

REDUCED COMMUNICATION FOR DISTRIBUTED
TRANSACTIONS THROUGH TIME-DEPENDENT
GUARANTEES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Thomas Robert Magrino

August 2019

© 2019 Thomas Robert Magrino
ALL RIGHTS RESERVED

REDUCED COMMUNICATION FOR DISTRIBUTED TRANSACTIONS THROUGH TIME-DEPENDENT GUARANTEES

Thomas Robert Magrino, Ph.D.

Cornell University 2019

Modern software is highly concurrent, with many operations contending for shared information stored across large geographic distances. The systems on which these applications are written must provide a well-defined semantics that is easy to understand so that programmers can ensure their software is correct. Strictly serializable transactions provide a particularly simple interface for writing code in a concurrent setting, but they come at a cost: implementations require commit protocols to resolve contention between potentially conflicting transactions, sometimes coordinating across distant nodes.

This dissertation explores *warranties*, time-dependent guarantees on the system's state, which improve the performance of distributed transaction systems by avoiding synchronous communication. Warranties guarantee that a predicate over the system's state and the current time holds until an associated expiration time. These predicates can express a wide variety of checks performed by applications ranging from simple comparisons to complex application-specific logic. Warranties can be constructed compositionally, using other warranties as evidence that a more complex predicate holds. Furthermore, these predicates can be *time-varying*, expressing guarantees about *trends* on the system's state. While holding an active warranty, nodes do not need to perform synchronous communication to validate the associated assertion.

The system enforces an active warranty by delaying updates that falsify the guarantees until the warranty is safely retracted or expires. To ensure the benefits of warranties outweigh delays to updates, the system uses a cost model to deter-

mine a warranty's expiration time and to select other simpler warranties that help enforce a warranty with low overhead. Using a variety of benchmarks and real-world applications, warranties are shown to significantly improve the performance of distributed transaction systems.

BIOGRAPHICAL SKETCH

Tom Magrino was born in San Diego, CA in 1990 to Tom and Kathy Magrino. Outside of a few initial years moving around due to his father's career in the Navy, he spent his childhood there with his two younger siblings Nicole and Alex. All three Magrino children probably never understood how lucky they were to enjoy the perfect sunny weather of San Diego.

Tom's family moved to Davis, CA in 2004, right before he started high school. In his high school chemistry class, Tom met a friend who first introduced him to programming. He was immediately enthralled with the practice of programming and had long been fond of the notion of engineering, leading him to join the high school FIRST robotics team (team 1678) where he had his first experiences with working on long term collaborative technical projects.

After graduating from high school in 2008, Tom went on to study electrical engineering and computer science at University of California, Berkeley. At UC Berkeley, Tom was encouraged by his peers to participate in academic research, which he found challenging and fulfilling. Those undergraduate research experiences convinced Tom he would be happiest pursuing a PhD after graduating in spring of 2012, leading him to Cornell University the following fall.

After his first year at Cornell, Tom met his wife, Marin Cherry, whom he married in 2017. They have shared countless adventures together over the years. Tom and Marin care for two cats, Shakira and Otto, who have provided great emotional support to both of their humans while living in Ithaca, NY.

For my family, Tom, Kathy, Nicole, and Alex, who taught me to dream big.

For my wife, Marin, who helps me turn those dreams into reality.

ACKNOWLEDGMENTS

While pursuing a PhD has the potential to be an uniquely isolating experience at times—long hours working away at experiments, reading papers, and staring at whiteboards thinking *deep* thoughts—it is often a fundamentally communal experience. I would not have gotten to this point without the amazing help of many, many people. I hope I’m not forgetting any names here.

First, I would like to thank my advisor, Andrew Myers. Andrew has been a thoughtful and patient advisor to me, even at times when I may not have been the most thoughtful and patient student. Over the years at Cornell, Andrew has taught me to think critically, work hard, and invest time in helping my peers. Most of all, Andrew has taught me so much about how to communicate research, both in writing and speaking.

Furthermore, I’d like to thank my committee members, Fred Schneider and Dexter Kozen, for their thoughtful feedback and challenging me to be more specific in my ideas.

The work presented in this dissertation is the product of funding from the National Science Foundation and a National Defense Science and Engineering Graduate fellowship. Furthermore, they were born out of collaborations with a number of extremely terrific and astonishingly smart folks: Jed Liu, Owen Arden, Mike George, Nate Foster, Johannes Gehrke, and Andrew Myers.

In particular, Jed taught me so, so much about how to conduct research as a PhD student during our collaborations. Jed taught me how to design systems, build systems, and—most importantly—how to *debug* systems. Without his help, I’d have probably taken much longer to build and fix research software and I’d have walked away from it with much less hair.

Ken Birman and Zhiyuan Teo taught one of my first graduate courses, Ad-

vanced Systems (CS6410), where I developed the earliest predecessor of computation warranties as a class project. Their early feedback on that project and their guidance on how to read and think about systems research was a fundamental early step to getting to where I am.

One of the greatest pleasures of being advised by Andrew Myers is being part of the quirky and energetic Applied Programming Languages (APL) research group. Over the course of my PhD work, APL has been home to a phenomenal band of individuals: Jed Liu, Michael D. George, Danfeng Zhang, K. Vikram, Owen Arden, Chinawat Isradisaikul, Yizhou Zhang, Isaac Sheff, Matthew Milano, Ethan Cecchetti, Rolph Recto, Drew Zagieboylo, Josh Acay, and Siqui Yao. Every week, I had the pleasure of attending the APL group meeting where I'd get to trade excessively bad jokes with everyone and hear about the diverse projects everyone was working on. The group has been a tremendous source of feedback on paper drafts and new perspectives when I needed a second opinions on research problems.

Beyond the research group, the systems lab and the Cornell CS department have been a wonderful academic family to me. Both the lab and the department have grown by leaps and bounds since I've started here and I'm astonished how it continues to get better and better at scale.

I don't think I'd have finished my PhD without the invaluable social and emotional support of my friends. Sofia Abreu Faro and Shrutarshi Basu were wonderful roommates in what we lovingly called the "fun house" for a year in Ithaca. I'll always look back fondly on our time living together. Andrew Hirsch, Isaac Sheff, Josh Moore, Hussam Abu-Libdeh, Ross Tate, Ann Tate, Jed Liu, Fabian Mühlböck, Laure Thompson, Alex Fix, Louise Felker, Erin Chu, Alyce Daubenspeck, Eleanor Birrell, Eston Schweickart, Ethan Cecchetti, Matthew Milano, Natacha Crooks, and Michael Roberts all made my time in Ithaca an absolute delight.

I'd also like to thank friends from home who helped support me throughout this journey—particularly Robert Shaffer, Rohit Poddar, Franklin Dingemans, Daisy Zhou, Kevin Ho, Aaron Wong, and Andrew Nguyen. Online gaming and conversations with all of you really kept my spirits up, particularly when I was either homesick or not feeling great about my progress in graduate school.

My family has been wonderfully supportive and loving throughout this entire process. They ensured that I was raised to be curious, hard working, and humorous.

Finally, I *know* I wouldn't have made it this far without the loving support of my wife, Marin Cherry. Marin, thank you for being such a saint. I thank my lucky stars everyday that you've been with me on this adventure.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgments	v
Table of Contents	viii
List of Tables	xi
List of Figures	xii
1 Introduction	1
1.1 Beyond Optimism	3
1.2 Predicates to Avoid Contention	4
1.3 Compositional Predicates for Low-Overhead Enforcement	5
1.4 Capturing Trends with Time-Varying Predicates	6
1.5 The Warranty Design Space	6
1.5.1 Consistency	7
1.5.2 Revocability	7
1.5.3 Predicate Scope	8
1.5.4 Time Dependence	9
1.6 Dissertation Outline	9
2 System Model	11
2.1 Strong Consistency	11
2.2 Optimistic Concurrency Control	13
2.3 Transactions Across Multiple Stores	14
2.4 Clock Synchronization	15
3 Public Warranties	16
3.1 Warranties as Optimistic Concurrency Control	17
3.2 Warranties as Generalized Read Leases	18
3.3 Issuing State Warranties	19
3.4 Distributing Public Warranties	19
3.5 Defending Public Warranties	21
3.6 Performance Trade-offs	22
3.7 Setting Warranty Terms with Workload Estimation	23
3.8 Using Warranties in Transactions	25
3.8.1 The Warranty Commit Protocol	26
3.8.2 Avoiding Protocol Phases	27
3.9 Public Warranties Implementation	28
3.10 Evaluation	29
3.10.1 Multiuser OO7 Benchmark	30
3.10.2 Course Management System	31
3.10.3 Comparing with Hibernate/HSQLDB	31
3.10.4 Experimental Setup	32

3.10.5	Results	32
3.11	Discussion	35
4	Computation Warranties	37
4.1	Example Applications	39
4.1.1	Generated Web Pages	39
4.1.2	Top N Items	40
4.1.3	Searching for Airline Seats	41
4.2	Programming with Computation Warranties	41
4.3	Ensuring Correct Behavior Using Computation Warranties	43
4.4	Proposing and Issuing Computation Warranties	44
4.5	Using Computation Warranties	45
4.6	Setting Computation Warranty Terms	46
4.7	Defending Computation Warranties	46
4.7.1	Incremental Revalidation	47
4.8	Public Computation Warranties Implementation	48
4.9	Evaluation	49
4.9.1	Top-Subscribers Benchmark	49
4.9.2	Course Management System	49
4.9.3	Results	50
4.10	Related Work	51
4.11	Discussion	53
5	Predictive Treaties	55
5.1	Predictive Treaties by Example: Voting	57
5.1.1	Enforcing Predicates with Slack	58
5.1.2	Time-Dependent Treaties	60
5.1.3	Preliminary Evaluation	62
5.1.4	Hierarchical Treaties	63
5.2	Predictive Treaties and Metrics	65
5.2.1	Predictive Treaties	65
5.2.2	Enforcing Predictive Treaties	67
5.2.3	Metrics	69
5.2.4	A Prediction Model for Metric Updates	70
5.2.5	Expiration	71
5.3	Using Predictive Treaties	74
5.3.1	Programming with Treaties and Metrics	74
5.3.2	Stipulated Commit	77
5.4	Automatically Creating Low-Coordination Treaties	80
5.4.1	Estimating Model Parameters for Metrics	80
5.4.2	Automatically Choosing an Enforcement Strategy	83
5.5	Implementation	86
5.5.1	Integration with Distributed Transactions	86
5.5.2	Opportunistic Slack Reallocation	89

5.6	Evaluation	89
5.6.1	Voting Microbenchmark	90
5.6.2	Distributed Top- k Monitoring	94
5.6.3	Modified TPC-C	97
5.6.4	Discussion	101
5.7	Related Work	102
5.8	Discussion	105
6	Conclusions	106
6.1	Public State Warranties	106
6.2	Computation Warranties	106
6.3	Predictive Treaties	107
6.4	Future Work	108
6.4.1	Combining Leased and Public Warranties	109
6.4.2	Fault Tolerance and Failure Recovery	109
6.4.3	Techniques for Warranty Search and Discovery	110
6.4.4	Population Statistics for Metrics	110
6.4.5	Support for Non-Numeric Metric Data	111
6.4.6	Alternative Prediction Models	112
	Bibliography	113

LIST OF TABLES

3.1	Policy parameters for setting a warranty's term.	24
3.2	Round trips performed during the commit protocol with Warranties vs. traditional OCC.	27
3.3	CMS throughput and latency on various systems with public warranties highlighted.	34
4.1	Top-N benchmark: maximum throughput, latency, and 95th percentile write delay.	50
4.2	CMS throughput with additional public computation warranties result highlighted.	50
5.1	Mean relative error of parameter estimates for various scenarios. . .	82

LIST OF FIGURES

3.1	Public warranty distribution architecture. Public warranties can be distributed through a CDN to reduce load on stores.	20
3.2	The warranty commit protocol.	26
3.3	OO7 maximum throughput on a 2%-write workload as the number of stores increases.	33
3.4	Effect of write percentage on OO7 maximum throughput on 3 stores with 24 workers.	34
4.1	Update in tree of computation warranties.	47
5.1	Locally enforceable predictive treaties imply a global predicate. . .	59
5.2	Time-dependent predictive treaties and corresponding local vote margins over time.	61
5.3	CDF of time until first synchronization under three different slack-allocation strategies.	62
5.4	Hierarchical predictive treaties.	64
5.5	A predictive treaty falsified by time passing.	67
5.6	System state s evolves within the intersection of local predictive treaties, enforcing global predicate $\phi(s)$	68
5.7	Setting a treaty's expiry based on predicted trajectory.	72
5.8	Treaty and Metric interfaces.	75
5.9	Voting example code using treaties.	76
5.10	Metrics tree created by example code.	76
5.11	Using stipulated commit to withdraw money from a sharded balance.	79
5.12	During the first step of treaty generation, subtreaty predicate templates are constructed by the derived metric based on the predicate being asserted.	84
5.13	CDF of time until first synchronization under the three strategies for slack allocation with varying bias.	91
5.14	CDF of time until first synchronization in the voting system under static optimal (SO) and predictive treaty (PT) strategies for slack allocation with 4 and 8 stations.	92
5.15	99th percentile vote and query latencies per second of 2-station adaptivity test.	93
5.16	99th percentile vote and query latencies for four-station adaptivity test with and without hierarchical treaties.	94
5.17	Using synchronizations to compare Babcock and Olston's top- k algorithm with using predictive treaties with four servers over 24 hours of page hits.	96
5.18	Using synchronizations to compare Babcock and Olston's top- k algorithm with using predictive treaties with 33 servers over one hour of page hits.	96

5.19	CDF of latencies for TPC-C NewOrder transactions, run on two sharded stores with geographic round-trip latency, with 50% of orders going to hot items uniformly across sites.	100
5.20	CDF of latencies for TPC-C NewOrder transactions with skewed order distribution across two and five replicas using lazy balancing (LB) and treaties (T).	101

CHAPTER 1

INTRODUCTION

Modern globally accessible applications are built to run in the cloud, operating on many compute and storage nodes distributed across the globe. Cloud computing allows applications to provide highly available, low-latency services to users across the globe. These services have transformed the way we trade, communicate, and learn about the world.

In this distributed setting, programmers encounter a tension between making an application performant and keeping the application’s implementation simple. The range of different platforms that programmers can choose among bears witness to this tension. Systems improve performance using *replication*, keeping multiple distributed copies of data, and *sharding*, allocating subsets of the data stored in the system to different locations. Replication and sharding introduces a question regarding the system or application’s *consistency*—how and when will operations on data be performed by the system and later observed by other operations [100]. Strong consistency guarantees simplifies the programmer’s task of determining if their application will behave as intended. Programmers to write their programs without thinking about how their data is replicated and sharded—interacting with data as though it was all on a single local machine. Weaker consistency guarantees allow the system to deviate from this simple model.

Many systems to weaken consistency in order to achieve greater scalability. However, strong consistency is critical when lives or money are at stake. Inconsistent behavior can be frustrating, dangerous, or unacceptable in application settings with serious impact on people’s lives—settings such as medicine, banking, politics, and the military. Even in settings with lower stakes, users of weakly consistent systems may be confused by applications that appear buggy. Moreover, weak con-

sistency can significantly complicate the job of developers who try to detect and repair inconsistencies at the application layer. Consistency failures at the bottom of a software stack can percolate up through the stack and cause mysterious behavior at higher layers, requiring defensive programming.

Strictly serializable transactions provide a general, strongly consistent interface for programming distributed applications [78, 90]. Transactions specify atomic groups of operations that are all-or-nothing—either all or none of a transaction’s effects are visible to other transactions. Strict serializability is a strong consistency guarantee that requires that transactions behave as if they all run in a simple serial order, one at a time, while respecting the order in which operations ran in real time (in other words, if one operation completes before another operation begins, the serialized behavior must reflect this ordering). This programming interface is a powerful tool for application designers: it provides an easy-to-understand model for the behavior of concurrent operations with arbitrary atomic operations.

Unfortunately, traditional methods for providing strictly serializable transactions are centered on checking data accessed by a transaction and coordinating with stores to ensure that data is simultaneously consistent at all sites. These checks require synchronously communicating with nodes that store this data to ensure that the transaction’s *view* was consistent before it takes effect, guaranteeing that transactions behave consistently. Synchronous communication for these checks takes a long time in geodistributed settings, increasing transaction latency. Furthermore, it requires blocking potentially conflicting concurrent operations from completing in the meantime, reducing the overall throughput of the system.

Communication delays motivate optimistic concurrency control (OCC), a classic technique for improving performance of transactions. In OCC, the stored data is *not* locked by clients before the data is used; instead, the client logs the values

read during computation. In distributed settings, this behavior allows clients to use local, optimistically cached copies of the persistent data. After the transaction’s computation is complete, clients contact the stores to validate the values used were consistent before applying updates to the stored state [31, 56, 2]. Unlike traditional pessimistic techniques which require synchronizing earlier in a transaction, potentially multiple times, OCC allows transactions to batch synchronizing operations into a single final validation message before committing.

This dissertation explores *warranties*, an abstraction which helps further avoid synchronizing communication overheads in strongly consistent distributed transaction systems. A warranty is a time-dependent guarantee that a predicate holds on a distributed system’s state:

$$\overbrace{\phi(s, t)}^{\text{predicate}} \text{ until } \overbrace{t_{\text{expiry}}}^{\text{time limit}}$$

with a predicate ϕ over the system’s current state s and the current time t that is guaranteed by the system until the time t_{expiry} . Warranty predicates can express simple conditions, such as an object’s value matching the client’s cached copy, or more complex, application-specific conditions, such as whether a company’s total stock across all warehouses is sufficient to fill a customer’s order. Clients and stores can use warranties to avoid synchronizing with remote nodes to determine if the asserted conditions consistently hold in the system’s current state.

1.1 Beyond Optimism

The simplest form of warranty predicate is $\mathbf{x} == \mathbf{v}$, guaranteeing an object \mathbf{x} has value \mathbf{v} . These *state warranties*, discussed in Chapter 3, allow the system to go a step beyond optimistic caching to avoid synchronization. Clients can use warranties guaranteeing cached values are consistent to *know* that their transactions were performed with a consistent view of the system state. When the state observed

is guaranteed to be consistent, there is no need synchronous communication to validate the observations. Furthermore, because the guarantee is time-limited, the system does not need to incur overheads to change the guarantee; the system only needs to wait until the guarantee expires.

State warranties can dramatically improve performance of a system by reducing or, in some cases, entirely avoiding synchronization overheads. This is particularly true in the case of high read contention, where many clients want to share the same popular—yet mutable—data. Such cases are common in modern applications. For example, a Twitter user’s display name is read by other users on the service much more often than it is updated by the user.

1.2 Predicates to Avoid Contention

Some data is updated too frequently to benefit from state warranties. Fortunately, applications often don’t care about *specific* values: many operations perform reads only to compute more abstract predicates over the system’s state.

For example, in the case of a banking service, withdrawal operations only need to check that the current available balance is greater than the amount being withdrawn. In other words, a withdrawal does not need to read the exact value but a more abstract read of the *predicate* on the system’s state $\mathbf{balance} \geq \mathbf{amt}$. The value of this predicate is often *much* more stable than the balance itself, and can be warranted for much longer than the underlying state.

Computation warranties, discussed in Chapter 4, generalize the benefits of state warranties to more general predicates over the system state checked within an application. Reading computation warranty guarantees in place of performing the computations directly acts as a form of memoization that avoids synchronous communication. Computation warranties continue to allow transactions to be strictly

serializable, although in a way that is more in line with a *black-box* view of transaction behavior that helps avoid contention.

1.3 Compositional Predicates for Low-Overhead Enforcement

Since computation warranties can represent arbitrary read-only computations over system state, they can be constructed and enforced using other warranties. This *compositionality* helps to reduce overhead by making update checks *incremental*, allowing update checks to stop at the simplest affected subexpression. When all of a computation’s subexpressions are unaffected, the computation is guaranteed to also be unaffected. Thus, we can efficiently determine when updates to values used by the computation do not conflict with a warranted result, without explicitly recomputing the entire result to check each update.

Furthermore, as demonstrated by treaties in the homeostasis protocol by Roy et al. [87], compositionality helps to divide enforcement responsibilities for predicates over a distributed subset of the system state. This division can help to avoid synchronization during update checks, limiting most checks to local state before checking a distributed statement. Treaties as envisioned by Roy et al. were not time-dependent and were limited to cases of a single layer of composition. However, they can be viewed as a form of warranty focused on enforcement overheads. In Chapter 5, we generalize the design of treaties to support time-dependent statements and arbitrary composition with *predictive treaties*. Arbitrary composition helps to keep synchronization costs low by attempting to restrict synchronization to relatively nearby subsets of nodes.

1.4 Capturing Trends with Time-Varying Predicates

The computation warranties design discussed in Chapter 4 is optimized for read-only computations over data stored at a single node in the system. Unfortunately, checking computations over distributed data may require synchronizing with stores. In geodistributed settings with high latencies, this creates a large overhead for update checks. Roy et al. [87] demonstrated that an application can avoid these synchronizations by carefully selecting the subpredicates used to enforce a distributed predicate.

Predictive treaties, presented in Chapter 5, generalize the design of treaties to further reduce the frequency and overheads of synchronization for enforcement. Predictive treaties are enforced by the system by composing subpredicates that, according to a prediction model, are expected to avoid synchronization costs as much as possible. This prediction model is constructed by estimating update behavior based on low-overhead tracking of past update behavior.

Furthermore, predictive treaties support predicates that *vary with time*. This allows the automatically chosen subpredicates to last even longer by asserting statements that talk about bounds on the *trends* in data rather than the values themselves. When update trends on the underlying local data are steady, distributed predicates over the data remain (and become increasingly) stable. As demonstrated in Chapter 5 this entirely avoids synchronization to enforce some predicates unlike earlier techniques with static predicates.

1.5 The Warranty Design Space

This dissertation explores various elements of how warranties can be designed, building on prior work in the space. To help clarify the relationship between

various warranty-like abstractions developed both in this dissertation and in prior work, we have identified four key aspects: consistency, revocability, scope of the predicate, and how the abstraction can depend on time. To be clear, these features are *not* a complete list of features that distinguish various designs; they outline major distinctions between various types of warranties. These aspects affect how clients use warranties and how the system manages them.

