congestion collapse sets in. Again, Sinfonia has approximately 60% higher latency under stable conditions, due to the additional communication phases. Latency for Sinfonia and Granola in locking mode degrade very rapidly once maximum throughput is reached, leading to a reduction in throughput and rapidly increasing latency, owing to the large number of aborted transactions when the system is fully saturated. Performance for Granola in timestamp mode degrades far more gracefully, since transactions are not aborted, and congestion collapse simply results in additional queuing of transactions.

We varied client load up to a total of 516 simultaneous transactions in this experiment. At this maximum load (outside the range displayed on the graph), Granola in timestamp mode had a latency of under 20 ms, while maintaining a throughput very close to the maximum capacity. At this same level of excess load, Granola in locking mode resulted in an average

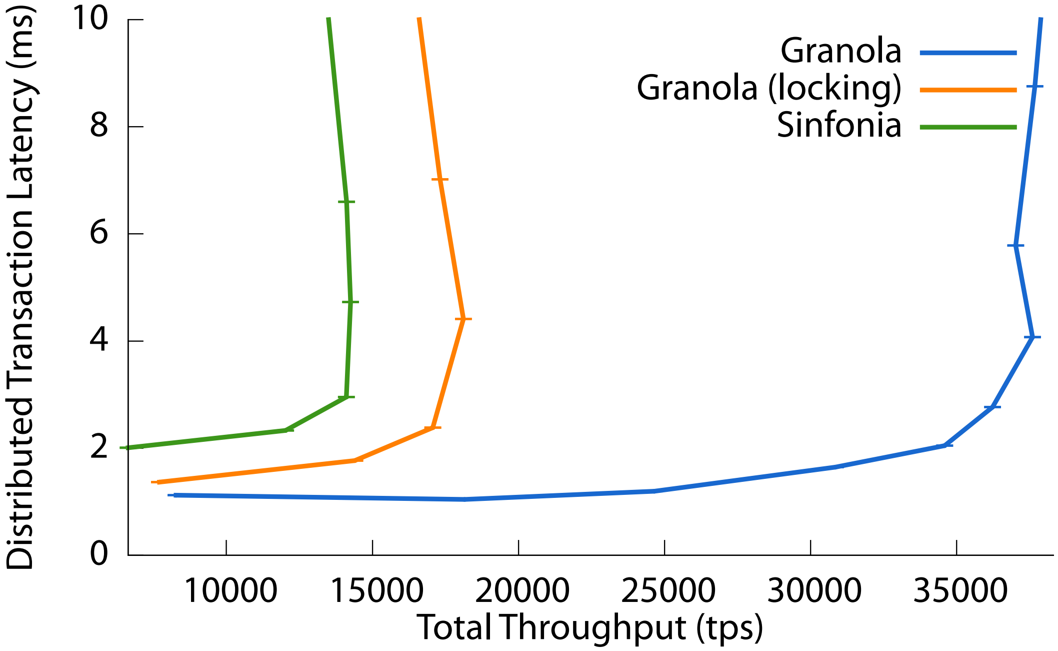

Figure 6.18: Latency for distributed transactions as a function of system throughput, showing congestion collapse for Granola, Granola in locking mode, and Sinfonia, on a six-node topology running the TPC-C benchmark.

of 420 ms latency, and Sinfonia resulted in an of average 500 ms latency, each with over a 50% reduction in throughput. The use of independent transactions thus not only achieves higher throughput at lower latency, but also avoids a collapse in performance when client load exceeds system capacity.

7 EXTENSIONS AND OPTIMIZATIONS

The previous chapters described the core components of the Granola protocol. This chapter presents additional components necessary in a practical deployment, including naming and data migration, along with extensions and optimizations such as supporting reads at backup replicas, cache nodes, implicit votes, and interactive transactions.

7.1 NAMING AND RECONFIGURATION

Granola presents the client application with the abstraction of individual repository nodes, addressable by RID. The client proxy manages the mapping from each RID to the IP addresses for the repository, on behalf of the client application, including tracking the current primary and backup replicas for each repository. This mapping is provided by a *membership service*, which tracks the membership of all repository nodes in the system.

Existing techniques for tracking system membership can be used to implement the Granola membership service [11, 17]. We developed our own membership management service called Census [21], which is specifically designed to provide a consistent view of system membership, while scaling to very large system sizes.

This section instead focuses on the mapping from application-level names to RIDs. This mapping is the responsibility of the application built on top of the Granola platform. Since Granola supports general operations and does not interpret the operations within a transaction, it cannot directly determine which repositories are needed to run a given transaction. Granola provides a *name service*, however, which stores this mapping on behalf of the application, and allows the mapping to be changed while the system is running, while maintaining consistency.

7.1.1 OVERVIEW

The application is responsible for deciding how to divide data among the repositories. The mapping from application-level names to RIDs could be stored locally at each client if the schema and set of repositories did not change. The mapping will likely change over time in a long-lived system however, and requires a mechanism for retrieving and updating the mapping.

Granola provides a name service that runs on a particular repository and maps application-level names to RIDs. Each client caches a portion of the mapping and uses it to decide which repositories are involved in a given transaction; the client will refresh its mapping from the name service if discovers that it is stale, or if it is missing information.

Responsibility for application-level names can be transferred between repositories, through a process known as *reconfiguration*. Each configuration of the system is identified by an epoch number, initially 0, which is incremented every time a reconfiguration occurs. Epoch numbers are used to identify the current version of the mapping, and ensure that all clients and repositories are up to date when processing a transaction. Reconfiguration transfers responsibility for a set of application-level names, but it does not migrate the corresponding data. Migrating data is the responsibility of the server application, and can be performed on-demand after reconfiguration has taken place.

We discuss our name service and the reconfiguration protocol in more detail in the following sections.

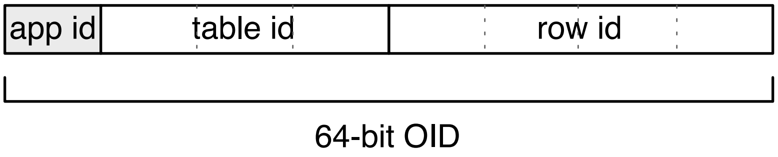

Figure 7.1: Typical object identifier format for a database application, identifying the application/database, table, and row. This format is used internally by the application, and is not interpreted by the Granola protocol.

7.1.2 NAMING

Granola provides 64-bit opaque identifiers to serve as application-level names; we term these *object identifiers* (OIDs). The application may use this OID space to identify individual data items and assign them to repositories as it chooses.

An OID in the namespace could correspond to a warehouse ID in an OLTP-style application, or an inode in a filesystem. One typical naming scheme for a database application would be to reserve the upper bits of the OID to identify the table name, and the lower bits to identify column or row indices within that table; Figure 7.1 illustrates this format, including 8 reserved bits to identify the particular database or application. A scheme like this allows the application to partition the database between repositories, based on tables or columns/rows, while maintaining a contiguous range of OIDs at each repository.

A given transaction may involve OIDs that map to multiple RIDs, in which case it will be issued as a distributed transaction to these repositories. A replicated data partition can also be represented by an OID that maps to multiple RIDs.

7.1.3 NAME SERVICE

One repository in the system is denoted as the *name service*, and stores the OID mappings for all other repositories. The name service has a well-known RID, and allows clients to retrieve or update each mapping as required. The name service may itself be partitioned over a set of repositories if necessary;

load on the name service is typically light, however, since clients keep a local cache of OID mappings.

The name service does not store the IP addresses of the replicas that comprise each repository, or the identity of each primary replica, since this is the responsibility of the *membership service* as described previously. This information is retrieved by the client proxy, and is not the responsibility of the client application.

The name service stores the mapping from OID to RIDs, for all OIDs in the system. This mapping is implemented using range structures to avoid storing individual entries for a large contiguous range of OIDs. The name service also records the epoch number corresponding to the most recent update made to the mapping.

The name service supports operations to retrieve or update the current mapping for a set of OIDs. Clients issue a single-repository transaction request to the name service to retrieve the latest mapping for a range of OIDs, and are sent the latest epoch number along with this mapping. Updates to the mapping are performed using a coordinated distributed transaction, as described in the following section.