1.5.1 Consistency

Warranties are a form of generalized caching—clients use locally stored warranties providing assertions on remote data to avoid blocking to fetch and validate the data. The consistency guarantees of the warranty’s assertion affects how they can be used to support various application consistency goals. *Strongly* consistent guarantees, like the warranties presented in this dissertation, are easy to use in a way that supports strong consistency for applications relying on them.

In contrast, *weaker* consistency guarantees trade off the overheads required to ensure a guarantee is consistent with how much work the client needs to do to achieve consistency using the guarantee. Using optimistically cached values that do not guarantee strong consistency requires clients to validate their guarantees to ensure consistency. However, weaker consistency does not require the system to perform complicated protocols to keep cache entries consistent.

1.5.2 Revocability

Another key design element is who can use a warranty and how the system handles updates that invalidate current warranties. Warranties can be classified into one of two categories with respect to this feature: *leased* or *public*.

Leased warranties, such as leases [40], promises [51], treaties [87], and predictive treaties [69], limit which nodes are allowed to use them. *Public warranties*, like those discussed in Chapters 3 and 4, allow any node in the system to use them.

The difference between leased warranties and public warranties is a tradeoff between retraction and sharing. Leased warranties are simple to retract, all nodes authorized to use the warranty can be notified by a change to the guarantee, but are difficult to share, each time the warranty is shared with a new node, the node has to be registered as a user of that warranty. Public warranties are easily shared, any node that has a copy of the warranty can use it, but difficult to retract, the system would have to ensure that *all* nodes are notified of a change to the guarantee, infeasible in systems with a massive or unknown number of nodes.

1.5.3 Predicate Scope

Warranty-like abstractions are often optimized for particular classes of predicates based on the scope of their statements. These abstractions exhibit three levels of predicate scope: *object-*, *store-*, or *system-*scope.

Object-scope warranties are the most common type and simplest for a system to enforce. This describes traditional leases [40] and state warranties [66] discussed in Chapter 3. Although most object-scope warranties are focused on guarantees about the exact value, object-scope warranties also include statements about bounds or other features of a single value, such as the bounds enforced by escrow transactions [76].

Store-scope warranties are more general than objects, covering state of objects located on a single storage node in the system. Examples of store-scope warranties include volume leases [110], promises [51], application caches such as memcached [33] or TxCache [82], and the computation warranties [66] discussed

in Chapter 4. Store-scope warranties allow for guarantees that are stable despite updates to the underlying data and allows applications to check store-wide statements. Enforcing store-scope warranties does not require synchronization because all of the data used by a predicate is located on a single node that can check updates locally.

System-scope warranties, such as treaties [87] or predictive treaties [69], support predicates over data located anywhere in the system. This general abstraction comes at the cost of potential synchronization costs for enforcing the guarantees. Thus it is crucial for performance to ensure that the distributed guarantees are enforced by long-lasting store-scoped warranties. Chapter 5 discusses how predictive treaties are designed to avoid these enforcement overheads.

1.5.4 Time Dependence

Finally, warranties vary in how they may depend on time. For example, many designs have *no* time dependence, like the original treaties design [87] or application caches like TxCache [82]. Many designs are *time-limited*, which limit how long a system must enforce a guarantee, as in various types of leases [40, 110, 67, 104] and the warranties discussed in Chapters 3 and 4. Chapter 5 demonstrates how warranties can be *time-varying*, using time in the predicate statement. Time-varying warranties can express guarantees about *trends* in the system state.

1.6 Dissertation Outline

This dissertation is organized as follows. First, the system model is presented for the designs discussed (Chapter 2). In Chapters 3 and 4, public warranties are presented based on work published at NSDI in 2014 with Jed Liu, Owen Arden,

Mike George, and Andrew Myers [66]. In Chapter 3, we focus on public state warranties, discussing how they improve performance for applications by avoiding communication for read validations. In Chapter 4, the design is generalized computation warranties that support arbitrary predicates on the system state. Next predictive treaties, leased warranties supporting time-varying predicates and designed to avoid synchronization during enforcement, are presented in Chapter 5 based on work published at EuroSys 2019 with Jed Liu, Nate Foster, Johannes Gehrke, and Andrew Myers [69]. Finally, in Chapter 6, conclusions and possible directions for future work are discussed.

CHAPTER 2

SYSTEM MODEL

We assume a geodistributed system in which each node serves one of two main roles: *client nodes* perform computations locally using persistent data from elsewhere, and *persistent storage nodes (stores)* store *primary copies*¹ of persistent data. For example, the lower two tiers of the traditional three-tier web application match this description: clients are application servers and stores are database servers. Client nodes' computations are organized into a series of *transactions*, atomic groups of operations that when *committed* are reflected by the stores. These transactions are performed using optimistic concurrency control (OCC) [31, 56, 2]—the primary copies on the stores are *not* locked by the client during computation; instead, the client logs the values read during the computation and afterward validates these values before updating persistent data at the stores.

In practice, some machines may serve both roles, acting as both clients and stores. Furthermore, a store may be implemented by a machine or by a set of machines, possibly replicated across an availability zone² for high availability. We abstract from such implementation details of a store and just treat it as a single storage node.

2.1 Strong Consistency

Warranties are intended to provide a simple programming model for application programmers, offering strong consistency so programmers do not need to reason about inconsistent or out-of-date state. In particular, the system should provide

¹As opposed to cached and otherwise weakly replicated copies of the data clients can use during computation.

²A group of data centers with low latency between them.

strict serializability [90], so each committed transaction acts as though it executes atomically and in logical isolation from the rest of the system. Linearizability and strict serializability strengthen serializability [78, 17]—transactions are performed in a way that is equivalent to a serial schedule [31]—with an analogue of external consistency [38]—the serial schedule is consistent with the *external schedule*, the order they are performed by the client.³ Strict serializability is equivalent to providing linearizability [46] where the object is the entirety of the system’s persisted data and transactions are the concurrent operations on that object. Thus, unlike some prior work (e.g., [14, 13]) that only enforces notions of consistency defined by programmer-specified invariants, we assume that the underlying system offers strong consistency for all data by default. This ensures that strong consistency is the *default* guarantee for applications that do not express more specific consistency requirements.

We refer to the system state as seen by committed transactions as the *current system state*; it is the set of object values that result from executing previously committed transactions in the serialization order guaranteed by strict serializability. In a running transaction that has not yet committed, an object may take on a new value that is not yet visible to other transactions, and this object value may be cached at the client(s) performing the transaction. Once the transaction is committed, the new object value is updated at the object’s store and becomes part of the current system state.

³This definition of external consistency is a bit less restrictive than Gifford’s definition of external consistency that orders overlapping transactions by the time they are completed. We do not require a particular ordering on transactions whose start and end times overlap, as in the formal definition of linearizability [46].

2.2 Optimistic Concurrency Control

In a distributed transaction system using OCC (e.g., Gemstone [70], Thor [62], Fabric [65]), clients fetch and cache persistent objects across transactions. Optimistic caching allows client transactions to largely avoid talking to stores until commit time, unlike with pessimistic locking. The system is faster because persistent data is replicated at the memories of potentially many client nodes. However, care must be taken to avoid inconsistency among the cached copies.

To provide strong consistency, OCC logs reads and writes to objects. As part of committing the transaction, clients send the transaction log to stores involved in the transaction. The stores then check that the state of each object read or modifies matches that in the store (typically by checking version numbers) before applying updates.

These *read and write validations* can turn stores hosting very popular objects into bottlenecks—all clients using an object must contact and check their cached version against the store’s primary copy before committing. This is a fundamental limit on scalability of traditional OCC, so a benefit of warranties is addressing this bottleneck.

OCC is a reasonably popular technique for running distributed transactions in industry. For example, a partially successful attempt at such a programming model is the Java Persistence API (JPA) [22], which provides an object–relational mapping (ORM) that translates accesses to language-level objects into accesses to underlying database rows. JPA implementations such as Hibernate [48] and EclipseLink [30] are widely used to build web applications. However, we want to improve on both the consistency and performance of JPA. Optimism has become increasingly popular for JPA applications, where the best performance is usually achieved through an “optimistic locking” mode that, in many implementations of

JPA, provides snapshot isolation, a relatively strong consistency guarantee that is weaker than strict serializability.⁴

2.3 Transactions Across Multiple Stores

To scale up a distributed transaction system, it is important to be able to add storage nodes across which persistent data and client requests can be distributed.

Data can be distributed across stores in two ways: sharding and replication. Sharding data across a distributed system partitions the data into subsets of the data called shards which are stored on separate stores. In contrast, a system using replication stores copies of the data on multiple stores. These two techniques can be combined in various ways: for example, you can shard the data between different data centers and then replicate the data in each shard across multiple nodes in a given data center. In this work, we will be focused on sharding and treating replication as an orthogonal or future consideration, unless otherwise specified.

As long as a given client transaction accesses data at just one store, and load is balanced across the stores, the system scales well: each transaction can be committed with just one round trip between the client and the accessed store.

In general, however, transactions may need to access information located at multiple stores. For example, consider a web shopping application. A transaction that updates the user’s shopping cart may still need to read information shared among many users of the system, such as details of the item purchased.

Accessing multiple stores hurts scalability. To ensure strict serializability, all data accessed during the transaction must be consistent with the stored primary

⁴The JPA 2 specification only guarantees snapshot isolation because it only guarantees that objects *written* by a transaction are up to date—but, unfortunately, not the objects *read* unless explicitly locked. Implementations differ in interpretation, however. When using implementations that only provide snapshot isolation, the programmer can sometimes turn on optional extensions or otherwise perform workarounds to ensure strict serializability.

copies. Clients performing a distributed transaction run a two-phase commit (2PC), coordinating with multiple stores to ensure strict serializability [17]. In the first phase (the prepare phase), each store performs read and write validations⁵ to check if the transaction can be committed and if so, readies the updates to be committed; it then reports to the coordinator whether the transaction is serializable. If the transaction’s effect is determined to be consistent by every store, all stores are told to commit in the commit phase. Otherwise, the transaction is aborted and its effects are rolled back.

2.4 Clock Synchronization

Both warranties and predictive treaties assume that system nodes maintain loosely synchronized clocks that agree with only limited precision. This assumption is reasonable; the accuracy of clock synchronization offered by older protocols such as NTP [75] and Marzullo’s algorithm [71] already suffices for the results presented in this work. In fact, recent work has shown that clocks can be kept synchronized with much greater precision and with failure rates that are lower than a host of other more serious failures such as bad CPUs [25, 60, 36, 91].

⁵Often we’ll refer to read and write validations performed during the prepare phase as *read and write prepares*.

CHAPTER 3

PUBLIC WARRANTIES

The need for strong consistency and a simple programming model has kept traditional databases with ACID transactions, such as Postgres [97], in business and motivates modern transaction systems such as Google’s Spanner [25] and CockroachDB [106]. However, transactions are traditionally considered to have poor performance, especially in a distributed setting. In this work, we introduce *public warranties*, a new mechanism that improves the performance of transactions, enabling them to scale better both with the number of application clients and with the number of persistent storage nodes. Warranties help avoid the unfortunate choice between consistency and performance.

A warranty is a time-dependent assertion about state issued by the system: it is guaranteed to remain true (*active*) for the warranty’s *term*, a fixed period of time. At the end of its term, the warranty *expires* and is no longer guaranteed to be true. Times appearing in the warranties are measured by the clock of the store that issued the warranty. As discussed in Chapter 2, I assume that clocks at nodes are loosely synchronized using a clock synchronization protocol such as NTP [75].

In the next two chapters, we focus on *public* warranties. Public warranties, unlike *leased* warranties, may be used and distributed by any node in the system—public warranties do not require the system to register and track nodes using the assertion. This novel design trades flexibility to retract the assertion early to enable wide distribution and usage among clients in the system.¹

The simplest form of warranty is a *state warranty*, an assertion that the concrete

¹In the original publication that introduced warranties, published at NSDI 2014 [66], the simple term *warranty* referred to what this dissertation calls *public warranties*. In this dissertation we use the term *warranty* to discuss features that apply to both the public and leased designs and otherwise will use the more specific terms when the discussion only applies to a particular variant.

state of an object has a particular value. State warranties improve scalability by eliminating the work needed for read preparing the warranted object. For example, a state warranty for an object representing a bank account might be

```
acct = {name = 'John Doe', bal = 20345} until 1364412767
```

Here, the state warranty specifies the state of the object **acct**, with fields **name** and **bal**, and the time marking the end of the warranty's term, **1364412767**. Client nodes can use this state warranty to read the fields of **acct** in a transaction. If that transaction prepares before time **1364412767**, the client does not need to read-prepare the version of **acct** seen; the state warranty guarantees the value is consistent with the store's copy.

In this chapter, I focus on public state warranties. Warranties generalize OCC (Section 3.1) and read-leases (Section 3.2). Clients request warranties from stores which issue them on demand (Section 3.3). Public warranties are distributed throughout the system to clients that need them (Section 3.4). Updates to the system are prevented from invalidating public warranties (Section 3.5), with implications for performance (Section 3.6). The traditional 2PC protocol for distributed transaction can be modified to take advantage of warranties to avoid round trips, improving performance (Section 3.8). Experimental evaluation using standard benchmarks and a real application for managing university courses demonstrates that public warranties improve performance for distributed transactions (Sections 3.9 and 3.10).

3.1 Warranties as Optimistic Concurrency Control

By making guarantees about the state of the system, warranties allow transactions to be committed without preparing reads against the objects covered by warranties. When all reads to a store involved in a transaction are covered by warranties, stores

need not be contacted for validation. Consequently, as we discuss in Section 3.8, two-phase commit can be reduced to a one-phase commit in which the prepare and commit phases are consolidated, or even to a zero-phase commit in which no store need be contacted. The result is significantly improved performance and scalability.

If a warranty expires before the transaction commits, the warranty may continue to be *valid*, meaning that the assertion it contains is still true even though clients cannot rely on its remaining true. Clients can, however, still use the warranty optimistically and check at commit time that the warranty remains valid.

Thus, state warranties generalize optimistic concurrency control. Traditional optimistic concurrency control equates to always receiving a zero-length warranty for the state of the object read, and using that expired warranty optimistically.

3.2 Warranties as Generalized Read Leases

Leases [40, 39] have been used in many systems (e.g., [103, 3]) to improve performance. Warranties exploit the key insight of leases: time-limited guarantees increase scalability by reducing coordination overhead for managing access to shared objects. As defined originally by Gray and Cheriton, leases confer time-limited *rights* to access objects in certain ways, and must be held by clients in order to perform the corresponding access. Conversely, warranties are time-limited *assertions* about what is *true* in the distributed system, and are not, therefore, rights conferred to a particular set of nodes.

Read leases are state warranties, time-dependent assertions on the state of a single object. Since *read leases* on objects effectively prevent modifying object state, they enforce assertions regarding the state of that data.

While read leases are, for all intents and purposes, a class of warranties, there is

a fundamental difference between the lease and warranty perspectives. The value of the warranty (assertion) perspective is that state warranties naturally generalize to expressive assertions over state—in particular, computation warranties that specify the results of application-defined computations over the state of potentially many objects, discussed in Chapter 4. While there has been work on volume leases for groups of objects [110], these leases treat the group as though it were an opaque single object and do not support predicates like computation warranties. By supporting predicates, computation warranties do not delay updates that do not change the predicate’s result—warranties help separate the *features* of the state used by the application from the particular *values* determining these features.

3.3 Issuing State Warranties

As clients perform transactions, they fetch objects they do not have locally cached and read-prepare objects for which they do not have active warranties. State warranties are requested automatically when objects are either fetched or read-prepared by a client. When a store issues a warranty, the warranty’s term is set to appropriately balance performance trade-offs. Stores track issued warranties until the end of their terms so the store can defend against invalidating updates as discussed in Section 3.5.

3.4 Distributing Public Warranties

Public warranties can be used regardless of how they get to clients and can be shared among any number of clients without contacting the store. Therefore, a variety of mechanisms can be used to distribute public warranties to clients.

Clients may directly query stores for warranties. However, the system can avoid

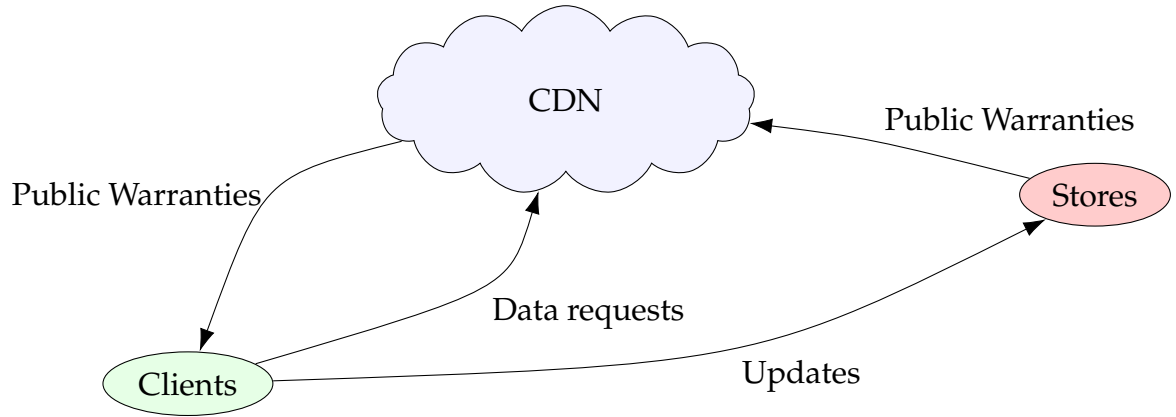


Figure 3.1: Public warranty distribution architecture. Public warranties can be distributed through a CDN to reduce load on stores.

load on the store by using a content distribution network (CDN) to serve public warranty queries as shown in Figure 3.1.

Going a step further, applications can *subscribe* to warranties that match a given pattern when requesting them. Stores automatically *refresh* warranties with longer terms before the original term expire, pushing these extended warranties either directly to clients or into the CDN. The CDN does not require a complicated update procedure to ensure consistency for public warranties because public warranties are *irrevocable*—if clients read an old value from the CDN before the refreshed value is propagated, consistency is not violated, the public warranty’s original term is still enforced. Warranty refresh makes it feasible to satisfy client requests with shorter warranty terms, consequently reducing write latency.

This design differs from using direct replication, a separate strategy used in many distributed storage systems to achieve high availability, low latency, and durability. Those three goals are handled separately here. Distributing public warranties through a CDN makes data objects highly available with low latency, without damaging consistency. Because the authoritative copies of objects are located at stores, a write to an object requires a round trip to its store; the latency this introduces is ameliorated by the support for relatively large transactions, in

which communication with stores tends to happen at the end of transactions rather than throughout. This CDN design does not inherently achieve high durability, this can be done by having stores replicate the data durably across multiple machines. To avoid the latency and throughput problems of geodistributed replication, these stores can keep replicas on a set of local machines rather than replicating across the entire system.

3.5 Defending Public Warranties

Transactions may try to perform updates that affect objects on which active warranties have been issued. Updates that invalidate active warranties would violate transactional isolation and consistency guarantees for clients using those warranties. Therefore, stores must defend warranties against invalidating updates.

A public warranty can be defended against an invalidating update transaction in two ways: the transaction can either be rejected or delayed. If rejected, the transaction will abort and the client must retry it. If delayed, the updating transaction waits to prepare until it can be safely serialized or is aborted by the client due to a failed prepare at another store. Rejecting the transaction does not solve the underlying problem of warranty invalidation, so delaying is typically the better strategy if the goal is to commit the update. To prevent write starvation, the store stops issuing new warranties on the affected state until after the commit. The update also shortens the term of subsequent warranties anticipating future writes, as projected by the estimation model discussed in Section 3.7.

Leased warranties can be enforced in a third way: *retraction*. Before the invalidating update is applied, the system notifies all holders of the warranty that it is no longer valid and then stops enforcing the warranty. This requires the store to track all nodes that currently hold the warranty, which creates an overhead that grows

with the number of nodes that hold the warranty. Although public warranties can, in theory, be retracted by similarly contacting *all* nodes, this is practically infeasible in many distributed systems where the set of clients is enormous or unknown.

3.6 Performance Trade-offs

Warranties improve read performance for the warranted objects, but require new overheads for writes to these objects. Such a trade-off appears to be an unavoidable when providing strict serializability. For example, in conventional database systems that use pessimistic locking to enforce consistency, readers are guaranteed to observe consistent states, but update transactions must wait until all read transactions have completed and released their locks. With many simultaneous readers, writers can be significantly delayed. Thus, warranties occupy a middle ground between optimism and pessimism, using time as a way to reduce the coordination overhead incurred with locking.

The key to good performance, then, is to issue warranties that are long enough to benefit readers, avoiding read prepares, but not so long that they delay writers noticeably. If there is no suitable term for a warranty that balances these concerns, the store should not issue a warranty. In Section 3.7, we discuss how a simple model with low-overhead estimates for setting warranty terms appropriately.

For applications that require both high write throughput and high read throughput to the same object, using replication is essential to scale the system. The cost of keeping replicas consistent makes it difficult to provide strict serializability with good performance. If weaker consistency guarantees are acceptable, however, there is a simple workaround: keeping weakly consistent replicas of the object by explicitly maintaining the state in multiple distinct strongly consistent objects. Writes can go to one or more of these distinct objects that are read infrequently, period-

ically propagating these updates (possibly after reconciliation of divergent states) to a frequently read object for which warranties may be issued. This is a much easier programming task than starting from weak consistency and trying to implement strong consistency where it is needed. The challenging part is reconciliation of divergent replicas, which is typically needed in weakly consistent systems in any case (e.g., [102, 88, 27]).

3.7 Setting Warranty Terms with Workload Estimation

Depending on how warranty terms are set, warranties can either improve or hurt performance. It is usually possible to automatically and adaptively set warranty terms to achieve a performance increase. Warranty terms should be set so the expected benefits to readers and stores outweighs the expected additional overhead to writers.

Warranties improve performance by avoiding read prepares for objects, reducing the load on stores and on the network. If *all* read and write prepares to a particular store can be avoided, warranties eliminate the need even to coordinate with that store.

Warranties can hurt performance primarily by introducing overheads for writes to objects. During a warranty's term, writers are delayed while the system defends the assertion. Longer warranty terms create a wider window during which writes will experience delays. The length of these delays depends on how the warranty is defended. In the case of public warranties, writes are delayed until the end of the term. For leased warranties, which can be retracted, writes are delayed until either the system can retract the warranty or the term ends, whichever occurs first.

Furthermore, excessively long terms may also allow readers to starve writers. The system mitigates this starvation by refusing to issue new warranties while

Table 3.1: Policy parameters for setting a warranty’s term.

Parameter	Description	Units
R	read rate	time ⁻¹
W	write rate	time ⁻¹
L	warranty term	time
L_{max}	max warranty term	time
k_1	max expected writes during term	unitless
k_2	min expected reads during term	unitless

writers are attempting to invalidate an outstanding warranty. Note that with traditional OCC, writers can block readers by causing all read prepares to fail [79]; thus, warranties shift the balance of power away from writers and toward readers, addressing a fundamental problem with OCC.

To find the right balance between the good and bad effects of warranties, we take a dynamic, adaptive approach. Warranty terms are automatically and individually set by stores that store the relevant objects. Fortunately, stores observe enough to estimate whether warranty terms are likely to be profitable. Stores see both read prepares and write prepares. If the object receives many read prepares and few or no write prepares, a state warranty on that object is likely to be profitable. A similar observation applies to computation warranties.

To determine whether to issue a warranty for an object, and the length of its term L if a warranty is issued, the system plugs measurements of object usage into a simple system model. The system measures the rate W of writes to each object, and when no warranty is issued on the object, it also measures the rate R of reads to the object. Both rates are estimated using an exponentially weighted moving average (EWMA) [50] of the intervals between reads and writes.

During a warranty’s term, many read prepares are no longer visible to the store. To account for this, our implementation modifies EWMA to exponentially decay historical read-prepare data during warranty periods. Empirically, this modification improves the accuracy of rate estimation. To lower the storage overhead of

monitoring, unpopular objects are flagged and given lower-cost monitoring as long as they remain unpopular.

To ensure that the expected number of writes delayed by a warranty is bounded by a constant $k_1 < 1$ that controls the trade-off between read and write transactions. The warranty term is set to k_1/W with a maximum warranty L_{max} used to bound write delays. Our goal is that warranties are profitable: they should remove load from the store, improving scalability. A public warranty eliminates roughly RL read prepares over its term L , but adds the cost of issuing the warranty and some added cost for each write that occurs during the term. The savings of issuing a warranty is positive if each write to an object is observed by at least k_2 reads for some value k_2 , giving us a condition $RL \geq k_2$ that must be satisfied in order to issue a warranty. The value for constant k_2 can be derived analytically using measurements of the various costs, or set empirically to optimize performance.

The tension between write latency and read throughput can also be eased by using warranty refresh. The term L is computed as above, but warranties are issued to clients with a shorter term corresponding to the maximum acceptable update latency. The issuing store proactively refreshes each such warranty when it is about to expire, so the warranty stays valid at clients throughout its term.

3.8 Using Warranties in Transactions

Warranties improve the performance of OCC by reducing the work required for two-phase commit [17] by avoiding read prepares and, in some cases, allowing prepare phases to be eliminated entirely.

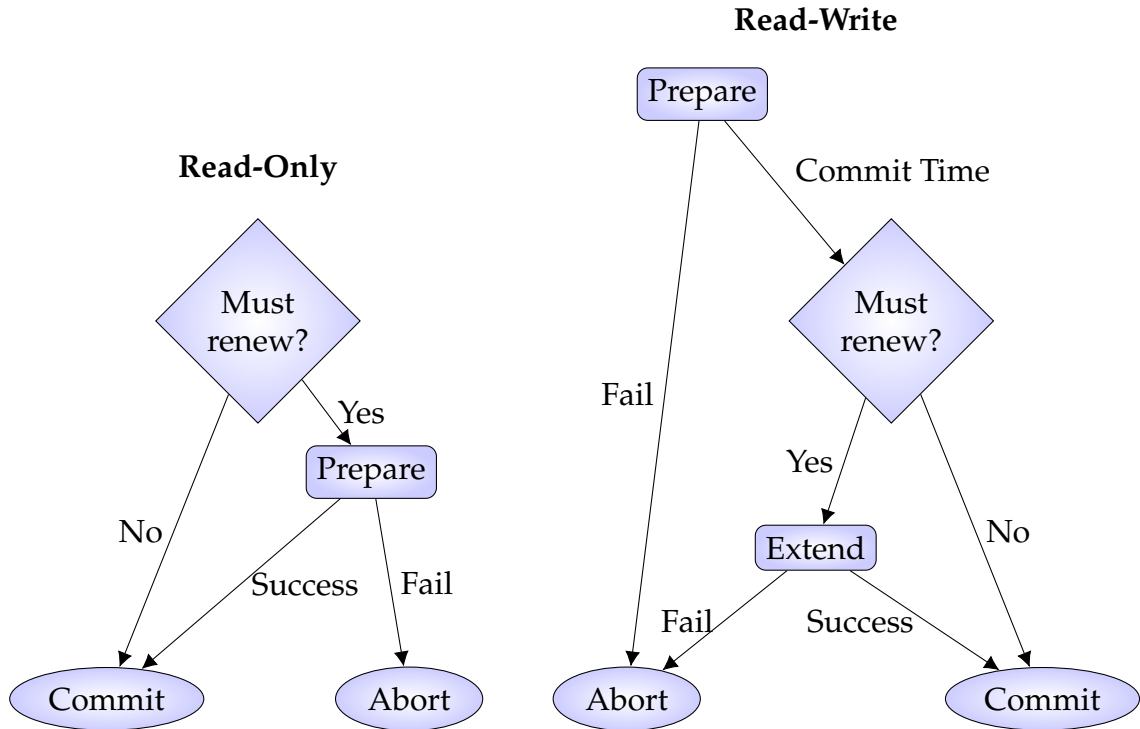


Figure 3.2: The warranty commit protocol.

3.8.1 The Warranty Commit Protocol

When a transaction completes, the client performs a modified two-phase commit, illustrated in Figure 3.2 for both read-only and read-write transactions. In the prepare phase, the client sends the write set of the transaction (if any), along with any warranties in the read set whose term has expired. If all warranties in the read set can be renewed, the transaction may commit. Since outstanding warranties may cause the updates to be delayed, the store responds with a *commit time* indicating when the commit may be applied successfully.

When the client receives a commit time from all stores, it checks that the terms of the warranties it holds exceed the maximum commit time. If not, it attempts to renew these warranties beyond the commit time in an additional *extend* phase. If active warranties are obtained for all dependencies, the client sends the commit message, and the stores commit the updates at the specified time.

Table 3.2: Round trips performed during the commit protocol with Warranties vs. traditional OCC. Warranties require fewer round trips than traditional OCC in highlighted cases.

Stores contacted	Stores written	Warrantied?	Round Trips:	
			Warranties	OCC
1+	0	Y	0	1
1+	0	N	1	1
1	1	Y/N	1	1
2+	1	Y	1	2
2+	1	N	2	2
2+	2+	Y	2	2
2+	2+	N	3	2

3.8.2 Avoiding Protocol Phases

While a two-phase commit, with two round trips of messages, is required in the general case, performance can be improved by eliminating or combining round trips performed when possible. For read-only transactions, the second round trip of messages for the commit phase is superfluous, and clients executing transactions that involve only one store can combine the prepare and commit messages into one round trip. The optimizations to 2PC that warranties make possible are summarized in Table 3.2.

The read-only (rows 1–2) and single-store optimizations (row 3) are available with or without warranties. However, unexpired warranties enable eliminating additional round trips, shown by the two rows highlighted in gray.

Row 1 shows that read-only transactions whose read set is covered by active warranties may commit without communicating with stores—a zero-phase commit. This optimization matters because for read-biased workloads, most transactions will be read-only.

Row 4 shows that transactions that read from multiple stores but write to only one store may commit after a single round trip of messages if their read set is warrantied on all other stores. This single-phase optimization pays off if objects

are stored in such a way that writes are localized to a single store. For example, if a user’s information is located on a single store, transactions that update only that information will be able to exploit this optimization.

While warranties usually help performance, they do not strictly reduce the number of round trips required to commit a transaction. Transactions performing updates to popular data may have their commits delayed. Since the commit time may exceed the expiration time of warranties used in the transaction, the additional *extend* message may be required to renew these warranties beyond the delayed commit time, as shown in the final row.

3.9 Public Warranties Implementation

To evaluate public warranties as a mechanism for improving performance for distributed transaction systems, we extended the Fabric secure distributed object system [65]. Fabric provides a high-level programming model that, like the Java Persistence API, presents persistent data to the programmer as language-level objects. Language-level objects may be both persistent and distributed. It implements strict serializability using OCC.

Fabric also has many security-related features—notably, information flow control—designed to support secure distributed computation and also secure mobile code [8]. The dynamic security enforcement mechanisms of Fabric were not turned off for our evaluation, but they are not germane to this work.

We extended the Fabric system and language to implement the mechanisms described in this dissertation. Our extended version of Fabric supports both public state warranties and public computation warranties. Computation warranties, discussed and evaluated in Chapter 4, were supported by extending the Fabric language with memoized methods. Client (worker) nodes were extended to use public

warranties during computation and to evaluate and request public computation warranties as needed. The Fabric dissemination layer, a CDN, was extended to distribute public warranties and to support public warranty subscriptions. Fabric workers and stores were extended to implement the new transaction commit protocols, and stores were extended to defend and revalidate public warranties.

The previously released version of Fabric (0.2.1) contained roughly 44,000 lines of (non-blank, non-comment) code, including the Fabric compiler and the runtime systems for worker node, store nodes, and dissemination nodes, written in either Java or the Fabric intermediate language. In total, about 6,900 lines of code were added or modified across these various system components to implement warranties.

Fabric ships objects from stores to worker nodes in object groups rather than as individual objects. State warranties are implemented by attaching individual warranties to each object in the group. Issuing public warranties for object groups, similar to volume leases [110], could potentially reduce the overhead of managing warranties; this has been left to future work, however.

Some features of the warranties design have not been implemented; most of these features are expected to improve performance further. The single-store optimization of the commit protocol has been implemented for base Fabric, but rows 3–5 of Table 3.2 were not implemented for warranties. The warranty refresh mechanism was not implemented for these experiments.

3.10 Evaluation

We evaluated public state warranties against existing OCC mechanisms, and other transactional mechanisms, primarily using two programs. First, we used the multiuser OO7 benchmark [23]. Second, we used versions of Cornell’s deployed Course

Management System [19] (CMS) to examine how public warranties perform with real systems under real-world workloads. Both of these programs were ported to Fabric in prior work [65].

3.10.1 Multiuser OO7 Benchmark

The OO7 benchmark was originally designed to model a range of applications typically run using object-oriented databases. The database consists of several modules, which are tree-based data structures in which each leaf of the tree contains a randomly connected graph of 20 objects. In our experiments we used the “SMALL” sized database. Each OO7 transaction performs 10 random traversals on either the *shared* module or a *private* module specific to each client. When the traversal reaches a leaf of the tree, it performs either a read or a write action. These are relatively heavyweight transactions compared to many current benchmarks; each transaction reads about 460 persistent objects and modifies up to 200 of them. By comparison, if implemented in a straightforward way with a key-value store, each transaction would perform hundreds of get and put operations. Transactions in the commonly used TPC-C benchmark are also roughly an order of magnitude smaller [105], and in the YCSB benchmarks [109], smaller still.

Because OO7 transactions are relatively large, and because of the data’s tree structure, OO7 stresses a database’s ability to handle read and write contention. However, since updates only occur at the leaves of the tree, writes are uniformly distributed in the OO7 specification. To better model updates to popular objects, we modified traversals to make read operations at the leaves of the tree exhibit a power-law distribution with $\alpha = 0.7$ [20]. Writes to private objects are also made power-law distributed, but remain uniformly distributed for public objects.

3.10.2 Course Management System

The CS Course Management System [19] (CMS) is a 54k-line Java web application used by the Cornell computer science department to manage course assignments and grading. The production version of the application uses a conventional SQL database; when viewed through the JPA, the persistent data forms an object graph not dissimilar to that of OO7. We modified this application to run on Fabric. To evaluate computation warranties, we memoized a frequently used method that filters the list of courses on an overview page, these results are discussed in Chapter 4.

We obtained a trace from Cornell’s production CMS server from three weeks in 2013, a period that encompassed multiple submission deadlines for several courses. To drive our performance evaluation, we took 10 common action types from the trace. Each transaction in the trace is a complete user request including generation of an HTML web page, so most request types access many objects. Using JMeter [52] as a workload generator, we sampled the traces, transforming query parameters as necessary to map to objects in our test database with a custom JMeter plugin.

3.10.3 Comparing with Hibernate/HSQldb

To provide a credible baseline for performance comparisons, we also ported our implementation of CMS to the Java Persistence API (JPA) [22]. We ran these implementations with the widely used Hibernate implementation of JPA 2, running on top of HyperSQL (HSQldb), a popular in-memory database in **READ COMMITTED** mode. For brevity, we refer to Hibernate/HSQldb as *JPA*. For JPA, we present results only for a single database instance. Even in this single-store setting, and even with Hibernate running in its optimistic locking mode, which does not enforce serializability, Fabric significantly outperforms JPA in all of our experiments.

(Note that JPA in optimistic locking mode is in turn known to outperform JPA with pessimistic locking, on read-biased workloads [96, 32]). This performance comparison aims to show that Fabric is a good baseline for evaluating the performance of transactional workloads: its performance is competitive with other storage frameworks offering a transactional language-level abstraction.

3.10.4 Experimental Setup

Our experiments use a semi-open system model. An open system model is usually considered more realistic [89] and a more appropriate way to evaluate system scalability. Worker nodes execute transactions at exponentially distributed intervals at a specified *average request rate*. Consequently, each worker is usually running many transactions in parallel. Overall system throughput is the total of throughput from all workers. To find the maximum throughput, we increase the average request rate until the target throughput cannot be achieved.

The experiments are run on a Eucalyptus cluster. Each store runs on a virtual machine with a dual core processor and 8 GB of memory. Worker machines are virtual machines with 4 cores and 16 GB of memory. The physical processors are 2.9 GHz Intel Xeon E5-2690 processors.

The parameters k_1 and k_2 (Section 3.7) are set to 0.5 and 2.0, respectively; the maximum public warranty term was 10s. In our experience, performance was not very sensitive to k_1 and k_2 , although this was not rigorously evaluated.

3.10.5 Results

Scalability. We evaluated scalability using the OO7 benchmark with different numbers of stores. A “shared store” was reserved for the assembly hierarchies of all modules. The component parts of the modules were distributed evenly across the

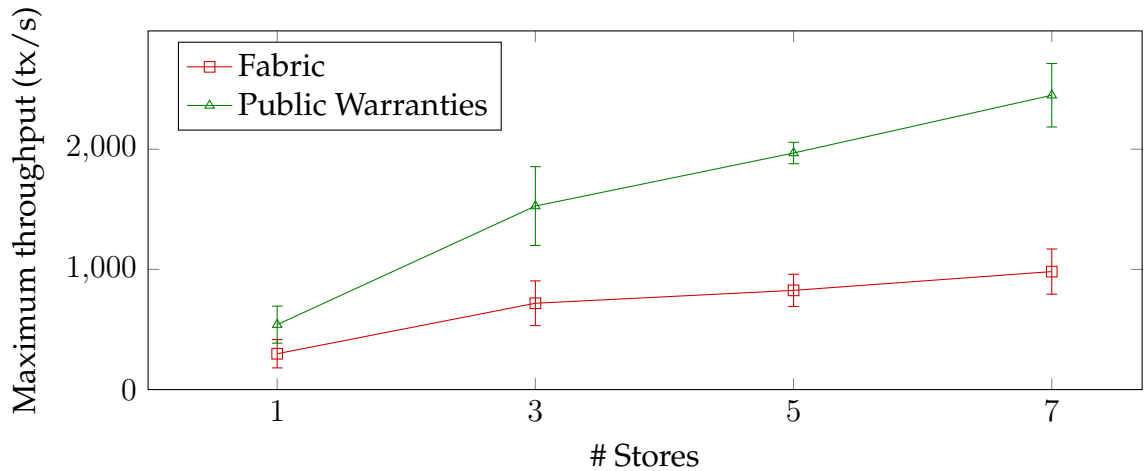


Figure 3.3: OO7 maximum throughput on a 2%-write workload as the number of stores increases. Public warranties allow throughput to scale up with more stores.

remaining stores. Only shared composite parts were placed on the shared store. Results presented are the average of three runs.

Figure 3.3 shows maximum throughput in total transactions committed per second by 36 workers, as the number of stores increases. Error bars show the standard deviation of the measurements across three trials of the configuration. As expected, adding stores has little effect on maximum throughput in base Fabric because the shared store is a bottleneck. Public warranties greatly reduce load on the shared store allowing us to add roughly 400 tx/s per additional store. Note that the plot only counts committed transactions; the percentage of aborted transactions for Fabric at maximum throughput ranges from 2% to 6% as the number of stores increases from 3 to 7; with public warranties, from 4% up to 15%.

Table 3.3 reports on the performance of the CMS application in various configurations with three trials each. The first three rows of Table 3.3 show that Fabric, without or without public warranties, delivers more than an order of magnitude performance improvement over JPA. Although the JPA implementation enforces

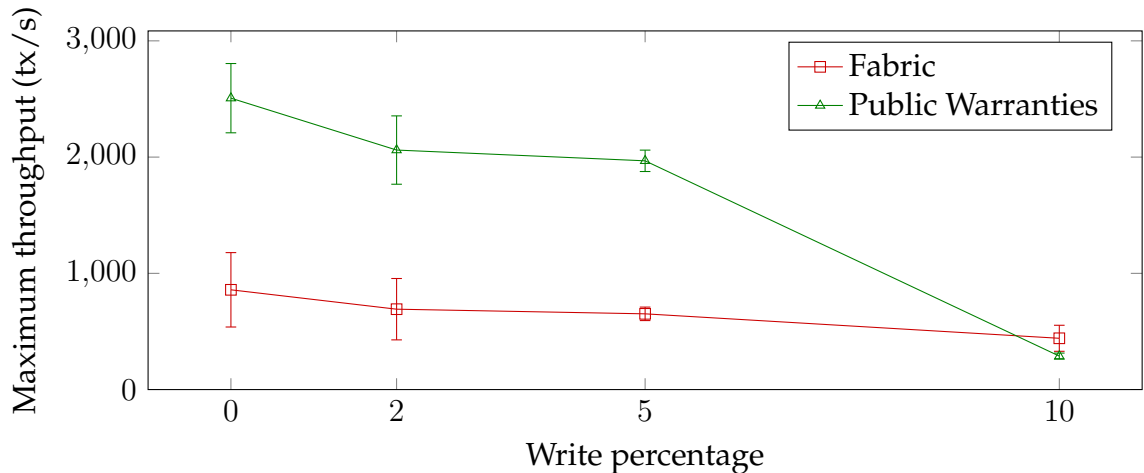


Figure 3.4: Effect of write percentage on OO7 maximum throughput on 3 stores with 24 workers.

Table 3.3: CMS throughput and latency on various systems with public warranties highlighted. Both are averaged over 10s at max throughput across three trials.

System	Stores	Tput (tx/s)	Latency (ms)
JPA	1	72 ± 12	211 ± 44
Fabric	1	3032 ± 144	143 ± 120
Public Warranties	1	4142 ± 112	27 ± 27
Fabric	3	4090 ± 454	311 ± 175
Public Warranties	3	5886 ± 124	35 ± 4

weaker consistency, Fabric’s more precise object invalidation helps performance as contention increases. Public warranties help improve performance further, even in a single-store configuration.

To evaluate how the system scales for a more realistic workload, we also ran CMS with 3 stores using Fabric and public warranties. Two stores each held data for multiple courses, while the third store contained metadata. As Table 3.3 shows, public warranties scale better than Fabric with the additional stores.

Latency. Increases in throughput would be less compelling if they came at the cost of high latency. Table 3.3 also reports the latency measured with the CMS workload on the various systems. Fabric has similar latency with or without public

warranties.

Figure 3.4 shows how the performance of public warranties is affected by the fraction of update transactions. Four different workload mixes were measured using three trials, each having a 94:6 shared-to-private traversal ratio and a 1:10 shared-to-private write ratio. When more than 10% of the transactions are updates, the cost of maintaining and issuing public warranties in the current implementation is too high to obtain a performance improvement. The latencies at some of these throughputs are higher than Fabric's, but still relatively low. At 2% and 5% writes, the latency of public warranties is about 400 ms higher than Fabric's but nearly the same as Fabric's at 0% and 10% writes.

Warranties can result in delaying transactions that are attempting to write to an object that has a state warranty. We call this *write delay*. For all of the runs depicted in Figure 3.4, the median write delay is 0 ms. However, some fraction of transactions are forced to wait until one or more public warranties expire. The more read-biased the transaction, the more frequently this happens. In the 2%-write workload, 70% of read-write transactions see no write delay. In the 10%-write workload, 82% see no write delay. Among those that encounter write delay, the delay is roughly uniformly distributed from 0 up to the max public warranty length.

3.11 Discussion

State warranties can dramatically improve the performance of a distributed transaction system by allowing clients to avoid read validation for state that is guaranteed to be consistent with the store. When all object values read by a transaction are guaranteed consistent with state warranties, a client can skip phases of the traditional two-phase commit protocol, leading to reduced load on stores and greater throughput.

These performance gains come at the cost of delaying updates that would invalidate warranty assertions, however, making state warranties beneficial primarily for objects that are read *much* more frequently than they are updated. In our design, we identify objects that are beneficial to issue public state warranties for using a low-overhead prediction model.

Fortunately, warranties are not limited to simple equalities for object values and can be beneficial even in cases where much of the data is not mostly read. In the next chapter I discuss *computation warranties* which demonstrate how the warranty design can be used to provide *application-level* assertions. Computation warranties can provide further benefits over state warranties by allowing applications to specify transaction behavior in a *black-box* manner. Transactions can use warranted computation results in place of performing the computation, avoiding contention from individual reads and writes that the explicit computation would require.