7.1.4 RECONFIGURATION

Reconfiguration is used to modify the OID→RID mapping, and must execute atomically at the name service and the set of repositories where the mapping is changing; this includes both the source and the destination for OIDs that are migrating between repositories.

The name service increments its epoch number during every reconfiguration. Each repository maintains a record of the epoch number for the most recent reconfiguration it was involved in. Each client also has a record of the epoch number corresponding to its current cached copy of the mapping.

The client includes its epoch number in each transaction request. If the repository that receives this request has a higher epoch number than was provided by the client, it rejects the request, since the client may have issued the transaction based on a stale mapping. The client will update its cached mapping by issuing a query to the name service, before retrying the transaction. If the client observed the results of a recent reconfiguration, its epoch number might be *higher* than the epoch number at the repository. The repository can still accept such a transaction, since any reconfiguration the repository doesn't know about cannot involve the mapping at the repository.

A reconfiguration request specifies a set of OIDs that are moving from the source repositories to the destination repositories, and is issued as a coordinated transaction. When a repository receives a reconfiguration request it goes into *reconfiguration mode* and will abort any subsequent transaction using a `CONFLICT` response.

The repository waits to finish executing any transactions that are already in progress, before processing the reconfiguration. The source and destination repositories will vote to commit a reconfiguration request if the client's epoch number is at least as high as the current epoch number at the repository.

The name service attempts to acquire a lock on the OIDs involved in the reconfiguration, and will abort the transaction if it is unable to acquire the locks. If the name service *can* acquire these locks, it advances its epoch number, and sends this number along with its `COMMIT` vote. If the reconfiguration commits, the name service will update its mapping, as will the source and destination repositories. The repositories can then resume processing new transactions, at their new epoch number.

This protocol ensures that the transfer of authority for OIDs executes atomically. It does not handle the migration of physical data between the repositories, however. This transfer can be handled on-demand using existing techniques for live data migration, e.g. [31].

7.2 NON-PRIMARY READS

In the protocol described in Chapters 4 and 5, the primary replica handles all communication with clients and other repositories; the backup replicas only process log messages and execute read-write transactions. This leads to an imbalance in server utilization, however, with the backups replicas being underutilized.

Our timestamp protocol allows us to shift work from the primary replica to the backups, by executing read-only single-repository transactions at a backup replica instead of the primary. Since we expect that workloads will be composed predominantly of read-only single-repository transactions, this can offload a significant fraction of work from the primary, and allow the system to handle a far higher request load.

Each backup replica maintains a `lastTS` timestamp value, corresponding to the timestamp for its most-recently executed transaction, as described in Section 5.2.1 and illustrated in Figure 5.2. Backup replicas do not operate in locking mode, and instead always execute in strict timestamp order. Since the replica has executed all transactions up to `lastTS`, it is up-to-date with respect to all transactions in the serial order up to this point.

The client proxy issues read-only single-repository transaction requests to a backup replica using the same protocol as for transactions issued to the primary. The protocol for handling a ⟨Request, `tid`, `highTS`, `rids`, `ops`, `ro`, `indep`⟩ message at a backup is as follows:

1. The backup first checks that the `rids` set includes only this repository, and that `ro` and `indep` are true. If not, the backup forwards the request to the primary. This step is necessary since a client may issue a non-read-only or distributed transaction to a backup replica if it does not have an up-to-date record of the current primary.

2. The backup then checks if `highTS`≤`lastTS`, for `highTS` in the Request and the `lastTS` value at the backup:

   (a) If `highTS`≤`lastTS`, the backup will have executed any transaction that may have been ahead of the client's transaction in the serial order, and can thus execute the client's transaction based on the current state at the backup. The backup executes the `op` immediately by executing a `run` upcall, and responds to the client with a ⟨Reply, `tid`, `rid`, `lastTS`, `status`, `result`⟩ response, where `status` is `COMMIT`.

(b) Otherwise the backup is unable to process the transaction, since it may not be up-to-date with respect to state the client has observed. It forwards this transaction to the primary replica. It is also possible to instead delay the transaction locally for a short period of time in case the backup catches up to `lastTS`.

The hit-rate at each backup is expected to be high, since the backups do not lag significantly behind the primary in transaction execution. Each backup replica executes transactions at the head of the stable log as soon as they have a final timestamp. It is thus likely that the backup will have executed a given transaction before the client issues a subsequent read-only transaction to the backup.

In the protocol just described, each backup replica functions as a cache node that supports reads in the past [46]. While this does not guarantee *freshness*, the protocol nonetheless guarantees serializability for all operations, despite the fact that a transaction may execute on a state that does not include recent transactions at the primary replica. Serializability is provided by serializing each read-only transaction at the `lastTS` timestamp; since the transaction is read-only, it can be serialized at this timestamp without affecting subsequent transactions in the serial order.

7.3 CACHES

Granola allows the use of dedicated infrastructure caching nodes, which support the execution of single-repository read-only transactions. These nodes offload work from the repositories, and allow the system to scale more readily for workloads that consist of a significant fraction of read-only operations. The protocol for cache nodes operates similarly to the non-primary read protocol just described, and also supports reads in the past without violating serializability.

Each repository is responsible for a set of cache nodes, and each cache node stores data for a given repository. Any node in the repository replica group can periodically send a state update to each cache node, using application state diffs to minimize the amount of data that needs to be communicated. When this cache push is performed, the replica also sends the latest timestamp corresponding to the state. Each cache maintains a record `lastTS` of this latest timestamp.

The location of cache nodes is recorded in the Granola name service, and a client can choose to issue a read-only single-repository transaction to the nearest cache node rather than directly to the repository. As in the previous section, a cache will accept a transaction only if the request is read-only and single-repository, and `highTS`≤`lastTS`, where `highTS` is provided in the client Request message. If these conditions are met, the cache node can execute the transaction based on its current state, and return a Reply message with a timestamp value of `highTS`. If the conditions are not met, the cache responds to the client with a `CONFLICT` response, and the client proxy will retry the request at the repository.

Cache nodes respect serializability, since they only accept a transaction request if the client hasn't observed any transaction that occurred since the last cache push, according to the global serial order.

7.4 IMPLICIT VOTES

The execution of an independent transaction is delayed until it has the lowest timestamp of all concurrent transactions at the repository. We wish to minimize the time that a transaction is delayed behind other transactions, however, especially if these other transactions will eventually be assigned a higher final timestamp and be serialized later in the serial order.

One common source of unnecessary delay can occur when two participants are involved together in a series of independent transactions. Consider two repositories A and B, which assign proposed timestamps to a series of concurrent transactions 1–5, as shown in Table 7.2. Repository A will first receive a vote for transaction 1, and assign it a final timestamp of 10. The repository is not yet able to execute this transaction, however, since transactions 2–5 currently have proposed timestamps lower than 10. In fact, transaction 1 will not have the lowest timestamp until it receives

	`proposedTS`	
Transaction	Repository A	Repository B
1	3	10
2	4	11
3	5	12
4	6	13
5	7	14

Table 7.2: Example of proposed timestamps for a series of independent transactions between two participants.

votes for *all* transactions 2–5. In situations where there is significant clock skew between participants and a high rate of independent transactions, this can lead to excessive buffering of transactions awaiting execution.

We solve this problem by introducing *implicit votes*. If a repository always assigns proposed timestamps in a monotonically increasing order, then a vote of timestamp 10 for transaction 1 implies that any subsequent transaction involving that participant will have a timestamp higher than 10. When a repository receives a vote for a given transaction, it updates its `proposedTS` value for all subsequent transactions involving that participant, to be at least as high as the timestamp vote. This protocol allows a repository to execute a complete transaction ahead of any such subsequent transaction.

Implicit votes introduce two additional requirements to the Granola protocol:

- Each repository must assign timestamps to transactions in a monotonically increasing order. This can be achieved by choosing timestamps as the maximum value of the current clock value, the client-provided `highTS` value, and the most recently assigned timestamp; this is distinct from the protocol presented previously, which chooses the maximum value of the current clock value, the client-provided `highTS`, and the `lastExecTS` value at the repository.

- Each vote message must be delivered in-order. This requirement can be met in a number of ways. We achieve in-order delivery in our implementation by including a sequence number on each vote from a repository to a given participant. If an out-of-order vote is received, the repository ceases processing implicit votes from the participant until in-order votes have been received for all concurrent transactions from that participant. Sequence numbers are logged in each stable log write to provide continuity of the sequence number space despite failures.