CHAPTER 4

COMPUTATION WARRANTIES

Computation warranties generalize state warranties to support arbitrary computation on a store's state. A computation warranty is a guarantee until time t of the truth of a logical formula φ , where φ can mention computational results such as the results of method calls. We focus here on the special case of public warranties generated by function calls, where φ has the form $o.f(\vec{x}) = v$ for some object o on which method f is invoked using arguments \vec{x} , producing a value v to be obtained from the warranty. Note that the value returned by f need not be a primitive value. In the general case, it may be a data structure built from both new objects constructed by the method call and preexisting objects.

Computation warranties generated by function calls can naturally be used to perform *memoization* where previously computed results are cached and reused in place of recomputing the result [73]. In distributed applications, it is common to use a distributed cache such as **memcached** [33] to cache previously computed results to be reused by many nodes. For example, web application servers can cache the text of commonly used web pages or content to be included in web pages. Often services like **memcached** are not built to ensure strong consistency of the cached results, and require additional effort or tolerating potentially stale results. Computation warranties can be used to cache such computed results without abandoning strong consistency.

For example, a computation warranty asserting a computation that checks there is at least \$100 in a bank account would be

```
acct.has_at_least(100) = true until 1556037120
```

Here, the account object **acct** has a method **has_at_least** which checks the balance's value against the given lower bound amount. Clients can use this com-

putation warranty in a transaction to check the lower bound on the account rather than explicitly reading the account object's state explicitly. If that transaction prepares before time `1556037120`, the client does not need to compute the result explicitly or read-prepare `acct`. Unlike a state warranty on `acct`, this computation warranty allows updates to `acct` that do not change the result.

Computation warranties improve scalability for three reasons:

1. Similar to state warranties, computation warranties reduce read prepare overheads.
2. Computation warranties enable the consistent distributed memoization of computed results, saving clients from repeating computations.
3. Computation warranties enforce strong consistency without requiring explicit version checks on individual objects the result was computed from. As a result, they help to avoid unnecessary contention between transactions whose application-level behaviors are not in conflict.

In this chapter I discuss how the design in Chapter 3 can be extended to support assertions on computations on stores, focusing on *public* computation warranties. A few example applications where computations warranties can improve scalability and performance are described (Section 4.1). Applications use computation warranties by marking functions for which computed results should be warranted (Section 4.2). Computation warranties are designed to ensure they can be used in place of performing a computation without visibly changing the program's behavior (Section 4.3). Computation warranties are requested by clients before or concurrent with computing a call of a flagged method. If no matching warranty has been issued, clients can propose a new computation warranty after performing the call (Section 4.4). When a client holds a computation warranty, they may use it in place of performing the call (Section 4.5). Computation warranty terms are

set using a generalization of the estimation model used for state warranties (Section 4.6). Computation warranties are defended like state warranties. However, many updates may not affect an asserted result, and therefore the store does not need to delay these transactions. To efficiently check if an update would invalidate a computation warranty, the store *incrementally* checks the effects of an update on warranted subcomputations (Section 4.7). Computation warranties offer improved performance over state warranties in cases where transactions are often reading *stable* results of computation on frequently updated state (Section 4.9).

4.1 Example Applications

In many distributed systems, ensuring that cached computation results are up to date is an involved and error-prone process. Computation warranties make this simple, however, as the computed results are guaranteed to be true during their term and can be constructed *compositionally* to help break up the work in checking potentially invalidating updates. Here are a series of examples where computation warranties may be used to improve performance.

4.1.1 Generated Web Pages

In many web applications, a significant fraction of the work to be done is the computation of the HTML code for the application's initial home page. Often, the home page does not change often even though fresh computation of the home page requires accessing a substantial amount of persistent information. For example, a social media service's home page may show the current top trending topics. The generated home page content, computed using data from various objects, could be cached and distributed with a computation warranty with an assertion like the

following;

```
1 app.home_page() = str
```

Here, the result `str` produced by the warranty is a web page represented as a string or perhaps as an abstract syntax tree. The latter representation would enable issuing warranties for different parts of the page.

In modern web applications, systems often cache generated pages in ways that are weakly consistent and sometimes require manual management. With computation warranties, however, the cached page is guaranteed to be consistent and does not require manual updates.¹

4.1.2 Top N Items

Many applications track and query the top-ranked N items among some large set such as advertisements, product choices, search results, poll candidates, or game ladder rankings. Although the importance of having consistent rankings may vary across applications, there are at least some cases in which the right ranking is important and may have monetary or social impact. Election outcomes matter, product rankings can have a large impact on how money is spent, and game players care about ladder rankings.

To cache the results of such a computation, we might define a computation `top(n, i, j)`, which returns the set `s` of the `n` top-ranked items whose indices in an array of items lie between indices `i` and `j`. A computation warranty with an assertion of the form `s = top(n, first, last)` then allows clients to share the computation of the top-ranked items within the full range of indices.

¹Of course, there updates to the content on the homepage may be delayed or blocked to ensure consistency. If blocking invalidating updates is unacceptable, treaties (Chapter 5) or some other leased warranty may be more appropriate than the public warranty design evaluated in this chapter, although this limits how widely the results can be shared.

The `top` function has index arguments `i` and `j` to permit `top` to be implemented recursively and efficiently using results from subranges, on which further warranties are issued. We discuss later in more detail how this approach allows computation warranties to be updated and recomputed efficiently in Section 4.7.

4.1.3 Searching for Airline Seats

In an online booking application, clients are likely to view many flights before purchasing their tickets. Thus flights are viewed much more often than their seating is updated. In this scenario, reducing read-prepare overheads helps improve scalability.

Efficient searching over suitable flights can be supported by issuing warranties guaranteeing that at least a certain number of seats of a specified type are available; for a suitable constant number of seats n large enough to make the purchase, a warranty asserting a method equivalent to the following works:²

```
1 flight.seats_available(type) >= n
```

This warranty helps searching efficiently over the set of flights on which a ticket might be purchased. It does not help with the actual update when a ticket is purchased on a flight. In this case, it becomes necessary to find and update the actual number of seats available. However, this update can be done quickly when it does not invalidate the result asserted by the warranty.

4.2 Programming with Computation Warranties

Computation warranties explicitly take the form of logical assertions, so they can be requested by using a template for the desired logical assertion. In the airline

²As discussed below, computation warranties are restricted to method calls and their results in practice.

seat reservation example above, a query of the form

```
1 flight.seats_available(type) >= ?
```

might be used to find all available computation warranties matching the query, and at the same time fill in the “?” with the actual value n found in the warranty. In the case where multiple warranties match, a warranty might be chosen whose duration and value of n are “best” according to application-specific criteria.

In this chapter, we pursue a more transparent way to integrate warranty queries into the language, via memoized function calls. For example, we can define a memoized method with a straightforward implementation

```
1 memoized boolean seats_lb(Seat t, int n) {  
2     return seats_available(t) >= n;  
3 }
```

that returns whether at least n seats of the desired type are still available on the flight. The keyword `memoized` indicates that its result is to be memoized and warranties are to be issued on its result. To use these warranties, client code uses the memoized method as if it were an ordinary method, as in the following code:

```
1 for (Flight f : flights)  
2     if (f.seats_lb(aisle, seats_needed))  
3         display_flights.add(f);
```

When client code performs a call to a memoized method, the client automatically checks to see if a warranty for the assertion $? = \mathbf{f.seats_lb(type, n)}$ has either been received already or can be obtained. If so, the result of the method call is taken directly from the warranty. If no warranty can be found for the method call, the client executes the method directly.³

³From a logical perspective, “?” represents an existentially quantified variable for which the system finds a witness.

4.3 Ensuring Correct Behavior Using Computation Warranties

Our goal is that computation warranties do not complicate programmer reasoning about correctness and consistency—there should be no *observable* differences between performing the call and using the asserted result. Therefore, given a memoized method f , a computation of the form $v = o.f(\vec{x})$ occurring in a committed transaction should behave identically whether or not a warranty is used to obtain its value. This principle has several implications for how computation warranties work—only some computations make sense as computation warranties and updates must be prevented from invalidating active warranties.

To ensure that using a computation warranty is equivalent to evaluating it directly, we impose two restrictions on warranted computations:

1. Warranted computations must be deterministic. All calls of the underlying computation starting in equivalent system states must compute equivalent results. Therefore, computations using a source of nondeterminism, such as input devices or the system clock, do not generate computation warranties.
2. Warranted computations cannot have *observable* side effects. Side effects are considered to be observable only when they update the state of objects that existed prior to the computation.

This definition of observable means that warranted computations are allowed to create and initialize new objects as long as they do not modify pre-existing ones. For example, the top- N example from Section 4.1.2 computes a new object representing a set of items, and it may be convenient to create the object by appending items sequentially to the new set. Warranties on this kind of side-effecting computation are permitted. Enforcing this definition of the absence of side effects

is straightforward in a system that already logs which objects are read and written by transactions. At commit time, the transaction's write set is intersected with the read set of each potential computation warranty. A computation is marked as non-memoizable when a write is logged to a preexisting object during the computation or after the computation in a parent transaction.

In situations where a warranted computation creates and returns new objects, it is crucial for correctness of the computation that the objects returned by the warranty are distinct from any existing objects. This desired semantics is achieved by creating copies of all objects created when the asserted result was computed whenever a computation warranty is used. These objects are identified and recorded in the transaction log when computing the result.

4.4 Proposing and Issuing Computation Warranties

Whenever code at a client makes a call to a memoized method, the client may search for a matching computation warranty in its local cache. If the client is not already holding such warranty, it may search using a CDN, if available, or request the warranty directly from the appropriate store. This can be optimized by having the client execute the method in parallel with the search in case no warranty exists.

If the client cannot find an existing computation warranty, it performs the computation itself. It starts a new transaction and executes the method call. As the computation is evaluated, the transaction's log keeps track of all reads, writes, object creations, computation warranties used, and computation warranties proposed by the call. When the computation is finished, the result is recorded and the log is checked to verify that the call does not violate any of the restrictions outlined in Section 4.3. If the warranty is still valid, the call, result, and the logged operations performed during the computation are gathered to form a *warranty*

proposal.

At commit time, if the warranty proposal has not already been invalidated by an update to its read set later in the parent transaction, the proposal is sent to the store. The store looks at the request and, using the same mechanism as for state warranties, sets a warranty term as discussed in Section 4.6. For state warranties, terms are set using usage statistics estimated for the associated object. Computation warranties terms can be set in a similar fashion, using statistics estimated for each call with the same set of arguments. Finally, the computation warranty is issued to the requesting client and the store begins to defend the new warranty or warranties proposed by the client.

4.5 Using Computation Warranties

Computation warranties are used whenever available to the client, to avoid performing the full computation. At the start of a call to a method flagged as potentially warrantied, the client checks if there is a matching computation warranty. If an appropriate computation warranty is present, the call immediately returns the associated result asserted by the computation warranty and the transaction adds the computation warranty to its read set.

If the client uses a computation warranty that would expire before the transaction commits, the client can still use that expired warranty optimistically, similar to state warranties. At commit time, the expired warranty is revalidated during the prepare phase, exactly like a read prepare. Computation warranties are revalidated by the store by recomputing the call and checking the result against the assertion.

4.6 Setting Computation Warranty Terms

The method for setting state warranty terms also applies to computation warranties, where the term is determined based on estimates of read and write rates for the computation warranty's asserted result. A computation warranty is *read* when a transaction uses the asserted result in place of a call. Transactions that perform updates that *change* a computation warranty's result *write* the computation warranty. On the other hand, updates which do *not* affect the result are *not* considered writes. A computation warranty is considered written even if an update changing the result occurs while the warranty is inactive, ensuring that statistics are accurate when a new computation warranty is issued later.

4.7 Defending Computation Warranties

As with state warranties, the issuing store must defend against updates which would invalidate the assertion of a computation warranty before the end of its term.

A conservative approach to defending computation warranties against updates would be to delay all transactions that update objects used by the warranted computation. This is safe because of the determinism of the warranty computation, but it likely prevents many transactions from performing updates, hurting write latency and throughput.

Instead, the system attempts to *revalidate* affected warranties when each update arrives. The store reruns the warranty computation and checks whether the result is equivalent to the result stored in the warranty. For primitive values and references to pre-existing objects (not created by the warranty computation), the result must be unchanged. Otherwise, two results are considered equivalent if they

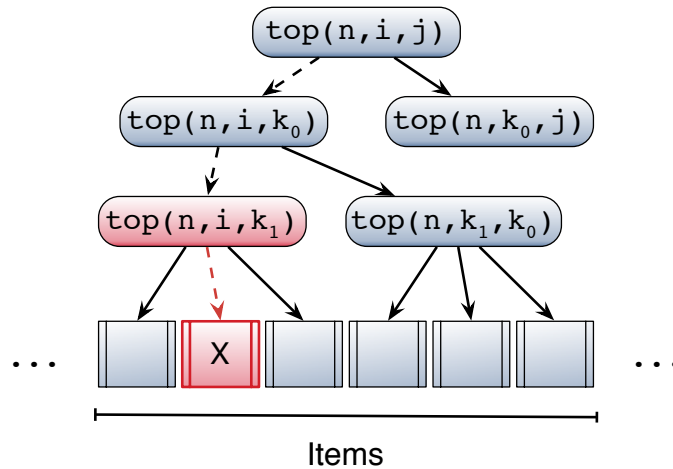


Figure 4.1: Update in tree of computation warranties.

are semantically equal per the `equals()` method, which may be user-defined, as in Java.

4.7.1 Incremental Revalidation

In general, computation warranties can be constructed *compositionally*—a warranted computation may use other warranties, either state warranties or other (sub)computation warranties. For example, in the top- N example from Section 4.1.2, if the method `top` is implemented recursively to process subsets of elements (see Figure 4.1), the warranty for a call to `top` depends on warranties for its recursive calls. The dependencies between warranties form a tree in which computation warranties higher in the tree depend on warranties lower down, and the leaves are state warranties.

Any warranty that has not expired must be defended against updates that could invalidate it. Defense is straightforward when the term of a warranty is within (a subset of) the terms of all warranties it depends on, including state warranties on all direct references to objects, because the validity of the higher-level warranty is

implied by the defense of the lower-level warranties.

A warranty can have a longer term than some of its dependencies. Updates to those dependencies may be delayed if they invalidate the computation warranty, even if the dependencies are expired warranties. However, it is possible to allow updates to warranty dependencies that do not invalidate the warranty; the value of dependencies may change without affecting the result. As a result, it is often feasible to give higher-level warranties longer terms than one might expect given the rate of updates to their dependencies.

For example, consider the recursive call tree for the method `top(n, i, j)` shown in Figure 4.1. If the request to see the top n items among the entire set is very popular, we would like to issue relatively long computation warranties for that result. Fortunately, updates to items (shown at the leaves of the call tree) that change their ranking might invalidate some of the warranties in the tree, but most updates will affect only a small part of the tree. Assuming that lower levels of the tree have either leased warranties or relatively short public warranties, most updates are not delayed much.

4.8 Public Computation Warranties Implementation

Public computation warranties were implemented as an extension of the state warranties implementation discussed in Section 3.9. To simplify the implementation for defending public computation warranties, the implementation only generates warranties for computations that involve objects from a single store.⁴ Also, our implementation does not use the dissemination layer to distribute computation warranties.

⁴Predictive treaties, discussed in Chapter 5, are not limited to single store computations and are designed to have low overheads for enforcing multistore warranties.

4.9 Evaluation

To evaluate public computation warranties, we used two benchmarks: a simple microbenchmark based on the top- N example described in Section 4.1.2, and the CMS benchmark described in Section 3.10.2, augmented with some memoized methods for generated page content and access control checks. The experimental setup was the same as in Section 3.10.4.

4.9.1 Top-Subscribers Benchmark

This new benchmark program simulates a relatively expensive analytics component of a social network in which users have subscribers. The analytics component computes the set of 5 users with the largest number of subscribers, using the memoized top- N function described in Section 4.1.2. The number of subscribers per user is again determined by a power-law distribution with $\alpha = 0.7$. The workload consists of a mix of two operations: 98% compute the list of top subscribers, corresponding to viewing the home page of the service; 2% are updates that randomly either subscribe or unsubscribe some randomly chosen user. This example explores the effectiveness of public computation warranties for caching expensive computed results.

4.9.2 Course Management System

For the CMS benchmark, we wanted to see if there were benefits or potentially prohibitive overheads when using computation warranties to memoize some common calls for generated content on various pages.

Table 4.1: Top- N benchmark: maximum throughput, latency, and 95th percentile write delay.

	Throughput (txn/s)	Median Latency (ms)	95th pct Write Delay (ms)
Fabric	17 ± 5	568 ± 500	N/A
Public State Warranties	26 ± 7	1239 ± 644	623 ± 387
Public Computation Warranties	343 ± 14	12 ± 3	16 ± 5

Table 4.2: CMS throughput with additional public computation warranties result highlighted.

System	Tput (tx/s)	Latency (ms)
JPA	72 ± 12	211 ± 44
Fabric	3032 ± 144	143 ± 120
Public Warranties	4142 ± 112	27 ± 27
Public Computation Warranties	4088 ± 189	114 ± 30

4.9.3 Results

For comparison, we ran the top- N benchmark with Fabric, with public state warranties, and with public computation warranties. Because the performance of the recursive top- N strategy on Fabric and on public state warranties was very poor, we used an alternate implementation that performed better on those configurations. Table 4.1 shows the average across three runs of the maximum throughput and the corresponding latency achieved in the system without any operations failing to commit during a 15 minute period. Public computation warranties improve throughput by more than an order of magnitude. Since the public computation warranty is on the value of the top 5 accounts rather than on each individual value used in computing the result, writes are not delayed as heavily as they are when using only state warranties.

For CMS, the results are shown with the additional data for the public computation warranties variant in Table 4.2, again across three trials of the configuration. Computation warranties only support computations on data from a single store, so

we did not compare results with the three store scenario explored in Section 3.10. CMS was originally designed without the support of computation warranties. The functions we designated to be memoized turn out not to have a significant impact on performance—CMS was already designed to keep these computations relatively cheap to evaluate on each request. However, the bookkeeping for public computation warranties adds no noticeable overhead.

4.10 Related Work

Many mechanisms for enforcing concurrency control have been proposed in the literature: locks, timestamps, versions, logs, leases, and many others [57, 42, 58, 86, 16, 40]. Broadly speaking, these can be divided into optimistic and pessimistic mechanisms. The monograph by Bernstein, Hadzilacos, and Goodman provides a broad overview from the perspective of databases [17]. Warranties are an optimistic technique, allowing clients to concurrently operate on shared data.

Haerder [45] divides mechanisms for validating optimistic transactions into “forward” and “backward” techniques. Backward validation is a better choice for the distributed setting [4], so Fabric uses backward validation: transactions are aborted in the prepare phase if any object in the read set has been modified.

Traditionally, most systems adopted *serializability* or *linearizability* as the gold standard of strong consistency [78, 17, 46]. But many recent systems have sacrificed serializability in pursuit of scalable performance. Vogels [107] discusses this trend and surveys various formal notions of *eventual consistency*. Much prior work aims to provide a consistency guarantee that is weaker than serializability; for example, causal consistency (e.g., [80, 68]) and *probabilistically-bounded staleness* [12]. Because this work is focused on stronger consistency guarantees, we do not discuss this prior work in depth.

Leveraging application-level information to guide implementations of transactions was proposed by Lamport [57] and explored in Garcia-Molina’s work on *semantic types* [34], as well as recent work on *transactional boosting* [47] and *coarse-grained transactions* [54]. Unlike warranties, these systems use mechanisms based on commuting operations. A related approach is *red–blue consistency* [61], in which red operations must be performed in the same order at each node but blue operations may be reordered.

Like warranties, Sinfonia [6] aims to reduce client–server round trips without hurting consistency. It does this through *mini-transactions*, in which a more general computation is piggybacked onto the prepare phase. This optimization is orthogonal to warranties.

Warranties borrow from leases [40] the idea of using expiring guarantees, though important differences are discussed in Section 3.2. In fact, the idea of expiring state guarantees occurs prior to leases in Lamson’s global directory service [59]. We are not aware of any existing system that combines optimistic transactions with leases or lease-like mechanisms, against which we could meaningfully compare performance.

A generalization of leases, *promises* [43, 51] is a middleware layer that allows clients to specify resource requirements via logical formulas. A resource manager considers constraints across many clients and issues time-limited guarantees about resource availability. Scalability of promises does not seem to have been evaluated.

The tracking of dependencies between computation warranties, and the incremental updates of those warranties while avoiding unnecessary invalidation, is close to the update propagation technique used in self-adjusting computation [1], but realized in a distributed setting. Incremental update of computed results has also been done in the setting of MapReduce with Incoop [18].

The TxCache system [82] provides a simple abstraction for sharing cached results of functions operating over persistent data from a single storage node in a distributed system. As with the Fabric implementation of computation warranties, functions may be marked for memoization. TxCache does not ensure that memoized calls have no side effects, so memoized calls may not behave like real calls. Compared to Fabric, TxCache provides the consistency guarantee of the underlying system, usually either snapshot isolation or serializability. Both of these guarantees are weaker than Fabric’s strict serializability.

Escrow transactions [76] have some similarities to computation warranties. They generalize transactions by allowing commit when a predicate over state is satisfied. Certain updates (incrementing and decrementing values) may take place even when other transactions may be updating the same values, as long as the predicate still holds. Compared to computation warranties, escrow transactions support very limited predicates over state, and their goal is different: to permit updates rather than to allow the result of a computation to be widely reused.

4.11 Discussion

Computation warranties realize the generality of the warranty abstraction: time-dependent assertions on the state of a distributed system. This generalization proves useful in cases where objects may be frequently updated without affecting *application-level* properties across multiple objects. This requires recomputing a result to check the effects of updates, which introduces new overheads for writes. Luckily, because computation warranties are *compositional*, these overheads tend to be low for updates which do not affect most of a computation tree, only the affected subcomputations need to be checked.

Computation warranties are beneficial in some cases, as demonstrated by the

top- N benchmark. However, as demonstrated in the CMS benchmark, they are not a general panacea for reducing contention and communication in strongly consistent distributed applications. Computation warranties work best when written with an appropriate level of composition: a computation should be constructed using subcomputations with *stable* results. In the next chapter, I discuss *predictive treaties* that leverage novel techniques for selecting subcomputations for assertions that are stable.

CHAPTER 5

PREDICTIVE TREATIES

In the previous chapter, we saw how computation warranties generalize time-dependent guarantees from explicit values to predicates over the state. The computation warranties presented in Chapter 4 were limited to store-scope predicates, however. While this is not a fundamental limitation of the computation warranties abstraction, system-scope predicates present new challenges for ensuring good performance—enforcing system-scope warranties requires synchronization whenever a multistore predicate is recomputed to check if its result has changed.

Treaties, an abstraction used in the homeostasis protocol [87], were designed with this enforcement overhead in mind. Unlike public warranties, treaties were not time-dependent and were revocable—treaties improve performance by avoiding multistore queries but do not remove the need for read validations entirely. Furthermore, treaties were not designed to support arbitrary hierarchical composition like computation warranties.

In this chapter, we discuss predictive treaties, which extend the original treaties design with features from computation warranties—time-limited guarantees and hierarchical structure—as well as a novel generalization to predicates that depend on *time*. Predictive treaties are a *leased* warranty abstraction; like the original treaty abstraction, predictive treaties are revocable and intended to avoid multistore transactions as much as possible.

Like the previous chapter’s computation warranties and the homeostasis protocol’s treaties, predictive treaties exploit the predictability of computation using logical predicates over system state. A key design element is the separation of the computation on system state from the predicate being enforced on that computation. *Metrics*, computations on the state that predictive treaties guarantee

predicates over, measure the expected “distance” to the violation of a predicate and predict how this distance will change. Similar to the model used to set public warranty terms discussed in Section 3.7, metrics construct estimates to predict future updates to this distance. These predictions can be used to avoid remote data access without harming consistency guarantees that applications rely on.

A novel feature of predictive treaties is that their predicates may be *time-dependent*: predictive treaties not only express conditions on the state of the system, but also how the state will change as a function of time. For example, if a metric $f(x)$ computes the amount of stock in a warehouse, a predictive treaty might guarantee the inequality $f(x) > 100 - 5t$, where t represents elapsed time in minutes. Such a treaty would allow this inequality (and any other predicate it implies) to be evaluated without any distributed communication. Note that a predictive treaty is not an invariant in the strictest sense, as it is not guaranteed to hold for all time. Even so, our evaluation shows that time dependence allows predictive treaties to reduce synchronization by orders of magnitude for some applications.

Of course, warranties and predictive treaties are not free: there are costs associated with creating and maintaining run-time objects to represent metrics and predicates. These costs are magnified when the predicates are over geodistributed state—communication to use and manage these objects requires high latency communication between nodes and increased contention for client transactions. In a naive implementation, these costs could be greater than the costs associated with executing the distributed application itself. To help support better reuse of predictive treaties and thereby reduce this overhead, we introduce *stipulated commit*, a mechanism that allows the programmer to propose updates that are applied only if doing so does not violate a treaty. As shown in our evaluation, stipulated

commit enables building distributed applications using simple predictive treaties whose performance is competitive with hand-written code.

This chapter explores the design and implementation of predictive treaties as a primitive for distributed programming. The intuition behind predictive treaties is outlined with an example application: running an election and tracking the front-runner. Preliminary results demonstrate that our intuitions can lead to significant performance improvements (Section 5.1). Predictive treaties and metrics are formally described along with how the system ensures they are consistent (Section 5.2). We provide a simple interface for programming with metrics and predictive treaties that abstracts away much of the details for ensuring consistency and good performance (Section 5.3). Techniques for automatically constructing low-synchronization policies for enforcing treaty consistency are discussed (Section 5.4). Modifications to a standard distributed transaction system for ensuring consistency and good performance when using treaties are discussed based on our experience implementing predictive treaties in Fabric (Section 5.5). We demonstrate that predictive treaties are effective at avoiding coordination using a series of benchmarks as well as evaluate the usefulness of some key design elements, such as the prediction model and estimation features (Section 5.6). At the end of this chapter, I discuss related work (Section 5.7) and some conclusions (Section 5.8).

5.1 Predictive Treaties by Example: Voting

We begin by considering a simple application where predictive treaties prove useful: a voting system that records and totals the votes for candidates in an election. Votes arrive at one of some number of geographically distributed voting stations and must be tallied to obtain candidate totals. The application keeps track of which candidate is leading—a global property—and makes this information widely available. While accurate up-to-the-minute winner determination is not a feature

of current voting systems, it is paradigmatic of a broader class of applications requiring tracking of data aggregates [10, 81, 24].

For simplicity, assume there are two candidates, A and B , and two voting stations, nodes S_1 and S_2 . The two nodes process transactions for casting votes, `vote(A)` or `vote(B)`, and query for the current front-runner, `winning_candidate()`. Voting increments a station-local vote total for the indicated candidate. At station S_1 , the vote totals for the two candidates are a_1 and b_1 ; at S_2 , they are a_2 and b_2 . Front runner queries return which candidate has a greater vote total across both nodes, returning A when $a_1 + a_2 > b_1 + b_2$, for example.

Because the current winner is a global property that depends on widely distributed data, any straightforward implementation of `winning_candidate()` using conventional serializable transactions will be slow, even if the winner changes infrequently. Each transaction must check who the new winning candidate is before committing, and this check requires synchronization among all voting stations to ensure that the vote totals observed are consistent with the system state. Fortunately, the state of this application evolves in a predictable way that can be exploited by predictive treaties to avoid synchronization. In particular, each update changes only one total at a single station. Furthermore, we may reasonably assume that the overall voting trend is fairly consistent at each station, over significant time periods.

5.1.1 Enforcing Predicates with Slack

Suppose A is in the lead, and the application creates a global predicate, stating `winning_candidate() = A`, to monitor the current winner. This global predicate can be enforced using predictive treaties that track the local vote totals at each station.

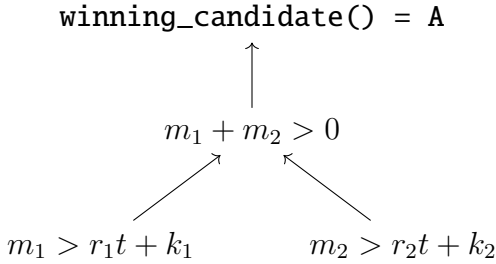


Figure 5.1: Locally enforceable predictive treaties imply a global predicate. Here $k_2 = -k_1$ and $r_2 = -r_1$.

With A currently in the lead, the global predicate is equivalent to the predicate $(a_1 - b_1) + (a_2 - b_2) > 0$: the total of the margins must be positive. Defining local margin variables $m_1 = a_1 - b_1$ and $m_2 = a_2 - b_2$, this predicate can be written more simply: $m_1 + m_2 > 0$.

The quantity $m_1 + m_2$ changes by 1 on each vote, so it tracks the minimum number of votes that might invalidate the global predicate. We call this quantity the *slack* of the global predicate, because it measures how far the global predicate is from being invalidated. The global predicate can therefore be enforced by identifying values k_1 and k_2 such that the inequalities $m_1 > k_1$ and $m_2 > k_2$ hold and the global slack $k_1 + k_2$ is nonnegative. As long as the local predicates hold at all of the nodes, no synchronization is required. If an update violates a local predicate, then synchronization among the nodes is required, either to invalidate the global predicate or to establish new local predicates that can then be enforced. However, consistency is not threatened by the falsification of local predicates: the system falls back to synchronization to ensure consistency when predictive treaties no longer hold.

Ignoring questions of how to obtain local predicates and of how to choose values for k_1 and k_2 , what we have described thus far corresponds to the approach taken in prior work [87]. We show next how to further improve performance using predictive treaties, which generalize static predicates with mechanisms for

predicting the evolution of system state in order to reduce synchronization.

5.1.2 Time-Dependent Treaties

Assume voting stations are associated with populations that each exhibit their own overall preference for the candidates. Further, for simplicity, assume that voters cast votes independently with different biases at the two sites. In this case, there may be some variation in the local margin observed at each node, but a trend is likely to be observable over a long series of votes. For example, if the population at node S_1 is split 60–40 for candidate A , and an equal-sized population at node S_2 is split 48–52, then the margin for A at S_1 is likely to increase over time, whereas the margin for A at S_2 is likely to (more slowly) decrease over time. In this case, we say that S_1 is a *positively biased* node, because its updates tend to increase the slack of the global predicate. Conversely, S_2 is a *negatively biased* node: its updates tend to decrease the slack of the global predicate. Assuming that voting patterns do not change over time, it is likely that the total bias across all nodes will be positive, meaning that the global predicate is unlikely to be falsified soon.

Note that the assumption of independence of voting does not need to hold perfectly. What is key is that there are predictable trends over time periods that are long enough to be useful for reducing synchronization. If votes are correlated with time—for example, if B voters tend to vote later than A voters—then the trend may depend on time, and global bias could become negative as the trend switches.

We can take advantage of these underlying trends by creating time-dependent treaties that automatically track the evolution of system state. Suppose that the nodes have the biases above (the system is positively biased), and for simplicity, that votes arrive at an average rate of one vote per time unit at each site. Then

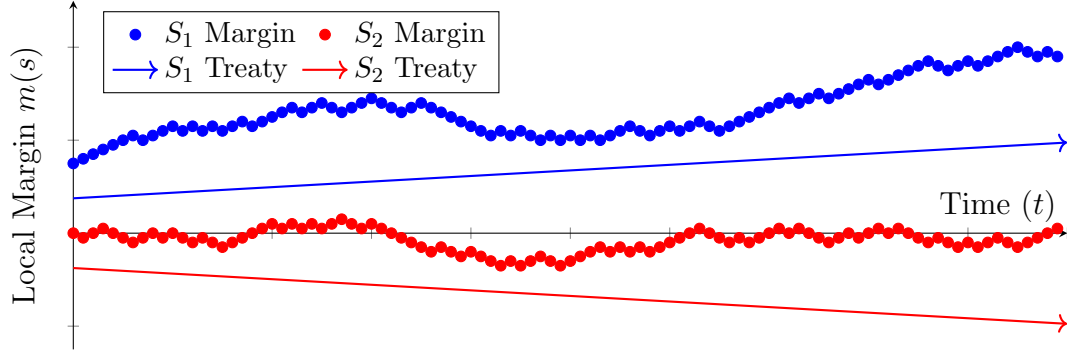


Figure 5.2: Time-dependent predictive treaties and corresponding local vote margins over time.

the expected rate of increase in global slack from node S_1 is 0.2 votes per time unit, whereas S_2 is expected to decrease global slack by 0.04 votes per time unit. We can then define local predicates that incorporate these *slack velocities*. For some constants k_1 , k_2 , r_1 , and r_2 , node S_1 enforces a local predicate—a predictive treaty—with the form $m_1 > r_1 t + k_1$, and S_2 enforces $m_2 > r_2 t + k_2$. If the sum $k_1 + k_2$ is no larger than the initial global slack and the sum $r_1 + r_2$ is nonnegative, the conjunction of these local treaties enforces the global treaty, as depicted in Figure 5.1.

The parameters r_1 and r_2 allow building slack velocities into the local predicates with the effect that slack is continuously transferred between nodes without any synchronization. If $r_1 < 0.2$ and $r_2 < -0.04$ (with $r_1 + r_2 \geq 0$ as before), the local predicates at S_1 and S_2 can remain true indefinitely, despite the negative bias of S_2 . Specifically, if we choose $r_1 = 0.12$ and $r_2 = -0.12$, local slack is expected to accumulate, equally fast, at both S_1 and S_2 .

An example run of this scenario is shown in Figure 5.2, plotting the two local margin values and corresponding treaty lower bounds over time. The upward slope of S_1 's treaty reduces the slack at S_1 . This slack reduction allows S_2 's treaty to increase slack at S_2 , sloping downward. The overall effect is a continuous transfer

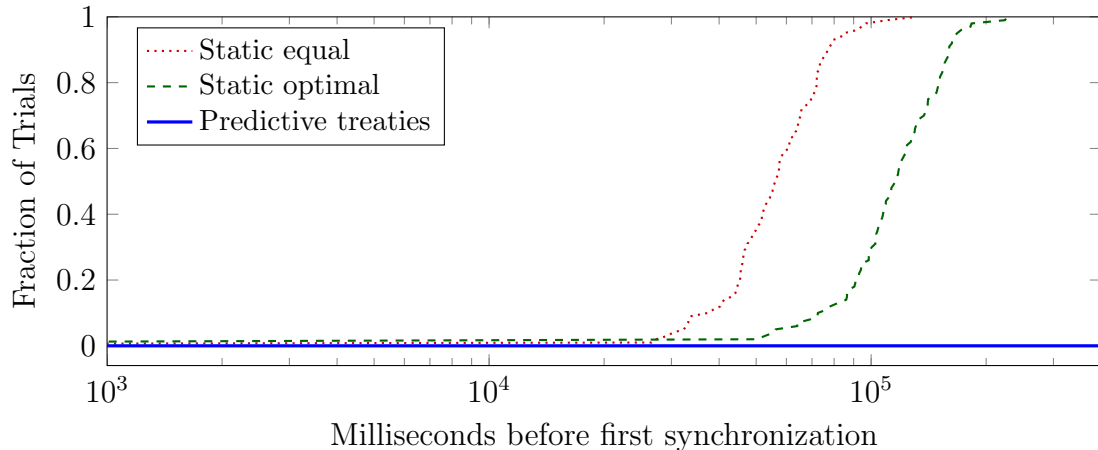


Figure 5.3: CDF of time until first synchronization under three different slack-allocation strategies. With predictive treaties, less than 1% of runs synchronize.

of slack from S_1 to S_2 . The sum of the two treaty bounds is always equal to zero, so their conjunction always implies that the total margin is greater than zero.

Of course, it would be awkward for programmers to have to choose values for parameters like r_i and k_i . Fortunately, they do not need to make this choice. The predictive treaties framework selects them automatically—see Section 5.4.2.

5.1.3 Preliminary Evaluation

To see the potential performance benefits of predictive treaties, consider the results from a distributed implementation of the two-station scenario just described, shown in Figure 5.3. (Section 5.6 discusses this benchmark and variations on it in more detail.) In this experiment, stations receive 100 votes per second with each vote selecting a candidate randomly according to the distribution in preferences associated with the station; so at S_1 for example, each vote transaction has a 60% chance that it will be a vote for candidate A . After 30 seconds of voting, the system creates a treaty which asserts the current front-runner and then continues to run a query every 100ms for the up-to-date front-runner. We then measure the time after

the treaty was created until the application next synchronizes during either a vote or query transaction—the distribution of times for this first synchronization acts a proxy for the distribution of times between synchronizations in the system.¹ The figure compares three strategies for avoiding synchronization, showing a cumulative distribution of times across 100 trials for each strategy. The dotted red line shows the result when dividing slack equally between the nodes, in a manner similar to a baseline used by some previous work [26, 10]. The dashed green line shows the result when using knowledge of the workload to optimally give most of the slack to the negatively biased node; this strategy, which almost doubles the median time to first synchronization, is most similar to the homeostasis approach [87], which uses workload data to approximate the optimal static division of slack. However, predictive treaties reduce synchronization even more dramatically, as shown by the solid blue line in the figure. Synchronization is usually avoided entirely, improving significantly on even the best possible static division of slack.

5.1.4 Hierarchical Treaties

A typical voting system would have more than two voting stations. The approach sketched above can be generalized directly to an arbitrary number of nodes by dividing up slack and slack velocity among all participating nodes. However, this approach does not scale well: the rate of synchronization increases because there are more predictive treaties that can individually fail, and the cost of synchronization also increases because more nodes need to achieve consensus on the new predictive treaties to be enforced subsequently.

A better alternative is to enforce predictive treaties hierarchically, similar to

¹The two distributions are not identical, however. With the stable voting patterns in this experiment, all three strategies take more and more time for each subsequent synchronization because the margin difference (slack) will grow.

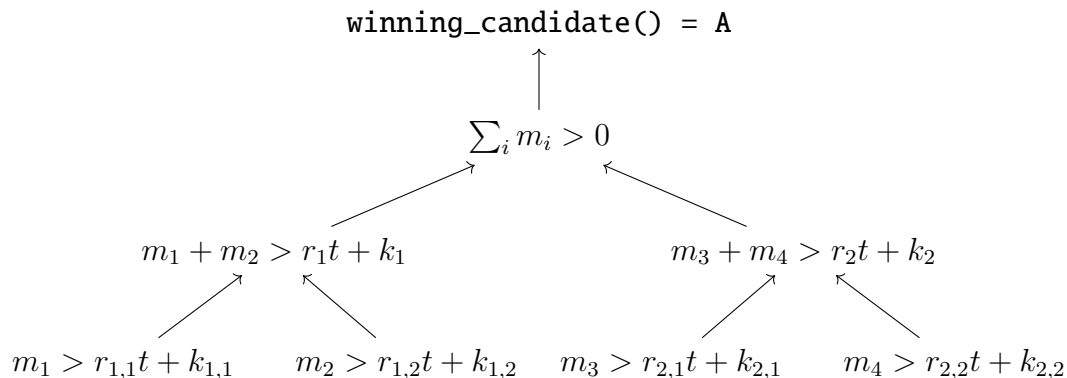


Figure 5.4: Hierarchical predictive treaties. Interior nodes allow the system to avoid propagating synchronizations to the entire system when they cannot be avoided.

the top- n computation in Chapter 4. As described above, a predicate of the form $m_1 + m_2 > 0$, where m_1 and m_2 are the margins at the two nodes, can be enforced via predictive treaties of the form $m_1 > r_1 t + k_1$ and $m_2 > -r_1 t - k_1$. But this strategy can be generalized. Assertions of the form $m_i > r_i t + k_i$ can themselves be enforced recursively via lower-level predictive treaties at other nodes.

Hence, we can organize the voting stations into a tree, like the one in Figure 5.4, in which connected nodes, especially near the leaves of the tree, are located near each other to reduce communication latency. Each tree node enforces a predictive treaty. Leaf nodes accept votes that update state and that potentially violate the predictive treaty which the node is enforcing. Interior nodes enforce predictive treaties of the same form by negotiating predictive treaties with their child nodes based on the relative bias of those nodes. Slack and velocity are distributed from the root downward in such a way that predicate failures occur infrequently and when they occur, usually do not propagate changes high into the treaty tree.

As the results in Section 5.6 show, hierarchical predictive treaties allow synchronization to be localized to just part of the system, involving relatively few nodes that are connected with relatively low latency.

5.2 Predictive Treaties and Metrics

We now present our approach in more formal manner, introducing predictive treaties as well as metrics. The system enforces the consistency of treaties used in a distributed application by monitoring for changes to metrics and treaties.

5.2.1 Predictive Treaties

A predictive treaty is a leased warranty that is both time-varying and time-limited with system-scope predicates. A predictive treaty’s predicate is an expression on the value of a *metric computation* and the current time:

$$\underbrace{\phi(m(s), t)}_{\text{metric}} \text{ until } \underbrace{t_{\text{expiry}}}_{\text{time limit}} \quad (5.1)$$

The predicate is guaranteed to be true until the associated expiry time has passed or until the treaty is explicitly *retracted* by a transaction, updating the associated expiration time.² Note that these predicates are *not* necessarily invariants of the system state, and may only hold briefly.

The components of the treaty are as follows:

- $m : S \rightarrow \tau$ is a *metric*, which represents a computation on a system state $s \in S$, with a result of type τ .
- $\phi : \tau \times \mathbb{T} \rightarrow \mathbb{B}$ is a boolean predicate over the metric’s value and the current time t .
- t_{expiry} is the expiry time, after which the predicate ϕ is no longer guaranteed to be true.

²Note that this is a difference with computation warranties, which cannot be retracted. Because predictive treaties can be retracted, they do not avoid read prepares.

The general form of predictive treaty (5.1) is a predicate over both the state s and time t . Time is left out of the metric, and is therefore treated differently from other parts of system state. This simplification is useful because the system has no control over how time evolves.

The voting example in Section 5.1 demonstrates *threshold treaties*, in which the predicate ϕ checks a time-dependent vector threshold against a vector-valued metric:

$$\underbrace{\overbrace{\vec{m}(s) \geq \vec{b}(t)}^{\text{predicate } \phi}}_{\substack{\text{metric} \quad \text{boundary}}} \text{ until } \underbrace{t_{\text{expiry}}}_{\text{time limit}} \quad (5.2)$$

Here, the metric type τ is \mathbb{R}^n for some number of dimensions n . For the treaty to hold, the predicate’s inequality $\vec{m}(s) \geq \vec{b}(t)$ must hold componentwise; effectively, the predictive treaty enforces a conjunction of n scalar constraints that share an expiration time.

Threshold treaties model two intuitions from Section 5.1:

- To enforce distributed assertions with low synchronization, we use local assertions that are “far” from being falsified. The metric represents the current system state as a point in \mathbb{R}^n that is compared with a boundary $\vec{b}(t)$. When the metric’s location is far from the boundary, the treaty is similarly far from being falsified.
- Slack can be implicitly shifted between nodes by having their treaties vary with time. The boundary term $\vec{b}(t)$ shifts over time to either reduce or increase slack.

Threshold treaties have a variety of uses and can be maintained with low synchronization overhead. *Linear* threshold treaties, in which the boundary $\vec{b}(t)$ depends linearly on t , are particularly useful, as in the voting example.

As in prior systems [63, 64, 4, 5, 25, 91], we rely on loosely synchronized clocks.

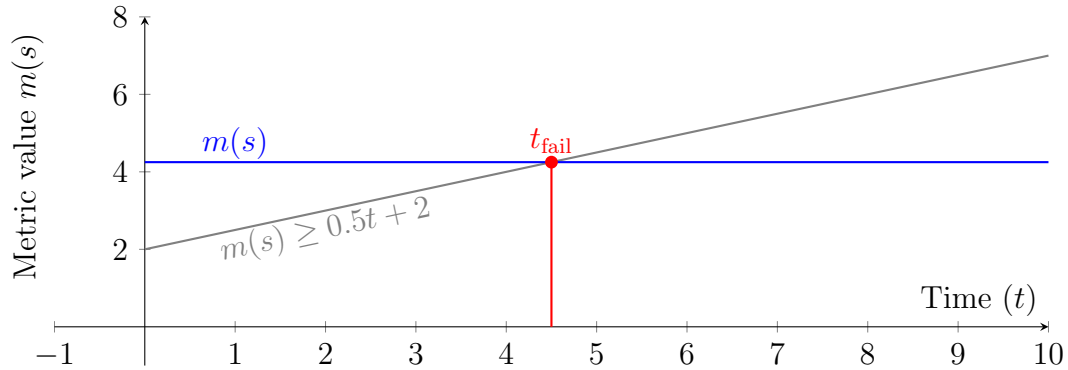


Figure 5.5: A predictive treaty falsified by time passing. Although the metric $m(s)$ never changes value, the treaty is eventually falsified at t_{fail} .