7.5 INTERACTIVE TRANSACTIONS

As described in Section 2.1, Granola is designed to support a one-round transaction model, where the client specifies all operations to be run in a single request message. Granola can also be used to support a more traditional interactive transaction model, however, to allow transactions that do not fit within the one-round model.

Operations in Granola can contain arbitrary code, and thus a transaction request can instruct the server application to acquire locks that remain held once the one-round transaction is complete. If the server application supports interactive transactions through the use of `begin transaction` and `end transaction` statements, Granola's one-round transactions can be used to communicate these instructions to the server application.

Single-repository interactive transactions can be expressed as a series of single-repository one-round transactions, whereas distributed interactive transactions can be expressed as a series of single-repository or independent transactions, followed by a coordinated transaction to determine the final commit decision.

The exact method used to support interactive transactions depends on the implementation of the server application, but we outline two common approaches as follows:

Optimistic Concurrency Control Interactive transactions can be implemented using optimistic concurrency control by using single-repository transactions to record transaction read sets and write sets, without modifying the back-ing data store. A coordinated transaction can then be used to check the validity of the read and write sets, and install the writes if these sets were not invalidated throughout the duration of the transaction. Implementing interactive transactions via optimistic concurrency control has the advantage of not requiring any additional locking throughout the duration of the transaction, but does require the management of read sets and write sets.

Lock-based Concurrency Control Lock-based concurrency control can also be used to implement interactive transactions as a series of single-repository transactions, followed by a coordinated transaction to determine the commit decision. In this case, the coordinated transaction must check that none of the locks acquired for the transaction was discarded, and then commit or abort the transaction accordingly.

The additional challenge when using lock-based concurrency control is to ensure that the repository does not agree to commit a concurrent independent transaction at a given timestamp, then discover during execution that the independent transaction conflicts with an interactive transaction. One solution to this problem is to abort and roll back any interactive transaction if a concurrent one-round transaction conflicts with it; the client that issued the interactive transaction would learn about the abort on the subsequent query. The other alternative is to allow the application to instruct the repository to remain in locking mode for the duration of an interactive transaction, which requires minor extensions to our server API.

8 RELATED WORK

There has been a long history of research in transactional distributed storage, including an extensive literature on distributed databases [15, 24, 42]. These original systems provide a rich transaction model, including support for interactive transactions. Granola targets a simpler transaction model, providing *one-round transactions*, but can nonetheless be used to support a wide range of online transaction processing applications [10, 32, 50, 53]. While traditional distributed databases provide a sophisticated data model, they typically do not provide the scalability and availability guarantees offered by Granola and other modern transaction processing systems.

We focus our discussion of related work on more recent approaches to distributed storage, first discussing the range of consistency models provided by modern storage infrastructures. We follow this with a more in-depth analysis of recent work that processes distributed transactions using a deterministic transaction ordering, and thus implements a similar execution model to Granola.

8.1 CONSISTENCY MODELS

Brewer's CAP Theorem [16] recently popularized the idea of the trade-off between consistency, availability, and partition-tolerance in a distributed storage system. While this work sparked a flurry of research in storage systems with weaker consistency guarantees, it has long been known that strongly consistent storage systems may have to sacrifice availability under certain failure scenarios.

Strong consistency not only poses a challenge in terms of availability, however. Strong consistency has also traditionally been associated with two-phase commit [28, 36] and strict two-phase locking [25], and an expectation of inherent limits to system performance and scalability. Granola is designed to provide strong consistency guarantees while avoiding these overheads.

We examine previous approaches to distributed storage with various consistency guarantees in the following sections.

8.1.1 RELAXED CONSISTENCY

Many systems [29, 33, 47, 52] relax consistency guarantees in order to provide increased scalability and resilience to network or hardware partitions. The growth of cloud computing has led to a resurgence in popularity of large-scale storage systems with weaker consistency, typified by Amazon's eventual-consistency Dynamo [23] and many others [3–5, 7, 20]. These systems target high availability and aim to be "always writable", but sacrifice consistency. Applications built atop these platforms must either be tolerant of inconsistent results and the potential loss of updates, or develop their own consistency protocols at the application level. Both cases pose a challenge to application development, and limit the types of systems that can be built on the platform.

Recent eventually-consistent storage systems also typically offer constrained transaction interfaces, such as Dynamo's read/write distributed hash table interface. Granola instead provides support for general operations, simplifying application development.

8.1.2 PER-ROW CONSISTENCY

Systems such as SimpleDB [2] and Bigtable [19] provide consistency within a single row or data partition, but do not provide ACID guarantees between these entities. A significant downside to relaxed-consistency storage systems is the complicated application semantics presented to clients and developers when operating with multiple data items. More recent protocols such as COPS [41] and Walter [49] attempt to simplify application development by providing stronger consistency models: *causal+* and *parallel snapshot isolation* respectively. These models to not prevent consistency anomalies however, and require the developer to reason carefully about the correctness of their application under a given model.

8.1.3 STRONG CONSISTENCY

Megastore [12] represents a departure from the traditional wisdom that it's infeasible for large-scale storage systems to provide strong consistency. Megastore is designed to scale very widely, uses state-machine replication for storage nodes, and offers transactional ACID guarantees. As in SimpleDB [2], Megastore ordinarily provides ACID guarantees within a single entity group, but also supports the use of standard two-phase commit to provide strong consistency between groups. CRAQ [51] primarily targets consistency for single-object updates, but mentions that a two-phase commit protocol could be used to provide multi-object updates. Granola is more heavily optimized for transactions that span multiple partitions, and provides a more general operation model.

Granola is most similar in design to Sinfonia [10]. Sinfonia also supports reliable distributed storage over large numbers of nodes, with strong consistency and atomic operations over multiple storage nodes. Sinfonia also supports a one-round transaction model, but its *minitransactions* express transactions in terms of read, write and predicate sets, whereas Granola supports arbitrary operations and does not require *a priori* knowledge of locksets. Granola also provides one less message delay in the core distributed transaction coordination protocol. The most significant difference between the two schemes is Granola's explicit support for independent distributed transactions, which requires no locking; this is a significant benefit in suitable workloads, since the cost of locking is often equivalent to the cost of executing a transaction itself [30, 50]. Granola is also able to avoid lock conflicts for independent transactions, significantly improving throughput on workloads where lock conflicts are prevalent, such as our TPC-C implementation.

The H-Store project [50] advocates for transaction execution models similar to Granola, and is an inspiration for some of our work. H-Store offers a significant performance improvement for online transaction-processing workloads, by eschewing many of the mechanisms employed by legacy relational database management systems. Like Granola, H-Store uses a single thread of execution on each repository, to minimize the overhead from locking used to support concurrent transaction execution. H-Store also divides transactions into a number of different transaction types, and observes that transactions in the "one-shot" and "strongly-two-phase" class can be executed without locking. These transactions are similar to the independent transactions provided by Granola.

H-Store does not offer a practical solution for executing distributed transactions without locking, however: the initial H-Store paper proposed assigning a deterministic order to transactions by stalling each distributed transaction by the maximal expected network latency, assuming all transactions arrive at all repositories within this window of time. Granola introduces support for lock-free transactions regardless of network latency, by adopting a timestamp voting and propagation protocol. Granola also addresses important issues such as recovery from failure when executing lock-free transactions. Later work in the H-Store project presented more complete protocols for distributed transaction coordination, but use different techniques and do not optimize for independent transactions [32, 55]. VoltDB [55] is a commercial database implementation based on work in H-Store, but handles distributed transactions by pausing transaction execution when processing a transaction that accesses data on multiple repositories (based on personal correspondence at the time of writing).

Jones' Dtxn system [31, 32] adds distributed transaction coordination to the H-Store project by utilizing speculative concurrency control. This work avoids running two-phase commit for distributed transactions by speculating that a commit vote will be received for every transaction, and rolling back execution if the speculation was incorrect. Dtxn is able to provide very good performance in a workload like TPC-C, where speculation usually succeeds, but encounters cascading aborts if speculation fails. Granola avoids the need for two-phase commit votes in TPC-C altogether, by implementing every distributed transaction as an independent transaction. Dtxn requires the presence of a central coordinator node that tracks speculative dependencies, which can limit the scale of the system to tens of nodes when distributed transactions are common.