We account for possible skew ε between clocks by enforcing the most conservative interpretation of a predictive treaty—i.e., assume that clocks may differ by ε .

5.2.2 Enforcing Predictive Treaties

We say that a predictive treaty created at time t_{create} holds if for all times $t \in [t_{\text{create}}, t_{\text{expiry}})$, the predicate $\phi(m(s), t)$ holds. In other words, a predictive treaty holds if, at any time before t_{expiry} after its creation, a distributed application can use it in place of a strongly consistent computation that explicitly checks the predicate.

A predicate $\phi(m(s), t)$ that currently holds may be falsified (i.e., stop holding) in two ways:

- An update to the system state may change the value of the metric m from value $m(s) = v$ to a new value $m(s') = v'$, such that $\phi(v', t)$ is false.
- As time passes, the predicate may become false due to its dependence on t . In the case of threshold treaties, $\vec{b}(t)$ can grow with time and become larger than the value of $\vec{m}(s)$ in the current system state, as shown in Figure 5.5.

For a given system state, the future time t_{fail} when this second scenario occurs, if

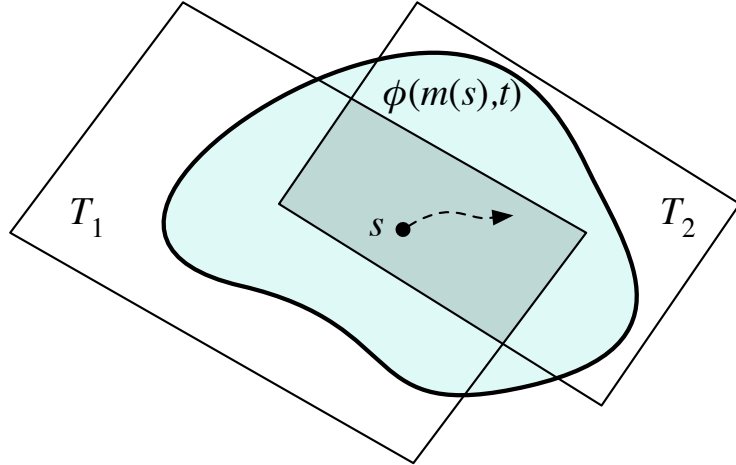


Figure 5.6: System state s evolves within the intersection of local predictive treaties, enforcing global predicate $\phi(s)$.

any, can be determined from the current value of $m(s)$, assuming $\vec{b}(t)$ is suitably well-behaved. Therefore, to ensure that a predictive treaty holds, the system ensures that $t_{\text{expiry}} \leq t_{\text{fail}}$, if it exists. This will falsify the treaty when t_{expiry} is set to earlier than the current time.

As a form of computation warranty, there are two choices for enforcing predictive treaty assertions: either directly or using other predictive treaties.

In the direct method, the system enforces predictive treaties by recomputing m and update t_{expiry} as needed on each update to the state s referenced by the metric m . However, recomputing m can be expensive and may require synchronization if it reads state from more than one node.

The alternative enforcement strategy, using other predictive treaties, avoids this synchronization. In this case, the system enforces multiple local *subtreaties* that in conjunction imply the original, higher-level treaty. The subtreaties are *local* in the sense that they depend on state that is localized to a subset of the state referenced by the treaty it helps to enforce. These local subtreaties can be enforced without synchronization with nonlocal nodes. This approach is illustrated in Figure 5.6, where a global predicate $\phi(m(s), t)$ is enforced using subtreaties T_1 and T_2 . As

long as system state s stays within the intersection of T_1 and T_2 , it also satisfies $\phi(m(s), t)$. Using subtreaties to enforce a predictive treaty, t_{expiry} only needs to be updated for the original predictive treaty if the minimum expiration time of the subtreaties has become earlier than t_{expiry} . Hence, the system requires less synchronization if subtreaties are chosen so that the minimum of their expiration times is unlikely to change, despite changes to the state read by their metrics.

5.2.3 Metrics

The metric in a predictive treaty is an object that tracks the value of a computation over stored state. The implementation of a metric may also track statistics for modeling how this value evolves over time. These *update statistics* can help predict future changes and, as a result, enable estimation of how long the predicate in the treaty will hold. We discuss specifics of a statistical model used for threshold treaty predictions in Section 5.2.4.

There are two kinds of metrics, *direct* and *derived*:

Direct metrics Direct metrics are computed directly from the state of the system, and are updated as the system state evolves. Recall that in the hierarchical voting system example, there is a tree of predictive treaties. Each leaf node in the tree maintains a direct metric for the margin observed at that node. As votes come in, changing the margin, the metric and its estimated statistics are updated.

Derived metrics Derived metrics are used for hierarchical predictive treaties. They depend on other metrics. In the hierarchical voting system, each interior node maintains derived metrics, which in this case are aggregate margins for the subtree at that point, like the node labeled $m_1 + m_2$ in Figure 5.1. To avoid synchronization, the state of a derived metric is constructed using the state of its submetrics and

is not updated until nodes are otherwise required to synchronize. When a node maintaining a derived metric synchronizes with the nodes maintaining the source metrics, the statistics for the source metrics are combined to construct statistics for the derived metric.

The hierarchy created using derived metrics creates a *metrics tree*, with direct metrics as the leaves and derived metrics as the internal and root nodes. Metrics trees are analogous to abstract syntax trees for representing program structure in compilers or logical plans in databases; their structure guides the system when automatically creating subtreaties.

5.2.4 A Prediction Model for Metric Updates

Accurate prediction of the system trajectory depends on tracking not only the current value of metrics, but also other attributes. For threshold treaties, the expected metric *velocity* (that is, rate of change in \mathbb{R}^n) allows the system to predict how long it will take for a given predictive treaty to fail. This prediction allows choosing predictive treaties at multiple nodes that allocate slack so that the earliest predictive treaty violation is expected to happen as late as possible.

Of course, system state does not typically exhibit perfectly predictable behavior. To predict the value of a metric, some model of its behavior is needed. If the model is inaccurate, it can harm performance, but not consistency.

We have explored a simple model for numeric metrics, as a random variable M that is the sum of two processes: a predictable linear process and a scaled Brownian process [28] that represents the accumulation of random variation. The variable M is defined as $M = m_0 + vt + \sigma B_t$, where the linear process has value m_0 at time $t = 0$ and moves at constant velocity v . The Brownian process σB_t at time t has a Gaussian distribution with a mean of 0 and a standard deviation of

$\sigma\sqrt{t}$. A numeric metric is therefore characterized by three parameters: m_0 , v , σ . In Section 5.4.1, we discuss how we estimate these parameters for metric objects based on updates observed by the system.

For example, in the voting example of Section 5.1.2, at site S_1 (where votes are split 60–40), the margin is a Markov chain that is approximated well by parameters $m_0 = 0$, $v = 0.2$, and $\sigma = 0.98$.

If underlying system state changes in an approximately linear way, it is reasonable to use a linear model for the nonrandom component of the metric; we have no evidence that more complex models of metric behavior are worthwhile. Work in settings with weaker consistency needs has found that linear models often work well in practice, with diminishing returns for more sophisticated models [37]. A larger class of functions could be captured by including a transformation in the metric, however. For example, a quantity expected to vary exponentially over time can be converted into a linear metric by using the logarithm of the quantity as the metric. Quantities expected to vary polynomially could be similarly transformed to more nearly linear behavior.

5.2.5 Expiration

The general form of a predictive treaty in Equation (5.1) includes an expiration time t_{expiry} . This component is useful for enforcing application-level predicates that include an expiration time, as in the case of the warranty design in the previous chapters. However, as noted in Section 5.2.2, there is a fundamental reason why expiration times are needed.

As discussed in Section 5.1, a time-dependent predictive treaty can transfer slack continuously from positively biased local treaties to negatively biased ones. A threshold predictive treaty is positively biased at time t if the term $b(t)$ is

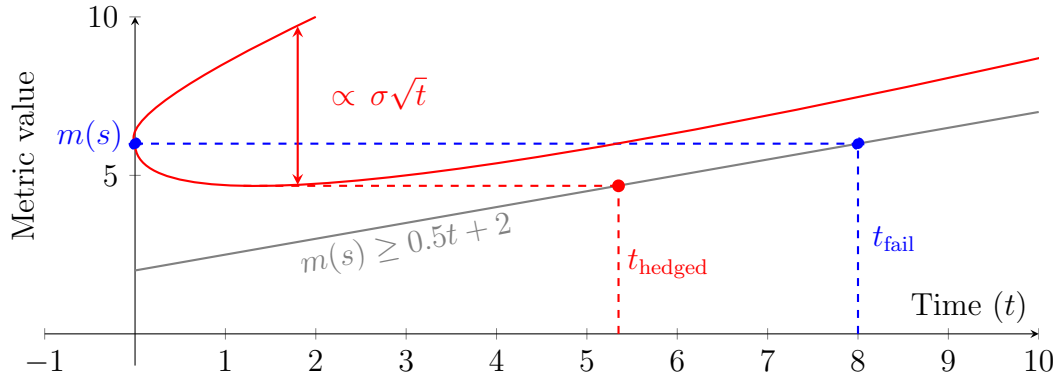


Figure 5.7: Setting a treaty’s expiry based on predicted trajectory. The red parabolic curve suggests the envelope of likely metric values over time. Hedging the expiration time to t_{hedged} leaves room for negative updates.

increasing over time, presumably because the workload updates are also positively biased. In the simple case of a linear bound $b(t) = rt + c$, the sign of r determines the treaty bias.

Predictive treaties that are not positively biased remain valid in the absence of updates to the state s because the bound $b(t)$ remains fixed or moves away from the metric value. Therefore, these predictive treaties can be enforced by forcing synchronization when updates arrive that would require them to be retracted. This synchronization may cause the global assertion to fail but its failure will be serializable and observations of state will remain consistent.

Positively biased predictive treaties, however, build in an expectation that incoming updates generate slack in the underlying metric. Consequently, if updates cease, a positively biased treaty becomes invalid at a time t_{fail} simply through the passage of time. In general, such invalidations are unavoidable for positively biased treaties in the absence of updates.

For example, Figure 5.7 shows a predictive treaty with the form $m(s) \geq 0.5t + 2$, corresponding to the gray diagonal line. The metric, shown in blue, is required to stay above this line. At time 0, the metric has value 6. Absent any updates, the

treaty becomes invalid at time $t_{\text{fail}} = 8$, so synchronization-free enforcement of the treaty requires that the value of t_{expiry} be at most 8.

Hedging While it is safe to use 8 as t_{expiry} in this example, updates that cause the metric’s value to drop below its initial value would require synchronization to commit. Any reduction below the initial value introduces the potential for the predictive treaty to become false earlier than the initially promised expiration time of 8. To allow some slack-reducing updates to occur without synchronization, the expiration time can be artificially shortened, as suggested by the red dashed line in the figure, corresponding to the time t_{hedged} . The amount of expiration-time hedging should depend on the noise parameter σ , to balance the cost of making the expiration time too short against the possible cost of synchronization arising from slack-reducing updates. In our implementation, we use 3σ to account for three standard deviations of noise.

Asynchronous extensions As time passes, a node managing a positively biased predictive treaty expects updates that increase a treaty’s expiration time. Importantly, these extensions can be performed by the system periodically, without requiring the client to synchronize, and communicated *asynchronously* to other nodes using the treaty.

Whenever a transaction performs updates that potentially allow a treaty’s expiration to be extended—either a metric update that moves away from a treaty bound or an extension to a subtreaty’s expiry—an asynchronous *extension message* is sent to the node managing the treaty. At any time, usually some time after receiving an extension message, the managing node can run a transaction to extend the treaty’s expiry. This transaction may trigger further extension messages to other nodes managing treaties that depend on the extended treaty.

There is no need to acknowledge extension messages; a lost or delayed message may lead to unnecessary synchronization but cannot cause inconsistency. Processing extension messages is also discretionary. To avoid gratuitous overhead, the receiving node can wait until just before the previously advertised expiration time to attempt an extension of the treaty and, if successful, sending out extension messages. Because such messages do not incur round-trip delays and do not delay client transactions, we do not consider them to be synchronizations.

5.3 Using Predictive Treaties

Predictive treaties and metrics enable complex, efficient implementations for distributed applications. However, the API is simple and intuitive.

5.3.1 Programming with Treaties and Metrics

Part of the appeal of predictive treaties is that they support a simple programming interface. To demonstrate this simplicity, Figure 5.9 gives the top-level code for the voting example, using several supporting definitions from Figure 5.8. The function `winning_candidate()` defines the top-level computation: it computes the election winner while memoizing the result by generating a treaty that can be used later to check the result. Under the covers, the implementation enforces the underlying predictive treaties to keep this result valid for as long as possible, avoiding recomputation and synchronization.

The method `winning_candidate()` computes a **Metric** for the margin between the candidates across a set of voting stations, using a helper method `margin()` that builds a metrics tree by partitioning the voting stations and recursively computing the sub-margins for those stations, combining the results into a single met-

```

1 interface Treaty {
2     /* For client use */
3     boolean valid();
4 }
5
6 interface Metric {
7     /* For client use */
8     double value();
9     // Constructs a SumMetric
10    Metric plus(Metric other);
11    // Constructs a ScaledMetric
12    Metric times(double scalar);
13    // Syntactic sugar for adding a -1 scaled Metric
14    Metric minus(Metric other);
15    // Constructs a MinMetric
16    Metric min(Metric other);
17    Treaty getTreaty(TreatyStatement stmt);
18
19    /* For internal use */
20    double velocity();
21    double noise();
22    Set<Treaty> policy(TreatyStatement stmt);
23 }
24
25 interface TreatyStatement {
26     /* For internal use */
27     boolean check(Metric m);
28 }

```

Figure 5.8: Treaty and Metric interfaces.

ric using the **plus** operation. Depending on the sign of the resulting margin, **winning_candidate()** then uses **getTreaty()** to generate a predictive treaty for either the returned metric or its negation (using the **times(-1)** operation). The parameters to the call to **getTreaty()** represents the lower bound on the value, set to **0** here. Figure 5.10 shows the tree of metrics used in the voting example. The association between the returned winner and the treaty can be treated as enforcing assertions of the form $f(s) = y$.

```

1 TreatyStatement zeroBound = new LowerBound(0);
2
3 /* Return the current winner and a treaty that implies
4  * this result still holds.
5  */
6 Pair<String, Treaty> winning_candidate(String u, String v) {
7     Metric diff = margin(u, v, allStations);
8     if (diff.value() >= 0) {
9         return new Pair<>(u, diff.getTreaty(zeroBound));
10    }
11    return new Pair<>(v, diff.times(-1).getTreaty(zeroBound));
12 }
13
14 /* Return the metric which computes the margin between
15  * candidates u and v across the given districts.
16  */
17 Metric margin(String u, String v, List<Station> ds){
18     int n = ds.size();
19     if (n == 1) {
20         Station d = ds.get(0);
21         return d.votesFor(u).minus(d.votesFor(v));
22     }
23     int mid = n / 2;
24     Metric fst = margin(u, v, ds.subList(0, mid));
25     Metric snd = margin(u, v, ds.subList(mid, n));
26     return fst.plus(snd);
27 }

```

Figure 5.9: Voting example code using treaties.

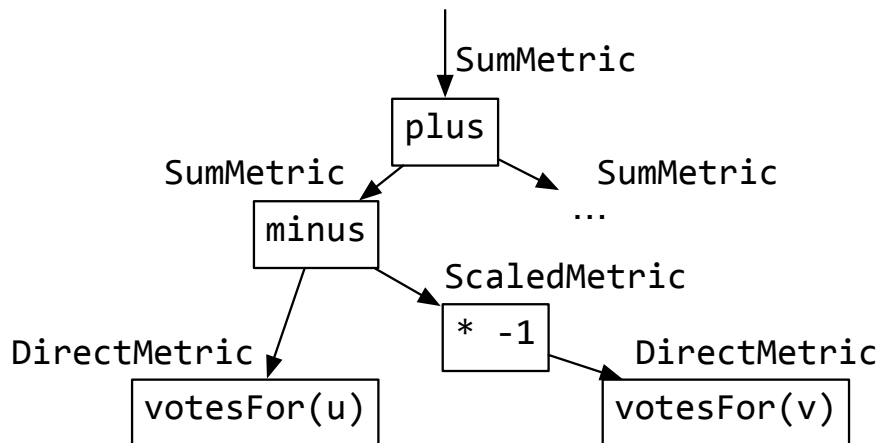


Figure 5.10: Metrics tree created by example code.

Figure 5.8 defines the interfaces for objects of type **Treaty**, **TreatyStatement**, and **Metric**. After a treaty is created, **valid()** returns true until the treaty expires. **Metric** objects provide methods for computing their **value()**, for estimating their **velocity()** and **noise()**, and for obtaining derived metrics. The method **policy(stmt)** automatically creates subtreaties to enforce a predictive treaty asserting the given statement on the metric's **value()**, using techniques discussed below in Section 5.4.2. If no subtreaties are returned, the statement is enforced directly.³

In our example, votes at an individual station are tracked using metrics that can be updated directly by the application. Votes across stations are tracked using **SumMetrics** and **ScaledMetrics** (produced by the **plus** and **minus** operators), which use the methods **velocity()** and **noise()** to divide up slack proportionally between their sub-metrics.

5.3.2 Stipulated Commit

To reduce the overhead of creating and maintaining predictive treaties, it can be better to beg forgiveness than to ask permission. The predictive treaties programming interface converts logical conditions that the programmer wants to test into treaties that are monitored and enforced. However, creating and maintaining a treaty (“asking permission”) has a cost that is not worth paying if the treaty cannot be reused enough. Unfortunately, programs as written often test logical conditions that are not worth promoting into treaties.

One problematic pattern arises when an application performs certain updates only if a postcondition would hold afterward, but where the success of the update depends on varying input. For example, in a banking application, a withdrawal

³This is a simplified presentation of the implementation, which returns different **Policy** values for different enforcement strategies.

from an account might be allowed only if the final account balance is nonnegative. Traditionally, a programmer would enforce such a postcondition by checking a sufficiently strong precondition before performing the update. For example, the banking application might guard the withdrawal with code like the following:

```
1 if (balance - amount >= 0)
2     balance -= amount;
```

This code does not expose a reusable predicate: first, the guard condition depends on the quantity **amount**, which may vary from request to request; second, when the balance is low, the guard condition may be immediately falsified by the update.

As another example, consider implementing the stock-order transaction from a sharded version of the TPC-C benchmark (cf. [87]). Each item in the inventory has some amount of stock that is depleted as orders are processed. A direct application of treaties would use a treaty ensuring there is enough stock for the order amount before decrementing the stock. However, this can lead to many treaties that are specific to each possible order amount: orders of five items would check a treaty ensuring there are five items while orders of three items would check a corresponding three-items-left treaty.

To make reusable treaties easier to express, the programming interface allows the specification of *stipulations*, postconditions that must hold after some set of updates is applied. The updates are performed optimistically, but if the resulting state does not satisfy the stipulations, the updates are rolled back and are not committed (“begging forgiveness”). The application then has the opportunity to perform alternative actions. In the banking example, it is enough to check that a treaty of the form **balance** \geq 0 would still be valid *after* the update to the balance. This predicate is reusable because it does not mention the amount being withdrawn, and it is never invalidated by withdrawals.

```

1 Metric balance_us, balance_eu;
2 Metric balance = balance_us.plus(balance_eu);
3 int withdraw(int amount) {
4     atomic {
5         balance.requireStipulation(new LowerBound(0));
6         // withdraw from the appropriate shard
7         withdraw_locally(amount);
8         return amount;
9     } catch (StipulationFailure f) {
10        return 0;
11    }
12 }

```

Figure 5.11: Using stipulated commit to withdraw money from a sharded balance.

Actual code for the withdrawal transaction using stipulated commit is shown in Figure 5.11. In this code, the account balance is sharded across two sites in `balance_us` and `balance_eu`, which may become negative as long as their sum (`balance`) remains nonnegative. The keyword `atomic` starts a nested transaction that aborts and rolls back all of its updates if an exception occurs.

In cases where an existing treaty already asserts the postcondition, it is enough to check that the enforcement logic, described in Section 5.2.2, does not determine that the treaty is falsified by this transaction. Thus, stipulated commit uses treaties to avoid synchronization using the same underlying mechanisms. On the other hand, if there is no active treaty for a satisfied postcondition, a treaty is automatically created and activated, ensuring that later transactions can avoid synchronization in their postcondition checks.

While stipulated commit relies on a rollback mechanism, it has a subtle difference from previous mechanisms for nested transactions: reads performed when checking the treaty statement postcondition must be treated as part of the parent transaction, even if the postcondition fails. This ensures the serializability of application logic that depends on the failure. For example, in our TPC-C implemen-

tation this ensures that **NewOrder** transactions do not refill the stock unnecessarily due to inconsistent reads.

The core insight of stipulated commit is that programmers should be encouraged to use treaties which are likely to be used across many transactions. Directly specifying treaties for postconditions shared across large varieties of updates helps produce reusable treaties: many transactions may perform slightly different updates but all require the same postcondition for the effects. This insight could be captured in alternative designs. For example, an alternative design could provide an interface that better mirrors the traditional `if` statement from the banking example. However, we opted for this interface because it required fewer changes to the compiler and allowed us to leverage existing nested transaction support.

5.4 Automatically Creating Low-Coordination Treaties

As mentioned above, predictive treaties are automatically enforced by the system using strategies driven by estimates of the update model presented in Section 5.2.4. Here we discuss how the model parameters are estimated and how these estimates are used by the implementation to choose a strategy to enforce a treaty predicted to synchronize as little as possible.

5.4.1 Estimating Model Parameters for Metrics

In our system, numeric direct metrics create estimates of v and σ . These estimates, in addition to the current value m_0 , are used by the system when constructing predictive treaties. Derived metrics construct estimates of v and σ by appropriately transforming estimates from their children.