8.2 DETERMINISTIC TRANSACTION ORDERING

Recent work by Thomson and Abadi on deterministic database transaction ordering [53, 54] shares many similarities with Granola, and was developed in parallel with our work. In their work, a coordination service assigns a global ordering to transactions before they are received by storage repositories, so that these transactions can be executed independently at the storage nodes. This protocol uses locking for concurrency control, but avoids two-phase commit style communication between repositories for transactions that fit into the independent transaction model, since there is a predetermined serial transaction order.

The authors' original work on the subject [53] argues for the applicability of a transaction model where each transaction can be executed in a deterministic order at each storage node. This work thus supports Granola's transaction model, and argues for the relevance of transactions that can execute without locking. This work also argues that a large fraction of typical online transaction processing workloads can be expressed such that the locksets for transactions can be precomputed; this lends credence to the practicality of Granola's protocol for recovering from long-term failures, since the locksets for each transaction will be equivalent to the lock-supersets, as discussed in Section 5.3.1.

The original protocol developed by Thomson and Abadi uses a single coordinator node that determines the ordering of all transactions. Clients send all transactions to this coordinator, including single-repository transactions, which poses as a serious limit to scalability. Our original work on Granola initially explored the use of a coordinator hierarchy to determine transaction ordering, while using timestamps so that single-repository transactions didn't need to be sent to a coordinator. We opted instead for our distributed timestamp voting protocol, to support higher scalability.

The Calvin protocol [54] avoids some of the bottlenecks of a centralized coordinator, by splitting the coordinator across all storage nodes, as a set of transaction *sequencers*. Each transaction is issued to a sequencer or set of sequencers, and all sequencers execute a round of all-to-all communication to determine a global transaction ordering, before forwarding the transactions on to the storage layer. Calvin must delay transactions and order them in batches, to minimize the overhead of this all-to-all communication. This allows Calvin to scale beyond a hundred nodes, but introduces additional transaction latency, which was a design choice we avoided in Granola. Granola instead uses a fully-distributed mechanism for transaction ordering, and requires no communication between nodes that are not participants in the same transaction, removing this scalability bottleneck.

Single-repository writes need to be communicated to the sequencer layer in Calvin, along with distributed transactions, whereas Granola only requires multi-party transaction coordination for distributed transactions. Read-only single-repository transactions are issued directly to the storage nodes in Calvin, bypassing the sequencer layer. This means that the protocol does not provide causality for read operations: a read from a given client may observe the effects of transaction T, but then a subsequent read may observe a pre-state of transaction T at another participant. This anomaly could be rectified by incorporating timestamps and propagating timestamp ordering dependencies in client transactions, as required in Granola.

Calvin uses a lock-based execution model, and can thus support concurrent execution on multiple compute cores, and provide efficient support for disk-based workloads. Calvin is tailored explicitly towards database transactions, whereas Granola targets a more general application model. Granola and Calvin optimize for different points in the design space, with Granola aiming for lower latency and potentially wider scalability, but each offer compelling advantages for distributed transaction processing workloads.

9 CONCLUSIONS AND FUTURE WORK

There is an extensive history of research on distributed transaction coordination, as discussed in Section 8. This thesis presents a new contribution to this large body of work, by specifically targeting the new class of *independent distributed transactions*. Granola thus represents not only an interesting protocol and architecture for running distributed transactions, but also raises a number of opportunities for future research.

9.1 FUTURE WORK

We briefly discuss some possible areas for future work, both in terms of extending the Granola platform, and in further exploring the transaction model.

9.1.1 PROTOCOL IMPROVEMENTS

Although the base Granola protocol was already designed for high performance and low transaction coordination overhead, there is still potential for further optimizations.

Avoiding Blocking

The protocol for processing transactions in timestamp mode may result in blocking if the transaction with the lowest timestamp has not yet received a full set of votes. Our solution to this problem is to transition into locking mode if the repository is blocked for an extended period of time. Less heavyweight options are also possible, however, such as acquiring locks just for the blocked transaction and pushing any subsequent transaction past it if they don't conflict with the locks. It may also be advantageous to opportunistically abort read-only transactions that are taking a long time to complete, to avoid slowing down subsequent transactions in the queue.

If blocking occurs as the result of a particular participant being slow, it is also possible for the repository to assign higher timestamps to transactions involving that participant. The higher timestamps would position these transactions further back in the timestamp queue, and allow other transactions to execute while the repository is waiting for the slow participant. We originally adopted an adaptive timestamp protocol along these lines, but found that it wasn't necessary to achieve good performance in our benchmarks.

A full examination of strategies like these could make the Granola protocol more resilient to slow participants and asymmetric workloads.

Parallel Execution

We adopted a single-threaded execution model for server applications, to avoid the overhead and complexity of concurrency control within the application. This choice was partially motivated by arguments presented in the H-Store project [50], but subsequent work has since advocated for a multi-threaded execution model to take advantage of multiple compute cores and disk-based workloads [53, 54].

Granola could be trivially extended to support multi-threaded execution in locking mode, since locks already provide isolation between transactions. Multi-threaded execution in timestamp mode would require the application to determine which transactions can be executed in parallel. The application would need to acquire locks for parallel transactions, but only for the duration of a `run` upcall. Similar techniques are adopted in the Calvin protocol [54].

9.1.2 TRANSACTION MODELS

Granola presents a specific transaction model for application developers. This section addresses techniques for expanding on this transaction model, and converting existing transactions to fit our model.

Converting Coordinated Transactions to Independent Transactions

Granola provides its best performance when operating in timestamp mode, where only single-repository transactions and independent distributed transactions are supported. Some distributed transactions are more obviously modeled as coordinated distributed transactions, however, such as a bank transfer that only proceeds if the source account has sufficient funds.

It is possible to convert many coordinated transactions *into* independent transactions. One way to do this is to partition the data such that the commit decision can be determined independently by each participant; we use this strategy in our TPC-C implementation, by replicating the Item table at every repository. Schism [22] automatically determines a partitioning and replication strategy based on analysis of a database schema and workload, and aims to minimize the proportion of distributed transactions. Schism does not optimize for independent transactions, however, and does not preference independent transactions over coordinated transactions.

Another way to convert a coordinated transaction into an independent transaction is through the use of *escrow transactions* [44]. An escrow transaction could be used for a bank transfer by first running a single-repository transaction that takes funds out of the source account and places them in an escrow account, followed by an independent transaction that atomically transfers the funds. Additional management is required to track the escrow balance, and to notify the participants if the transfer needs to be aborted due to the client failing before completing the transfer. There is also the additional overhead of executing the transfer as two transactions instead of one.

Other techniques such as *demarcation* [13] can be used under many workloads to convert coordinated transactions into single-repository transactions. A full study of all such techniques, as applied to typical online transaction processing workloads, would serve as a valuable contribution and argue further for the relevance of Granola's transaction model.

Automatic Transaction Classes

Granola expects the application developer to determine which transaction class to use for each transaction; this information is communicated to the client proxy when a transaction is invoked. It would be far more desirable, however, to automatically determine the transaction class for a given transaction.

Granola currently cannot determine transaction classes itself, since it treats the operation in each transaction as an uninterpreted byte string, in order to support arbitrary applications. It may be possible, however, to automatically infer the transaction class for transactions that have a well-defined syntax and partitioning strategy, such as the use of SQL queries and a database schema. The Houdini system [45] uses machine learning to predict what repositories are going to be involved in a given transaction, but precisely determining the transaction class for a given transaction is still an open question.

Exchanging Data in Votes

Granola's one-round transaction model requires that transaction participants execute operations entirely in isolation, apart from a commit/abort vote for coordinated transactions. We could extend our model further by allowing arbitrary data to be exchanged along with the vote for the transaction. It may also be possible to exchange this data in lieu of a commit/abort vote, and have the application at each participant independently determine the commit decision based on the data exchanged in the vote messages.

One key constraint is that the data exchanged in vote messages cannot affect the locks required by the transaction, since the locks must be acquired at each participant before the votes are sent. The main open question thus pertains to the relevance of exchanging data in typical online transaction processing workloads, given this locking constraint.

9.1.3 DYNAMIC RECONFIGURATION

Granola provides mechanisms for migrating data between repositories, and for mapping logical data partitions to the repositories that are responsible for this data. We also provide a mechanism for tracking and updating system membership, by way of the Census membership management system [21]. It would be highly desirable in a real-world deployment, however, to automatically manage this system configuration in response to load.

Desirable features in a dynamic management system include:

- Automatically splitting or merging partitions in response to client load.
- Adding additional replicated partitions to handle read-heavy workloads.
- Migrating data to machines close to the users accessing that data.
- Migrating repositories to nearby machines if they are frequently involved in distributed transactions together.
- Locating primary replicas in a single data-center to minimize voting latency.
- Colocating witnesses and backups on particular machines to improve utilization while maintaining failure independence [56].

A configuration management system that supports some of the features above would be a valuable contribution for distributed storage systems in general, not just for Granola.

9.2 SUMMARY OF CONTRIBUTIONS

This thesis has presented Granola, an infrastructure for building distributed transactional storage applications. Granola provides strong consistency for all operations, and supports atomic distributed transactions, while maintaining high throughput, low latency, and wide scalability. Granola thus presents an argument against the conventional wisdom that it's infeasible to provide strong consistency in a high-performance distributed storage system.

Granola's protocol design and lack of dedicated transaction coordinators allows it to provide low per-transaction latency. Granola is able to execute single-repository transactions in one network round-trip from the client plus a stable log write, and to execute distributed transactions in only three one-way message delays and one stable log write, including the round-trip from the client. This latency is significantly lower than traditional approaches to distributed transaction coordination, and compares favorably with existing storage systems that do not provide strong consistency or distributed transactions.

One of Granola's main contributions is its use of timestamps to support *independent distributed transactions*, a new class of transaction where the commit decision can be computed independently by each transaction participant. Granola is able to provide serializability for independent transactions without locking or two-phase commit. This can significantly reduce transaction coordination overhead, and allows each repository to coordinate many independent transactions in parallel, without encountering a reduction in throughput due to lock conflicts and retries.

Granola's timestamp-based coordination protocol is entirely distributed, and does not require the presence of a centralized coordinator or coordination layer; each distributed transaction only involves the repositories that are participants for that transaction. This allows us to avoid the scalability bottlenecks inherent in centralized coordination systems, and also to minimize the number of communication phases for a given transaction.

Independent transactions occur commonly in online transaction processing workloads, and we were able to implement the TPC-C transaction processing benchmark using only single-repository transactions and independent distributed transactions. Our experiments show that the use of independent transactions can result in a significant improvement in throughput compared to existing mechanisms, as well as a reduction in per-transaction latency.

A MESSAGES

This chapter lists the message formats used in Granola, showing what information needs to be communicated in an implementation of the protocol.

A.1 MESSAGE FORMATS

The contents of each message are defined as follows.

A.1.1 REQUEST

Request	`tid`	`ts`	`rids`	`ops`	`ro`	`indep`

`tid`: The TID assigned to the transaction. This is composed of the unique client ID and unique transaction sequence number.

`ts`: The highest timestamp observed so far at the client.

`rids`: The IDs for the set of repositories that are involved in the transaction.

`ops`: The byte-string operations to be sent to each repository in `rids`. If only one operation is provided, it is sent to all repositories in `rids`.

ro: [optional] Whether the transaction is read-only. This flag is only relevant for single-repository transactions or independent transactions.

indep: [optional] Whether the transaction is independent. This flag is only relevant for distributed transactions (where `length(rids) > 1`).

A.1.2 VOTE

Vote	tid	rid	ts	status	retry

tid: The TID for the transaction.

rid: The ID of the repository sending the vote.

ts: The proposed timestamp for the transaction.

status: Set to `COMMIT` if voting to commit the transaction, `CONFLICT` if aborting due to a lock conflict, or `ABORT` if aborting due to application logic.

retry: [optional] This flag is set if the repository is resending this vote to trigger a retransmission response, since it has not yet received a vote from the recipient.

A.1.3 REPLY

Reply	tid	rid	ts	status	result

tid: The TID for the transaction.

rid: The ID of the repository responding.

ts: The final timestamp assigned to the transaction.

`status`: Set to `COMMIT` if the transaction committed, `CONFLICT` if the transaction aborted due to a lock conflict and should be retried, or `ABORT` if it aborted due to application logic.

`result`: [optional] The transaction result, if one is given by the application. Results may also be included for `ABORT` responses.

A.2 PROTOCOL BUFFERS DEFINITIONS

A Protocol Buffers [9] definition file for these messages is provided as follows in Figure A.1.

```
/*
 * Datatypes for use in other messages
 */

message TID {
    required uint32 cid = 1;
    required uint32 seqno = 2;
}

message Addr {
    required string host = 1;
    required uint32 port = 2;
}

enum Status {
    STATUS = 1;   // commit vote or result
    CONFLICT = 2; // abort due to lock conflict
    ABORT = 3;    // abort due to application logic
}
```

Figure A.1: Basic message definitions in Protocol Buffers format.

Continued on next page.

```
/*
 * Messages
 */

message Req {
    required TID tid = 1;
    required uint64 ts = 2;
    optional Addr caddr = 3;
    repeated uint32 rids = 4 [packed = true];
    repeated bytes requests = 5;
    optional bool ro = 6 [default = false];
    optional bool indep = 7 [default = true];
}

message Vote {
    required TID tid = 1;
    required uint32 rid = 3;
    required uint64 ts = 4;
    required Status status = 5;
    // whether this vote is a retry
    // if so, the the receiver should send one back
    optional bool retry = 6 [default = false];
}

message Rep {
    required TID tid = 1;
    required uint32 rid = 2;
    required uint64 ts = 3;
    required Status status = 4;
    optional bytes result = 5;
}
```

Figure A.1: Basic message definitions in Protocol Buffers format.

Continued from previous page.

B RELEASING LOCKS FOR INDEPENDENT TRANSACTIONS

Granola was designed to operate primarily in timestamp mode, and this mode is where the protocol achieves best performance. Section 6.6.2 evaluates how often a repository will be operating in timestamp mode, given the likelihood that a repository will be processing no coordinated transactions at a given point in time. These results assumed, however, that a repository can switch back into timestamp mode immediately once it has completed execution of any coordinated transactions. Whether the repository can actually switch immediately depends on how we deal with locks that were acquired for current independent transactions.

In practice independent distributed transactions will likely be received while in locking mode, and these transactions will be processed using the locking mode protocol, i.e., these transactions will undergo a prepare phase, and the application will acquire locks for them. These locks need to be released before the repository can switch to timestamp mode; this is performed in our default implementation by issuing an `abort` application upcall for each independent transaction, then later re-executing each transaction with a `run` upcall, following the timestamp mode protocol. This chapter examines alternative approaches to releasing these locks, to try to avoid the overhead of extraneous `abort` upcalls.

Figures B.1 and B.2 examine the fraction of time spent in locking mode, and the total system throughput, for three different protocols for transitioning from locking mode to timestamp mode:

Stay Coord.: The repository remains in locking mode until there are no active independent transactions, and won't switch back until this point. Any new independent transaction that arrives during this period will also be processed in locking mode.

Undo Immediate: The repository immediately issues `abort` upcalls for all active independent transactions, to undo any changes and release their locks. These transactions can then be completed using the timestamp mode protocol. This is the default policy used in our Granola implementation.

Undo On-Demand: The repository starts accepting new transactions in timestamp mode, but does not immediately abort any independent transactions that were processed in locking mode. If a new transaction conflicts with one of the existing locks when executed using a `run` upcall, then the transaction that triggered the lock conflict will be aborted and re-run using the timestamp mode protocol.

These benchmarks include real transaction work and locking overhead to accurately represent the costs of the three protocols; locking overhead is set to 50% of the transaction execution cost. In each figure we run the system at maximum throughput, on a two-repository topology, and vary the fraction of the workload that is comprised of independent distributed transactions.