In the case of constructing direct metric parameter estimates, we treat this as

a matter of estimating distribution parameters using each update as a new sample. The model assumes that each update to a direct metric’s value comes from the random variable

$$dx = v dt + \mathcal{N}(0, \sigma^2 dt)$$

where dx is the change in value, dt is the time since the last observation, v is the velocity parameter, σ is the noise parameter, and $\mathcal{N}(0, \sigma^2 dt)$ is a normally distributed random variable with a mean of zero and variance of $\sigma^2 dt$. Rearranging the distribution to get v , we have

$$v = \frac{dx - \mathcal{N}(0, \sigma^2 dt)}{dt}.$$

Since $\mathcal{N}(0, \sigma^2 dt)$ has a mean of zero, we expect that an estimate of $\frac{dx}{dt}$ gives an estimate of v . Similarly, rearranging to get an expression of σ , we get

$$\mathcal{N}(0, \sigma^2 dt) = dx - v dt.$$

This implies that estimating the variance of $dx - v dt$ gives an estimate of $\sigma^2 dt$, so σ is estimated as the square root of the variance $dx - v dt$ divided by dt .

Direct metrics track sampled means of dx , dt , and $dx - v dt$, as well as the sampled variance $dx - v dt$ to construct the v and σ estimates.⁴ In our system, we use exponentially weighted moving averages (EWMA) and an exponentially weighted moving variance for these sample statistics [21, 49, 72]. With EWMA, past behavior of the metric is forgotten over time, allowing the system to more rapidly adapt to shifts in the workload behavior at the price of lower accuracy for stable workloads. In our implementation, we use a parameter of $\alpha = 0.001$ for EWMA, keeping an effective window of 1,000 samples for each statistic.

To ensure that our estimation techniques are reasonably accurate across a variety of underlying update distributions, we simulated using our estimation techniques on various scenarios and looked at the relative error between the estimates

⁴Here we use the current estimate of v in the $dx - v dt$ terms.

Table 5.1: Mean relative error of parameter estimates for various scenarios. Relative errors are averaged across 1,000 trials with 10,000 updates each. Each row shows a different distribution for the amount the value is updated after each interval.

	Fixed Interval ($dt = 2$)	Exponential Interval ($\mathbb{E}[dt] = 2$)
Brownian ($v = 1, \sigma^2 = 0.5$)	$v_{\text{err}} = 0.91\%, \sigma_{\text{err}}^2 = 2.4\%$	$v_{\text{err}} = 0.87\%, \sigma_{\text{err}}^2 = 3.5\%$
Constant (+1)	$v_{\text{err}} = 0.0\%, \sigma_{\text{err}}^2 = 0.0\%$	$v_{\text{err}} = 1.8\%, \sigma_{\text{err}}^2 = 4.0\%$
Binary (25% +1, 75% -1)	$v_{\text{err}} = 3.2\%, \sigma_{\text{err}}^2 = 2.1\%$	$v_{\text{err}} = 3.7\%, \sigma_{\text{err}}^2 = 2.4\%$
Gaussian ($\mu = 1, \sigma^2 = 0.5$)	$v_{\text{err}} = 1.3\%, \sigma_{\text{err}}^2 = 2.4\%$	$v_{\text{err}} = 2.3\%, \sigma_{\text{err}}^2 = 3.5\%$

produced for velocity and noise and the actual values based on the underlying distribution. We looked at a total of 8 scenarios across 2 distributions for intervals between updates (dt) and 4 distributions for update values (dx). For intervals, we looked at scenarios where dt was a constant value and scenarios where the dt followed an exponential distribution with a given target mean value. For updates we looked at cases where the underlying dx were selected to following one of the following distributions:

- Brownian motion with drift, using dt to choose dx . This allows us to validate that we can recover the modeled distribution’s parameters.
- Constant valued updates. This simulates scenarios like simple counter increments.
- Binary updates, where updates are selected from a discrete distribution across +1 and -1. This simulates a scenario similar to how the vote margin changes in the voting scenario.
- Gaussian updates, with set mean and variance. This captures a reasonable alternative assumption about a value’s update distribution: in this case, values are *independent* of the interval between updates.

The average relative error of the velocity and noise parameter estimates across 1,000 trials with 10,000 updates sampled in each trial is given in Table 5.1. The

relative error is no more than 4% across all scenarios. These results indicate two points:

1. Our estimation is reasonably accurate at recovering the original parameters when the underlying distribution is a Brownian motion with drift.
2. When the underlying distribution for updates is some reasonable alternative, the estimation obtains reasonably accurate parameters for treating the distribution as if it were truly a Brownian motion with drift.

5.4.2 Automatically Choosing an Enforcement Strategy

When using predictive treaties, programmers are only required to specify the top-level treaties used by their application. Any subtreaties used to enforce that treaty and avoid synchronization are automatically chosen by the runtime system. Subtreaties are chosen by a recursive procedure that starts from the top-level treaty metric and works down the metrics tree. At each derived metric m (a parent node in the metrics tree), subtreaties are chosen for the submetrics using a two-step procedure similar to syntax-directed translation in compilers [7] and to query planning in databases [85]. Our work has explored the case of a particular choice of prediction model (Brownian motion with drift), derived metrics (sums, scaling, minimums, and maximums), and predicates (equalities and inequalities on numeric data). However, this approach generalizes to other choices of prediction model and further kinds of derived metrics and predicates.

First, our implementation obtains templates for the subtreaties, similar to the local treaty templates used in the homeostasis protocol [87]. Templates are predicates with parameters to be filled with specific values. The subtreaty templates are determined by calling the `policy` method on the derived metric, passing in the and the form of its treaty's predicate. For example, if the derived metric is a

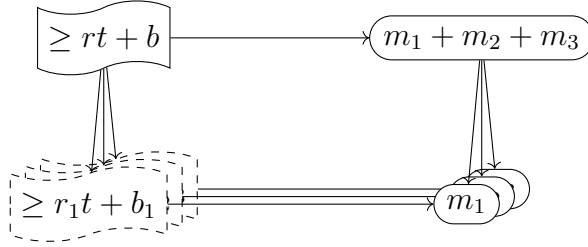


Figure 5.12: During the first step of treaty generation, subtreaty predicate templates are constructed by the derived metric based on the predicate being asserted. In the second step, the template parameters are filled by the system to using a numerical solver to select parameters that maximize the expected time until any of the subtreaties are expected to synchronize.

sum and the treaty’s predicate is a threshold, as in the voting example, the system may choose subtreaty templates specifying thresholds on each summand. Thus, in the voting example in Section 5.1, each subtreaty γ uses a template of the form “ $\geq r_\gamma t + k_\gamma$ ”, with parameters r_γ and k_γ , as diagrammed in Figure 5.12. Alternatively, if the derived metric is a minimum of other metrics and the treaty predicate is the strict equality $\min(x, y, z) = 5$, the templates chosen by the system could be an equality for the current minimum metric argument, and thresholds for the other arguments. If x were the current minimum argument, subtreaty templates would take the form $x = x_\gamma$, $y \geq y_\gamma$ and $z \geq z_\gamma$. Extending the system to support additional derived metrics and predicates requires updating the **policy** method implementations to appropriately handle new combinations.

Second, the parameters in subtreaty templates are filled with specific values chosen by solving a constrained optimization problem. This problem corresponds to maximizing the predicted time until any subtreaty will become invalid, subject to constraints that ensure the subtreaties imply the original treaty holds. There are two sets of constraints used in the problem, projected expiry constraints and correctness constraints. The resulting constrained optimization problem ensures the subtreaties chosen are expected to avoid synchronization as long as possible

according to the prediction model.

Projected expiry constraints determine predictions for how long each subtreaty will hold based on a prediction model, like the one discussed in Section 5.2.4. Extending support for additional predicates or alternative prediction models for metric values requires determining new projected expiry constraint equations.

Correctness constraints ensure that the selected parameters produce predicates on the submetrics that, in conjunction, imply the original predicate on the derived metric holds. Extending support for additional combinations of predicates and derived metrics requires determining new correctness constraints.

For example, consider the two-station voting scenario in Section 5.1, where the first and second stations have respective margins m_1 and m_2 , with estimated velocities v_1 and v_2 , and noise σ_1 and σ_2 . With the treaty $m_1 + m_2 \geq 0$ and templates $m_1 \geq r_1 t + k_1$ and $m_2 \geq r_2 t + k_2$, the system solves the optimization problem:⁵

$$\begin{aligned} & \arg \max_{r_1, r_2, k_1, k_2} (\min(t_1, t_2)) \\ & \text{such that} \\ \text{Projected Expiry} & \begin{cases} m_1 \pm \sigma_1 \sqrt{t_1} + v_1 t_1 = r_1 t_1 + k_1 \\ m_2 \pm \sigma_2 \sqrt{t_2} + v_2 t_2 = r_2 t_2 + k_2 \end{cases} \\ \text{Correctness} & \begin{cases} k_1 + k_2 \geq 0 \\ r_1 + r_2 \geq 0 \end{cases} \end{aligned}$$

The system solves for values of the parameters r_1, r_2, k_1, k_2 that maximize the shorter of two projected expiration times t_1 and t_2 . The first two constraints are projected expiry constraints, determining t_1 and t_2 . The remaining constraints are correctness constraints to ensure that when the thresholds on the individual values

⁵Here, we are accounting for a single standard deviation of noise on each metric. For clarity, we elide some rewriting of these formulae to reduce the parameter space, such as requiring equality for the final two conditions, and elide checks to ensure that assumptions hold, such as the treaty being currently valid.

hold, the treaty on the sum also holds.

For simple cases, our implementation directly solves for parameters; for example, the treaty $\min(x, y) \geq 5$ using templates $x \geq a_\gamma$ and $y \geq b_\gamma$ has optimal parameter value 5 for both a_γ and b_γ . In more complex cases, our implementation solves for parameters using a numerical optimization library.

5.5 Implementation

We implemented predictive treaties and the API described in Section 5.3 on top of Fabric [65].⁶ Fabric is a persistent programming language that supports nested, distributed transactions. Fabric’s security features are not germane to this work, but its support for linearizable multistore transactions and optimistic concurrency control make it a good fit for the geodistributed setting. However, we made a few changes to Fabric to support predictive treaties.

Ignoring comments and blank lines, the implementation added about 5,000 lines of FabIL and Java code to implement the API and changed about 4,000 lines of Java code in the Fabric runtime.

5.5.1 Integration with Distributed Transactions

While the majority of the implementation is written in FabIL and implements the enforcement and construction procedures defined above, small modifications to Fabric’s transaction management system were needed to support predictive treaties.

Fabric implements distributed transactions using optimistic concurrency control and 2PC [4]. Fabric worker nodes optimistically use local copies of objects

⁶This implementation is orthogonal to the implementation used in Chapters 3 and 4.

to compute transactions and then act as coordinators with the stores holding the persistent versions of these objects.

During the prepare phase, the coordinator ships version numbers associated with the objects read and written to be checked to ensure they're consistent with the persisted version. If all objects used are consistent and can be successfully locked by the prepare phase, the coordinator then runs the commit phase. Otherwise, the coordinator aborts, sending messages to stores indicating they may release the locks acquired during prepare.

Checking the expiration of predictive treaties in 2PC Fabric's transactions are strictly serializable, so a transaction's reads and writes behave as if they happen atomically in a single step. Because a predictive treaty's validity depends on the relation between t_{expiry} and the current time, the use of a predictive treaty in a transaction must behave as if it was performed at the time the transaction was committed. Therefore, the transaction protocol must determine for each transaction a *commit time* that respects strict serializability.

Our implementation sets the commit time of a transaction to be the latest time at which the prepare phase finished read- and write-locking the persisted objects at any of the stores. This commit time respects strict serial ordering, because it is guaranteed to be within the period of time the transaction appears to have occurred and will be strictly before or after the times other (possibly conflicting) transactions are applied.

We modified Fabric's prepare-phase responses to include the time after all objects at the store have been prepared. The coordinator is modified so that, if there are no failed prepares, the latest of these timestamps is compared against the expiration times of predictive treaties that appear valid to the transaction. If all predictive treaties expiration times are later than the commit time, the coor-

dinator sends out a commit message. Otherwise, an abort message is sent and the coordinator retries the transaction. During the retry, the new attempt will either observe the treaties used as invalid or updated with longer expiries by other transactions.

Specialized versioning for predictive treaties We specialized Fabric’s handling of version numbers for predictive treaties, to help avoid *false conflicts* between transactions and increase concurrency. In particular, Fabric was modified to only increment the version number of a predictive treaty when the expiration time was *retracted* to an earlier time. This is safe for consistency because in any case when the coordinator’s version could possibly be later than the persisted value the coordinator’s copy and the persisted object will have different version numbers. However, the coordinator’s copy could be “old” with the same version number, but it would only mean the coordinator’s version has a more conservative expiration time. To help avoid using overly conservative expiration times, successful prepare responses include updated expiration times for the predictive treaties read by the transaction persisted at that store.

Lazy metric update processing Recall that when metrics change value, the expiration times of predictive treaties may need to be updated to ensure consistency. A transaction may change the value of many metrics at once, and it is possible that a transaction’s aggregate changes do not affect the expiration time of a predictive treaty while individual writes of the transaction may have. To avoid unnecessary and possibly redundant computation of changes that result from updated direct metric values, these resulting effects are lazily determined only when either the transaction completes or a treaty or derived metric that depends on the updated direct metric is read by the transaction.

5.5.2 Opportunistic Slack Reallocation

Distributed synchronization is sometimes required because some treaty is potentially falsified. Coordination allows the involved nodes to determine whether the treaty is actually falsified and if not, to determine new locally consistent subtreaties that suffice to enforce the treaty. When synchronization already must occur, it saves work to piggyback on this existing synchronization to reallocate slack for other treaties that are still known to be consistent but no longer optimal for updated model projections. Other treaties involving the synchronizing nodes can be identified, and new local subtreaties can be constructed for these other treaties, to balance slack better among the involved nodes than the existing subtreaties do. This opportunistic reallocation of slack further reduces synchronization and is important for performance of some applications (see Section 5.6.2).

5.6 Evaluation

Our evaluation aims to address several questions:

1. Do predictive treaties reduce synchronization? (Section 5.1)
2. How does bias difference affect synchronization? (Section 5.6.1)
3. How does this scale with the number of sites? (Section 5.6.1)
4. What happens when the model's assumption of a stable update trend does not hold? (Section 5.6.1)
5. Does hierarchy reduce synchronization costs? (Section 5.6.1)
6. Do predictive treaties work on realistic workloads? (Section 5.6.2)
7. How does performance compare with prior related techniques? (Sections 5.6.2 and 5.6.3)

8. Does stipulated commit help performance? (Section 5.6.3)

The first question is answered by our initial results given in Section 5.1: using predictive treaties in the voting example can significantly reduce synchronization. We now explore the remaining questions.

5.6.1 Voting Microbenchmark

In Section 5.1, we implemented the voting example as a microbenchmark. We make further use of the microbenchmark to investigate questions 2–5.

Behavior with different biases We ran a series of variations of the voting example in which we varied the voting bias of the pro-*A* station to see how the results change with the overall bias in the system. In each experiment, the pro-*B* station is biased with 48% of votes for *A*, as in the previous experiment, and the pro-*A* station prefers *A* at 56% and 60%. As in Section 5.1, for each configuration we ran 100 trials where the system votes for 30 seconds and then create a treaty that asserts the current winner is in the lead. We then measure the time that elapses until either a query or voting transaction must synchronize with a remote station, stopping after 400 seconds if there are no synchronizations. This measured time captures how often the voting application would require client transactions to synchronize.

The results are shown in Figure 5.13. When the overall bias of the system trends toward favoring neither candidate, there is less time between synchronization for all strategies. This occurs because the expected margin between the two candidates is lower and therefore there is less slack to allocate across the two sites. With predictive treaties, synchronization is avoided in the majority of trials when trends are stable, even in less biased scenarios. In nearly all cases where the trials hit

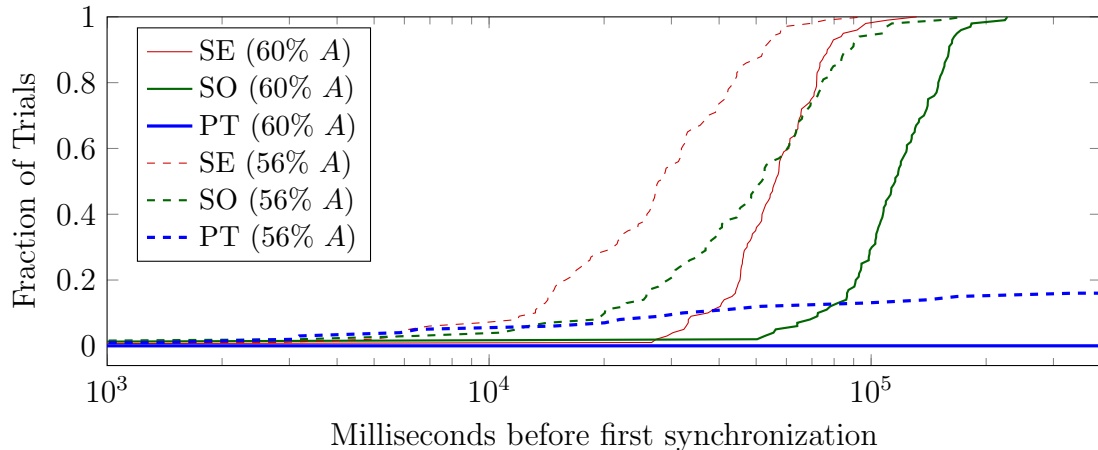


Figure 5.13: CDF of time until first synchronization under the three strategies for slack allocation with varying bias: static equal (SE), static optimal (SO), and predictive treaties (PT). Bias was varied at the first station. Stations received 100 votes per second.

the cutoff time, the system state was such that the treaties would continue to hold indefinitely; slack was increasing in all stations, reducing the likelihood of synchronization.

Scaling with number of sites To see how our approach scaled with the number of voting stations, we ran additional comparisons using 4 and 8 voting sites, again with 100 trials. Each additional pair of voting stations had the same biases and voting rates as the two stations in the previous experiment, ensuring that the overall system bias was the same, 54% votes for candidate A overall, while the overall rate of voting scaled with the number of stations. The measurement is the same as before: the distribution of time from the treaty being created until a client transaction needed to synchronize with a remote station.

The results of this experiment are shown in Figure 5.14. For static strategies, the time until the first synchronization with either static strategy falls as the number of stations increases. With predictive treaties, few trials ever synchronize even in the largest configuration. This is because the predictive treaties are time-

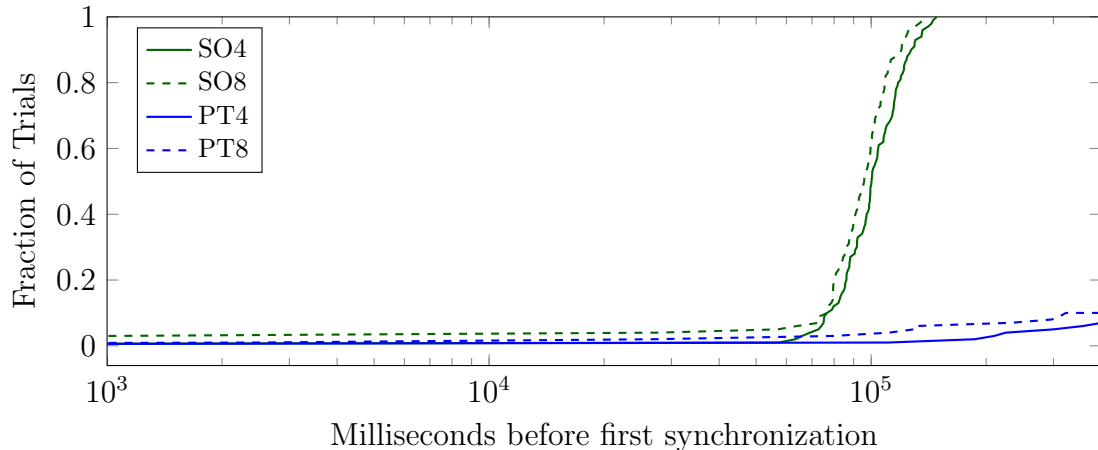


Figure 5.14: CDF of time until first synchronization in the voting system under static optimal (SO) and predictive treaty (PT) strategies for slack allocation with 4 and 8 stations. With predictive treaties, synchronization is rarely needed.

varying and implicitly shift slack.

Adapting to changing update trends Predictive treaties are extremely effective in avoiding synchronization in scenarios where updates exhibit a stable trend. This is the assumption of our predictive model: past trends in updates can be used to predict future behavior. However, in many realistic workloads this might not be the case: a video may suddenly go viral or the underdog candidate’s voters show up in strength later in the day. To evaluate how well predictive treaties adapt to a sudden change in update trends, we ran 10 trials of a scenario where the bias in voting suddenly changes after a period. We started the system with 2 stations and an overall bias of 54% pro-*A* and ran it for 10 seconds of warm-up, voting only, followed by 2 minutes with querying. Then, clients flip the voting bias to a new overall bias of 46% pro-*A*, which we ran for another 13 minutes. During this period, *B* is expected to pull ahead of *A*.

In this and following experiments, network latencies between locations are set to simulate round-trip times (RTT) between different Amazon EC2 regions, based

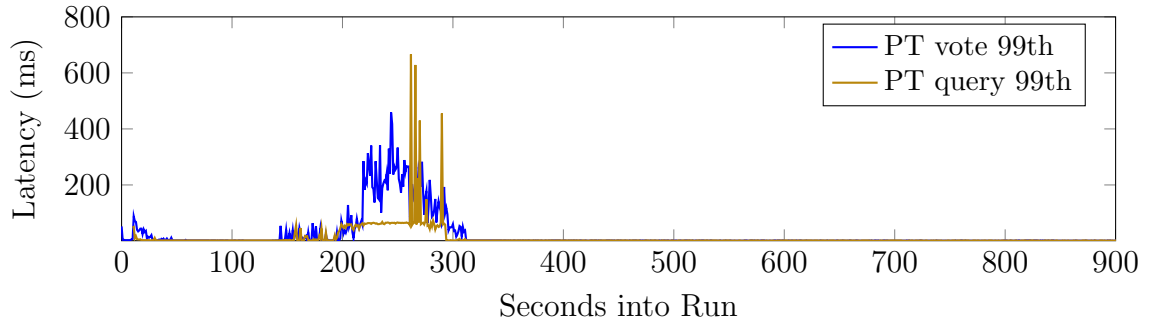


Figure 5.15: 99th percentile vote and query latencies per second of 2-station adaptivity test.

on real latency data from Roy et al. [87]. This allows us to validate system behavior under realistic geodistributed conditions. The RTT between each pair of regions ranges between 64 and 372 ms.