These experiments aim to examine the impact of a given fraction of independent transactions. Each experiment includes a constant 0.05% rate of coordinated transactions, to trigger each repository to switch in and out of locking mode. The remainder of the workload is made up of single-repository transactions and independent transactions. We vary the fraction

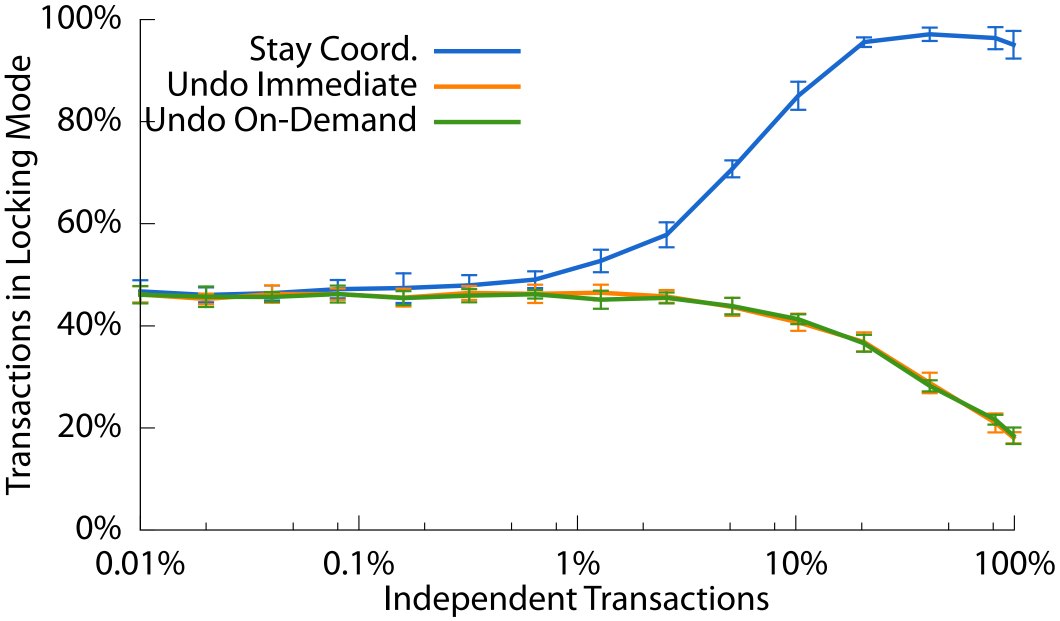

Figure B.1: The number of transactions processed in locking mode at a given repository, for the three different transition protocols, as a function of fraction of the total workload that's composed of independent transactions. Coordinated transactions comprise a fixed 0.05% of the workload.

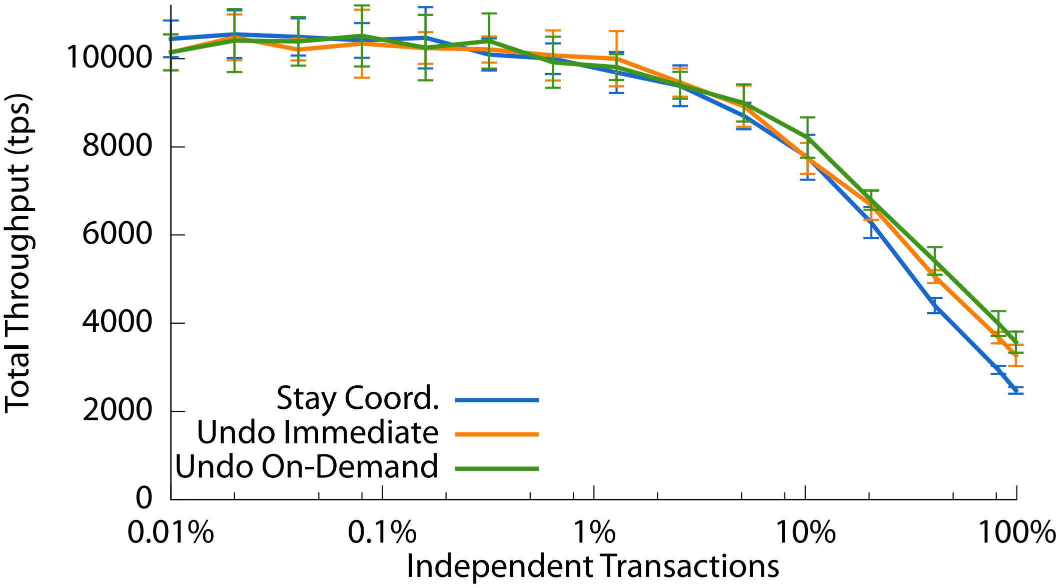

Figure B.2: Two-repository throughput for the three different transition protocols, as a function of fraction of the total workload that's composed of independent transactions. Coordinated transactions comprise a fixed 0.05% of the workload. Locking overhead is set at 50% of the transaction execution cost.

of independent transactions between 0% and 99.95%, with a corresponding fraction of single-repository transactions between 99.95% and 0%.

Figure B.1 shows the fraction of time spent in locking mode, for a given workload distribution. We see that *Stay Coord.* is as effective as the *Undo* protocols at transitioning into timestamp mode at low rates of independent transactions. At high rates, however, subsequent coordinated transactions arrive before all previous independent transactions have completed, preventing the repository from successfully transitioning into timestamp mode. The curves for *Undo Immediate* and *Undo On-Demand* taper off at high rates of independent transactions since the maximum throughput decreases, and hence the number of concurrent active transactions also decreases. Note that the x-axis in this figure extends to 99.95%, not 100%, due to the 0.05% of the workload that is composed of coordinated transactions.

Figure B.2 shows the total system throughput for the three protocols. The throughput drops off as the fraction of independent transactions increases, since the repositories are processing a larger number of distributed transactions. We only observe a minor throughput difference between the *Undo Immediate* and *Undo On-Demand* protocols. This is because transitions between timestamp mode and locking mode are more rare at high rates of independent transactions, and thus the relative overhead of undoing transactions is low. Since the performance difference is small, we adopt the simpler *Undo Immediate* protocol in our default implementation.

Stay Coord. performs less well than the *Undo* protocols in terms of throughput, but also incurs 20% higher transaction latency due to the more frequent use of locking mode. There are no lock conflicts in this specific benchmark: if lock conflicts are introduced, the throughput of *Stay Coord.* drops. *Undo Immediate* and *Undo On-Demand* have even closer performance at higher rates of lock conflict, since there is less benefit from deferring transaction undo.

C MASTER NODES

Our original design for Granola included a layer of dedicated *master nodes* that were responsible for ordering all distributed transactions. Master nodes did not determine the final serial order of transactions, but rather just the order in which each repository received each transaction; the serial order was determined by the timestamp voting protocol, as in our current protocol. Masters ensured that all participants received transactions in the same relative order, and hence all participants would issue a `prepare` upcall for coordinated transactions in the same order. This deterministic ordering eliminated the possibility of deadlock, and allowed us to block transactions that encountered a lock conflict, rather than aborting them.

We determined through our experimental evaluation that it was better to abort transactions that conflict on a lock, rather than result in head-of-line blocking as in the original protocol. Aborts result in additional transaction delay and message communication cost, but they provide a degree of independence between transactions — in the blocking protocol a long chain of distributed transactions could end up being delayed due to a single transaction that conflicts on a lock. This delay is also propagated to other participants, since it results in vote messages being delayed.

Once we opted to abort conflicting transactions rather than block them, the master nodes no longer provided a significant performance benefit.

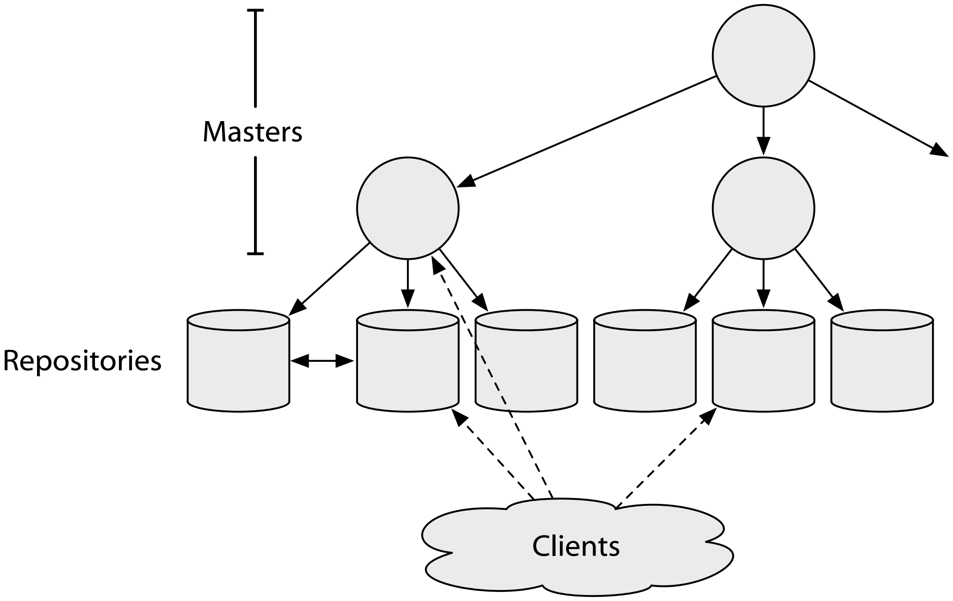

Figure C.1: Granola system topology when using master nodes.

They also added complexity and latency to the transaction coordination protocol, hence we removed them from the core Granola protocol. We outline our master-based protocol in this chapter, and discuss some of the minor performance benefits from using master nodes to assist in transaction coordination.

C.1 PROTOCOL OVERVIEW

Master nodes are arranged into a hierarchy as shown in Figure C.1. Each client issues a distributed transaction to the lowest master in common with all transaction participants. The master then forwards the transaction down the tree to the participants. When a repository receives a transaction from its parent, it processes it using the standard Granola protocol.

The hierarchy allows the system to scale and support multiple masters, while ensuring that a single master is responsible for the ordering of transactions issued to any given set of participants. The master hierarchy scales well if the system is able to be partitioned such that distributed transactions typically involve repositories that are nearby in the hierarchy, in which case the majority of transactions will be issued to a lowest-level master. The hierarchy works less well, however, if the system does not exhibit locality, and a significant fraction of distributed transactions end up being issued to the root of the hierarchy.

Master do *not* need to record a stable log write when receiving a transaction. Masters are entirely soft-state, except for a generation number that is incremented every time the master recovers from failure. The generation number can be stored using a replicated state machine at each master, or by using a single management service that stores the generation numbers for all masters and replaces a master in the case of failure.

Since masters do not keep a stable record of the transactions that they forward, a master might resend a set of transactions in a different order after recovering from failure. The generation number ensures a consistent ordering at the repositories: a vote for a transaction at a newer generation number supersedes a vote for an older generation number, and any transaction committed at the previous generation number will retain its old ordering.

The protocol at the masters is very lightweight, and hence each master can support a large number of children in the hierarchy. We were able to support many tens of repositories under a single master with high rates of distributed transactions. Hundreds of repositories could be supported under a single master at lower distributed transaction rates.

C.2 DOWNSIDES OF MASTERS

The main downside of the use of masters is the additional complexity it introduces to the protocol, both in terms of the protocol design and in the administrative overhead of running the system. Masters also introduce additional points of potential failure.

The use of masters requires at least one additional one-way message delay for distributed transactions, and potentially more message delays if a transaction is issued to a master higher up in the hierarchy. This additional latency is particularly undesirable since low latency was a major goal in

Granola's design.

Masters also introduce an additional assumption regarding the transaction workload: that the workload exhibits sufficient locality that the majority of transactions will be issued to a lowest-level master. This assumption introduces an additional constraint to our transaction model for deployments intended to scale beyond a single master.

C.3 PERFORMANCE BENEFITS

Masters were deemed to be of insufficient performance benefit once we changed the Granola protocol to abort conflicting transactions rather than blocking them. Masters still retain one benefit, however, which is providing each repository with transactions in a consistent order. In the absence of masters, one repository might receive a pair of transactions in one order, while another repository receives them in the reverse order. This does not pose a correctness issue, since the use of timestamps defines a consistent transaction order, but it can lead to reduced throughput in our two transaction modes:

Timestamp Mode: A transaction T that arrives at two participants at significantly different times will be queued at the participant that first received T. If the first participant assigned a low proposed timestamp to T, this can result in the execution of subsequent transactions being blocked until the vote for T arrives.

Locking Mode: If a pair of transactions T_1 and T_2 both conflict on the same locks, and arrive at two participants in the same order, then T_1 will commit and T_2 will be aborted. If they arrive at the participants in opposite orders, however, then both T_1 and T_2 may abort.

We examine the performance impact of transactions arriving in different orders in the following sections, for our two transaction modes.

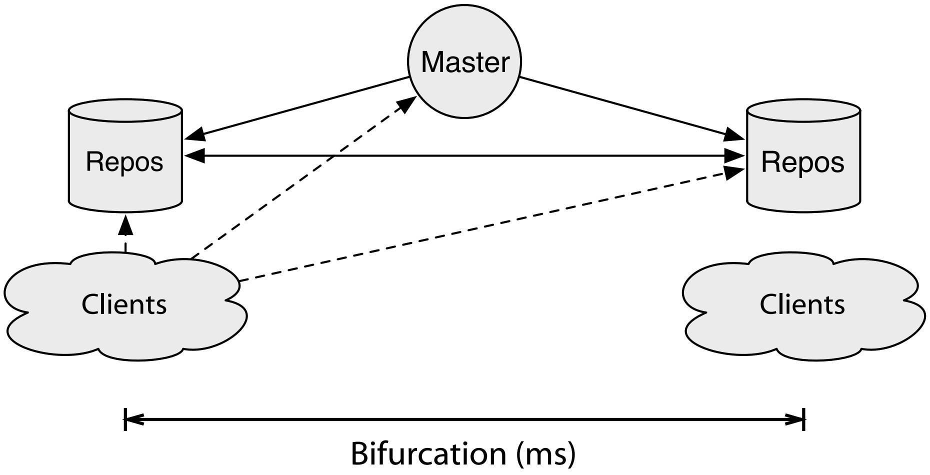

Figure C.2: Bifurcated network topology used when evaluating the performance benefit of using masters. The repositories are located a given network delay apart, and half of the clients are colocated with each repository. The master is located mid-way between the repositories when using the master-based protocol.

Timestamp Blocking in Timestamp Mode

We examine the impact of a skew in transaction arrival times by using a bifurcated topology, as illustrated in Figure C.2. We split the system into two halves, with half of the clients colocated with one repository, and half colocated with the other repository. The two clusters are separated by a one-way network delay that we refer to as the *bifurcation*, varied between 0 and over 30 ms. Each client issues a workload comprised entirely of independent distributed transactions, to both repositories. This represents a worst-case scenario, where every transaction arrives at one participant x ms earlier than the other, for a given bifurcation value x.

We examine the Granola protocol both with and without masters. The master is located mid-way between the two repositories, such that transactions are received at approximately the same time at both participants when using masters.

Figure C.3 shows the total throughput for two repositories for a given bifurcation value. Throughput drops for protocols as the bifurcation increases, since the additional network delay leads to a reduction in per-client

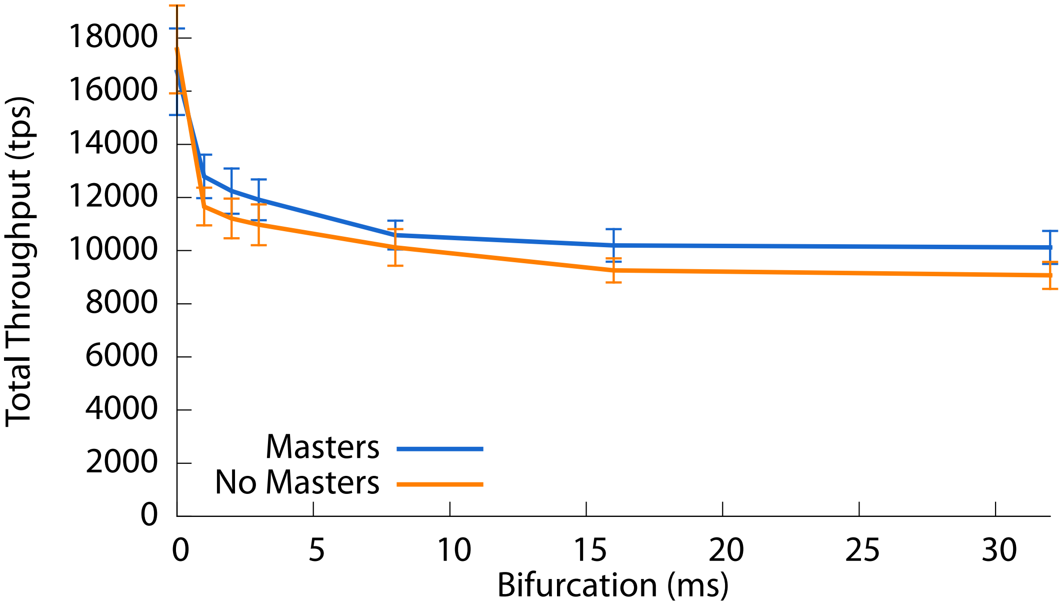

Figure C.3: The impact on throughput by avoiding timestamp blocking in a master-based protocol, for a bifurcated network topology with workload comprised of 100% independent transactions on two repositories. The repositories are separated by the bifurcation delay (one-way network delay), and half the clients are colocated with each repository. The master is located mid-way between the two repositories.

throughput, and each repository has to buffer a much larger number of transactions while awaiting votes. At 0 ms bifurcation there is minimal benefit from using masters, since transactions already arrive at each participant at roughly the same time. At higher bifurcation levels there is a larger difference in throughput, but the advantage from using masters is still minimal.

Excess Aborts in Locking Mode

We use the same topology as in the previous section to evaluate the impact of aborts due to lock conflicts. We set bifurcation at 10 ms, and use a client workload comprised entirely of coordinated distributed transactions. We then vary the fraction of transactions that attempt to acquire a lock on a common shared field, and thus potentially encounter a lock conflict.

Figure C.4 shows the throughput on our two-repository topology with

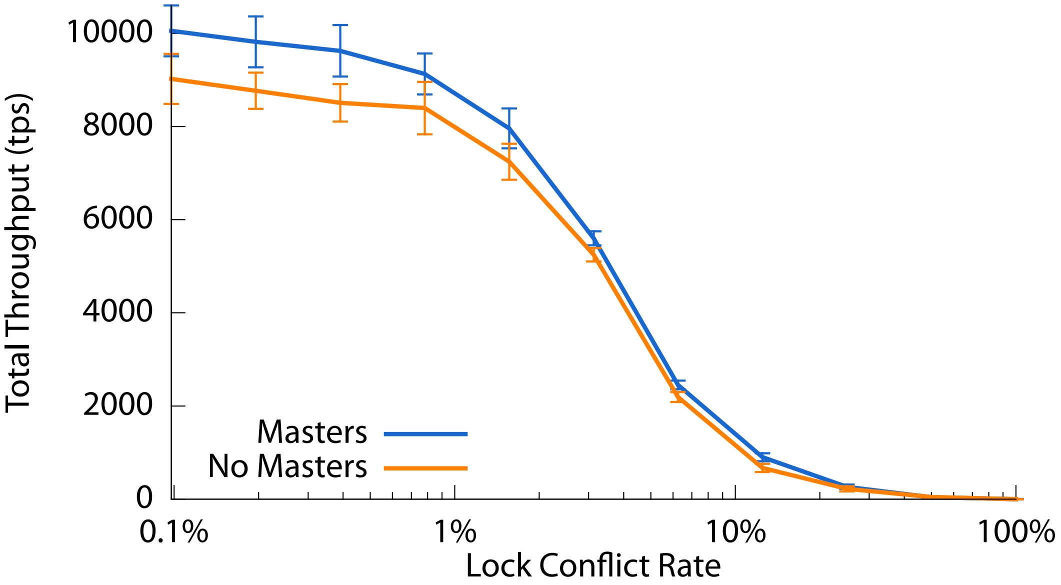

Figure C.4: The impact on throughput by avoiding transaction aborts in a master-based protocol, for a bifurcated network topology with workload comprised of 100% coordinated transactions on two repositories. The repositories are separated by 10 ms one-way bifurcation delay, and half the clients are colocated with each repository. The master is located mid-way between the two repositories.

a lock conflict rate varied between between 0.1% and 100%. At low levels of lock conflicts, masters provide a visible benefit, since slightly fewer transactions abort for each lock conflict. At higher conflict rates, however, this advantage diminishes. This is because multiple transactions tend to conflict simultaneously, and we no longer gain a significant benefit from having the first transaction in a chain of conflicts commit without aborting.

We observed only a very minor benefit from the use of masters on more realistic workloads, where there are many repositories and transactions involve different sets of participants. Each repository acts as a participant for different transactions, and hence encounters different lock conflicts. Under these circumstances there is far less benefit from receiving transactions in a consistent order when in locking mode.

BIBLIOGRAPHY

[1] Distributed TP: The XA Specification. Number C193. The Open Group, February 1992.

[2] Amazon SimpleDB. `http://aws.amazon.com/simpledb/`, 2007.

[3] Apache Cassandra. `http://cassandra.apache.org`, 2008.

[4] Apache CouchDB. `http://couchdb.apache.org`, 2008.

[5] Apache HBase. `http://hbase.apache.org`, 2008.

[6] Shore - A High-Performance, Scalable, Persistent Object Repository. `http://research.cs.wisc.edu/shore/`, 2008.

[7] MongoDB. `http://www.mongodb.com`, 2009.

[8] TPC benchmark C. Technical report, Transaction Processing Performance Council, February 2010. Revision 5.11.

[9] Google Protocol Buffers. `http://code.google.com/p/protobuf/`, 2012.

[10] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 27(3), November 2009.

[11] Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, Baltimore, MD, USA, 1998.

[12] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2011.

[13] Daniel Barbará and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3), June 1994.

[14] Philip A. Bernstein and Nathan Goodman. Serializability theory for replicated databases. *Journal of Computer and System Sciences (JCSS)*, 31(3), December 1985.

[15] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[16] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM symposium on Principles of Distributed Computing (PODC)*, Portland, OR, United States, July 2000.

[17] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, November 2006.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4), November 2002.

[19] 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. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, November 2006.

[20] 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. *Proceedings of the VLDB Endowment*, 1(2), August 2008.

[21] James Cowling, Dan R. K. Ports, Barbara Liskov, Raluca Ada Popa, and Abhijeet Gaikwad. Census: location-aware membership management for large-scale distributed systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2009.

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

[23] 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. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, USA, October 2007.

[24] David DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[25] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), November 1976.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, October 2003.

[27] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[28] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science. Springer-Verlag, 1978.

[29] R. Guy, J. Heidemann, W. Mak, Jr. Page, T., G. Popek, and D. Rothneier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, Anaheim, CA, USA, June 1990.

[30] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, June 2008.

[31] Evan P. C. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis in Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 2012.

[32] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 2010.

[33] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems (TOCS)*, 10(2), November 1992.

[34] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7), July 1978.

[35] L. Lamport. The Part-Time Parliament. Technical Report Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, USA, September 1989.

[36] B. Lampson. Atomic transactions. In *Distributed Systems: Architecture and Implementation*, volume 105 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[37] K.-J. Lin. Consistency issues in real-time database systems. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume 2, January 1989.

[38] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, USA, October 1991.

[39] Barbara Liskov and James Cowling. Viewstamped Replication revisited. Technical report, MIT CSAIL, 2012.

[40] John D. C. Little. A proof for the queuing formula: L= λW. *Operations Research*, 9(3), 1961.

[41] Wyatt Lloyd, Michael J. Freedmand, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[42] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4), December 1986.

[43] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM symposium on Principles of Distributed Computing (PODC)*, Toronto, ON, Canada, August 1988.

[44] Patrick E. O'Neil. The Escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4), December 1986.

[45] Andrew Pavlo, Evan P.C. Jones, and Stanley Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *Proceedings of the VLDB Endowment*, 5(2), October 2011.

[46] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010.

[47] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[48] F. B. Schneider. The state machine approach: A Tutorial. Technical Report TR 86-600, Cornell University, Dept. of Computer Science, Ithaca, NY, USA, December 1986.

[49] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[50] Michael Stonebraker, Samuel R. 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 Databases (VLDB)*, Vienna, Austria, September 2007.

[51] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, June 2009.

[52] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, USA, December 1995.

[53] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1), September 2010.

[54] Alexander Thomson, Thaddeus Diamond, Shu chun Weng, Philip Shao, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Scottsdale, AZ, USA, May 2012.

[55] VoltDB Inc. VoltDB. `http://voltdb.com`.

[56] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the Sixth European Conference on Computer systems (EuroSys)*, Salzburg, Austria, April 2011.