In Figure 5.15, we see the results of this experiment with 2 stations using predictive treaties measuring the 99th percentile latencies of votes and winner queries throughout the run. After the shift in bias, the votes start to tip the margin in favor of B . During this tipping period votes take longer because they must check whether they require retractions for the pre-existing treaties, stating that A is still the winner, and queries take longer as the treaties enforcing the previous result are being retracted. However, eventually the system stabilizes to a new bias and winner, and tail latencies become relatively stable again, with occasional spikes to adjust to new subtreaties. Thus, we see that predictive treaties are able to adapt to changes in bias.

Benefits of hierarchical structure Another feature of predictive treaties relevant to adapting to changes in bias is that predictive treaties can be constructed *hierarchically*. Hierarchical structures allows updates to avoid synchronizing to reallocate slack for the top-level treaty whenever it violates a local treaty, reducing the number of locations the update synchronizes with. To demonstrate the

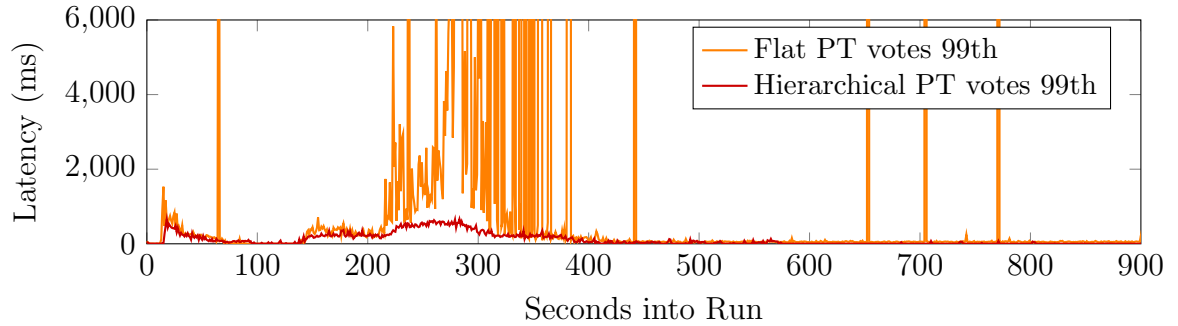


Figure 5.16: 99th percentile vote and query latencies for four-station adaptivity test with and without hierarchical treaties. Hierarchy helps avoid large latency spikes.

effectiveness of hierarchy, we ran 10 trials of the same experiment but with four stations. In this case, station three voted similarly to one but in Ireland and station four voted similarly to two but in Singapore.

The result of this comparison in Figure 5.16 shows that the maximum 99th percentile latency per second of vote operations rises much higher for the flat organization. In the flat organization, all synchronization occurs across all four sites, increasing peak latencies and creating more contention with other transactions. However, the hierarchical organization allows synchronization to sometimes be localized to pairs of sites that are nearby each other, reducing peak latencies.

5.6.2 Distributed Top- k Monitoring

Babcock and Olston [10] showed how to efficiently monitor the top k items from a set with sharded counts, by tracking the validity of constraints across the distributed system. We implemented a simpler alternative top- k monitoring algorithm, taking advantage of predictive treaties to automatically construct and maintain related constraints.

In this scenario, counts for a set of identifiers are incremented periodically across a number of geodistributed servers; the goal is to be able to quickly query

the identities of the top k items in the set. Our algorithm introduces a pseudo-item that we call the *marker*. Its count always lies between that of the k -th and $k + 1$ -th items. The algorithm maintains global predicates asserting that the items in the current top k all are above the marker, and that the rest of the items are all below. Our framework automatically maintains these predicates. For this problem, opportunistic renegotiation is particularly effective, because slack can be rebalanced for many local treaties at synchronization points.

To evaluate this algorithm, we used a benchmark based on the HTTP request logs for the 1998 FIFA World-Cup website, which was served from 33 servers distributed across 4 regions [9]. This benchmark has been used in related work [10, 35]. For comparison purposes, we implemented the more complex algorithm of Babcock and Olston in the Fabric system. Unlike prior work, both implementations guarantee strict serializability for updates and queries.

We created a top-level predictive treaty that queried for the 20 most popular pages. We ran two different tests with this data: a 24 hour run with 52 million requests, the heaviest traffic day,⁷ using a single server for each of the 4 regions (Figure 5.17), and an hour run with 84,000 requests⁸ with 33 servers across the 4 regions as in the original scenario (Figure 5.18). In both scenarios, a client requested the current top 20 page identifiers every second, and both implementations ensured the top-20 set was up to date at all times.

The 24 hour run is shown in Figure 5.17, with the number of synchronizations performed for each implementation plotted relative to the time in the logs. For the first half of the day, there is little difference between the specialized protocol and our treaties implementation. After that, however, the Babcock and Olston protocol starts synchronizing much more than the predictive treaties implementation.

⁷This was day 47 in the dataset.

⁸This was from the first hour of day 30 in the dataset.

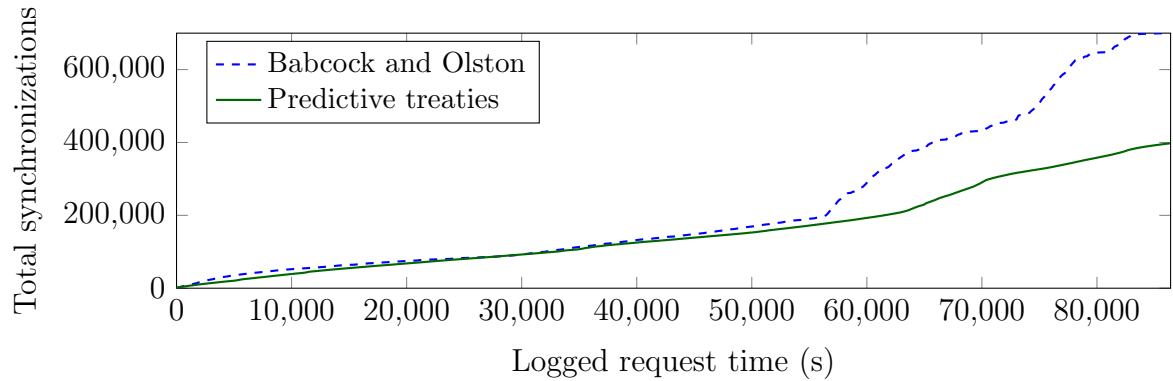


Figure 5.17: Using synchronizations to compare Babcock and Olston’s top- k algorithm with using predictive treaties with four servers over 24 hours of page hits.

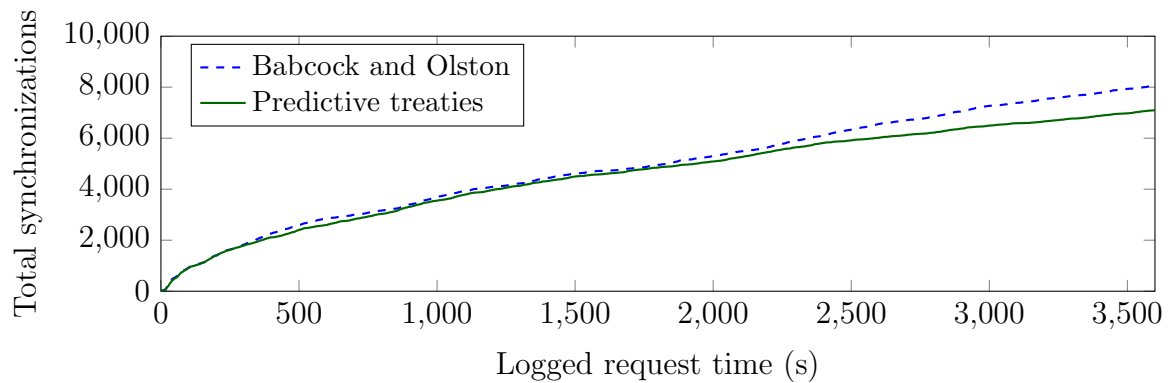


Figure 5.18: Using synchronizations to compare Babcock and Olston’s top- k algorithm with using predictive treaties with 33 servers over one hour of page hits.

This corresponds to a shift in the request data patterns, with an increased load across sites and a shift in which regions were receiving more requests. While both implementations started synchronizing more frequently, the predictive treaties implementation does not experience as dramatic of an increase. This difference comes from the ability for predictive treaties to adapt to changes in the data trends, which the slack-allocation heuristic for the Babcock and Olston protocol is not designed to do.

In the second run, shown in Figure 5.18, the results give a finer detailed look at the difference between the two implementations with a configuration closely

mirroring the original system’s organization. This plot, like the 24 hour run, shows that our algorithm synchronizes less by the end of the experiment, again delivering better performance than that of a more complex algorithm specialized to the problem.

In addition to performing less synchronization, the treaty implementation was also simpler than the Babcock and Olston protocol: its update routine was 100 lines of code, half as many as the 210 lines of code for our implementation of Babcock and Olston’s algorithm.

5.6.3 Modified TPC-C

The TPC-C benchmark allows us to compare with prior work and to validate that our approach scales to a larger benchmark. TPC-C is an OLTP⁹ benchmark that simulates a system for order entry and fulfillment.

For purposes of comparison, our variant of TPC-C is based on the one used by Roy et al. [87]. We similarly shard the database across multiple stores, with each item’s stock sharded across the stores. We make the realistic assumption that items are ordered with a nonuniform popularity, skewed across both items and the locations where they are ordered.

As in Roy et al., the database is initialized with ten warehouses, ten districts per warehouse, and 100 customers per district. There is an inventory of 1,000 items, for a total of 100,000 Stock objects. Initial stock levels are set randomly between zero and 100.¹⁰ There are no orders in the initial state. Each store, along with an associated eight clients, experiences latency simulated to act like one of the EC2 regions.

⁹OLTP (Online Transaction Processing) applications handle online transaction requests from external clients.

¹⁰Random values are drawn from a uniform distribution.

Some object fields are never written by the benchmark. We do not validate reads of these fields. Object fields that are written (except ones in **Order** and **OrderLine** objects) are sharded across a number of geodistributed data stores. For instance, each customer object tracks a customer’s account balance. Each data store maintains a local balance for that customer, reflecting credits and debits made by clients local to the data store. Adding a charge or a debit can be done locally, but reading the customer’s balance would incur a synchronization across all shards.

The workload consists of two of the TPC-C transactions, based on the two potentially distributed operations of the three most frequent transactions in TPC-C. The **NewOrder** transaction orders a random quantity (between one and five) of a random item from a random district at a random warehouse. If there is insufficient stock to meet the order quantity, item stock is first replenished by adding 100 more items before decrementing the stock amount. The **Delivery** transaction enqueues the oldest order at a random warehouse and district for deferred processing. A thread at each warehouse later fulfills the order and charges the appropriate customer. Except when comparing our baseline with the performance reported by Roy et al., we do not include the **Payment** transaction, the remaining of the three most common transactions. **Payment** transactions do not require synchronization; they pad out the workload with operations that don’t have read–write conflicts with the other two transactions.

Like Roy et al., we avoid synchronization on every **NewOrder** transaction by relaxing the requirement for globally monotonic order IDs. Instead, they are generated monotonically on a per-shard basis. To determine the oldest order, the **Delivery** transaction requires a total ordering; we obtain one by breaking ties with the shard ID. Like Roy et al., we report **NewOrder** latencies as a distribution

plot that captures the core system performance; throughput is directly affected by how long operations take, longer latencies leads to more bottlenecks and contention in the system producing worst throughput. Furthermore, these plots help identify the percentage of orders which coordinated or experienced contention with other transactions, where latency is nontrivial.

Lazy balancing baseline For a performance baseline, we use a simple algorithm for sharded TPC-C orders that we call *lazy balancing*. It tries to fulfill orders entirely locally, but when the current store lacks enough stock, it synchronizes with the other stores to obtain the missing stock, and divides remaining stock equally among the stores. Lazy balancing does not pay any cost for setting up treaties, and performs especially well when there is no bias across stores, because all stores run out of stock around the same time and hence, treaties do not offer much performance benefit.

To determine whether lazy rebalancing is a competitive baseline, we compared it against the results published for the homeostasis protocol by Roy et al. [87]. When running the same TPC-C workload with the same network configuration they reported, lazy rebalancing achieves a 90th percentile **NewOrder** latency of 130ms and a 99th percentile of 200ms, whereas Roy et al. reported a 90th percentile of roughly 260 ms and a 99th percentile of well over one second.

The next two experiments involve a mix of 95% **NewOrder** and 5% **Delivery** transactions. The system is given ten minutes of warm-up time so caches heat up and model parameters reach a steady state, followed by ten minutes of measurement.

Benefits of stipulated commit To demonstrate the benefits of stipulated commit, we implemented one version of the benchmark using stipulations, similar to

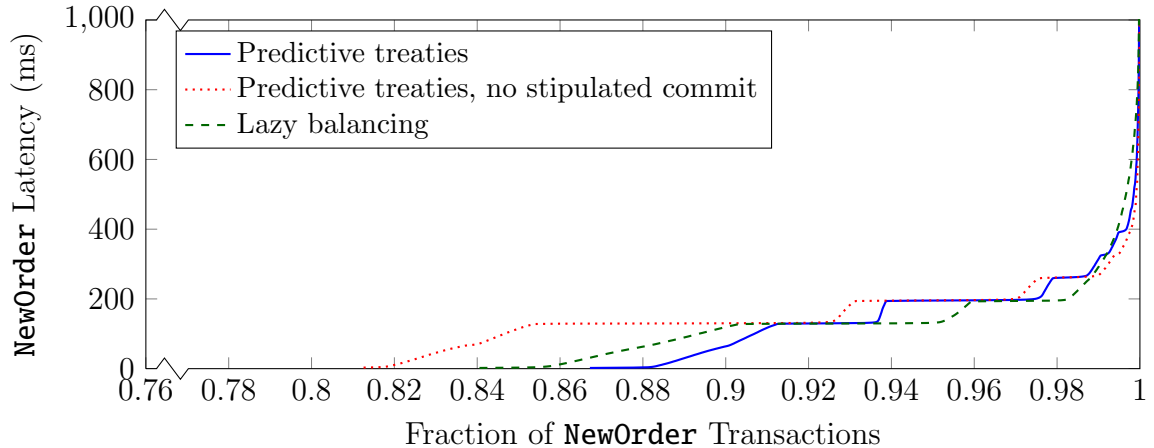


Figure 5.19: CDF of latencies for TPC-C **NewOrder** transactions, run on two sharded stores with geographic round-trip latency, with 50% of orders going to hot items uniformly across sites. Stipulated commit allows performance of predictive treaties to be comparable to that of lazy balancing.

the withdrawal example in Section 5.3.2, and one using precondition treaties; both enforce the invariant that total stock is positive for all items. These treaties allows clients to remove stock from the local region without synchronizing to ensure there was enough stock to accommodate oversales across the entire database.

We compare the performance of these two implementations with lazy balancing in a scenario with *globally* popular items: 1% of the items are ordered 50% of the time. The symmetry of this scenario makes it favorable to lazy balancing, but Figure 5.19 shows that in a 2-shard scenario, predictive treaties using stipulated commit perform similarly.¹¹ Without stipulated commit, the latencies are higher with predictive treaties. Stipulated commit allows the application to avoid creating treaties tailored to specific order amounts.

Skewed popularity We also evaluated a second scenario to which predictive treaties are particularly well suited: skewed popularity across shards. In these experiments, each replica has a share of “locally hot” items, and a majority of

¹¹To facilitate comparison with prior work [87], the CDF is oriented with probability along the horizontal axis, so the area under the curve is proportional to expected latency.

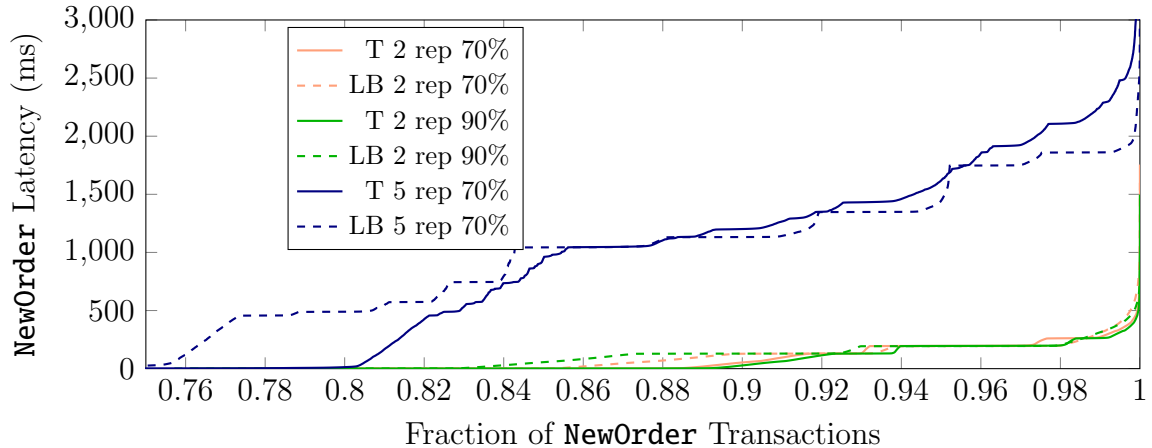


Figure 5.20: CDF of latencies for TPC-C **NewOrder** transactions with skewed order distribution across two and five replicas using lazy balancing (LB) and treaties (T).

orders on each replica go to its locally hot items. This results in a skewed distribution of updates for each item across its sharded values, with most orders for each item happening at the replica where it is locally hot. We see the results for a skewed order distribution in Figure 5.20. Predictive treaties allow the system to adapt to uneven popularity across replicas, reducing synchronization over a lazily balanced implementation. With two replicas and 70% of orders going to locally hot items at each replica, the lazily balanced implementation synchronizes on 10.5% of orders, while the treaties implementation only synchronizes on 8%. In the highly skewed case where 90% of orders go to locally hot items, this gap in synchronization widens: the treaties implementation synchronizes on only 7.5% of orders, while lazy balancing synchronizes on 12.5%.

5.6.4 Discussion

Predictive treaties are particularly effective when treaties' slack grows over time, as in the voting and top- k benchmarks. In this case, predictive treaties improve on previous techniques by rebalancing slack in the background, avoiding synchro-

nization during client operations.

When, as in TPC-C, global slack does not grow, slack cannot be continuously rebalanced. However, the results show that predictive treaties still have benefits. The predictive model allows the system to *automatically* identify and adapt to the trends and distribution of slack across nodes.

In the unfavorable case where the workload violates the predictive model’s assumption of stable trends, the system can compensate by detecting and adapting to new trends. Organizing treaties hierarchically helps reduce overheads in these chaotic scenarios by restricting synchronizations to small local subsets of the nodes when possible.

5.7 Related Work

Many prior projects have investigated methods for avoiding contention and synchronization in applications by leveraging application-specific semantics. Even in single-store systems, the notion of using higher-level semantics to better manage contention has been proposed, including hierarchical reader-writer locking [41] and predicate locks [31]. Particular attention has been paid to this idea in the distributed application setting, with recent work trying to identify rules for when synchronization is unnecessary [11, 108].

The running example of the voting application has some similarities with the bancomat problem [98], a scenario where the application manages funds across multiple automated teller machines. The bancomat problem was introduced to compare different approaches in terms of *cushion*, the maximum amount of money that can be lost in the system due to asynchronous message loss and regrouping in a dynamic group membership system. In our work, however, we are not focused on considerations regarding network partitions.

Predictive treaties are designed to improve performance of applications built on top of strongly consistent systems by enforcing application invariants. Some work instead starts with efficient systems with weaker consistency guarantees, such as eventual [107] or causal consistency [68], and introducing techniques such as reservations [83] or CRDTs [93, 92, 14] to enforce stronger guarantees where necessary.

In particular, Indigo [13] allows creating and using reservations to enforce programmer-specified application invariants in a causally consistent setting. Unlike predictive treaties, Indigo’s invariants are statically specified at compile time. They are limited to predicates expressible in first-order logic; for example, Indigo’s annotations could not enforce a graph connectedness invariant because it is not expressible in first-order logic [29] without further restrictions on the application, such as knowing all vertices in the graph at compile time. Predictive treaties do not have this limitation because they are generated at run time.

Like predictive treaties, some work focuses on monitoring distributed results that may not be invariant for the lifetime of the application, such as results of computation on stored state or the values seen in a stream [26]. Some of this work has examined thresholds on vector values [94, 53] and predictive models [37], but focuses on settings like sensor networks where strong consistency is not required.

Leveraging similar insights to anticipate how remote values will continue to behave, distributed simulations and games use dead reckoning [95, 99, 77]. This technique extrapolates the last known state and behavior of remote objects to perform tasks such as generating visuals. Dead reckoning is useful when immediately computing an inconsistent result is better than blocking the program to ensure a consistent result. In contrast, predictive treaties use a predictive model to make consistency cheaper.

Similar to the goal of predictive treaties for providing a basis of high level strong

guarantees on the system state, Conits [111] aims to provide high level consistency guarantees which allow for a continuous trade-off between performance and consistency in a manner similar to epsilon serializability [84]. Similarly, systems like Pileus [101] offer APIs to directly specify SLA-style guarantees on reads and updates, allowing the application to explicitly accept weaker consistency behavior for operations that are likely lower latency. However, these guarantees are primarily concerned with consistency of individually read and updated items whereas predictive treaties are intended to construct high-level semantic guarantees.

In settings that require stronger consistency guarantees, problems such as monitoring the top k items in a ranked listing [10], thresholds on a single quantity [76], or thresholds on linear combinations [15, 87] have been studied. Prior work similarly divides a slack-like resource between nodes. MDCC [55] applies similar techniques to provide better concurrency for georeplicated values, processing transactions that commute without determining an explicit ordering. However, this work is focused on either specialized scenarios or guarantees on individual objects and has not leveraged predictive models nor time dependence to shift slack.

Warranties [66], like predictive treaties, allow for compositional predicates built from arbitrary computations. These computations were more general than metrics but assertions are limited to state on a single storage node. Like predictive treaties, warranties have time limits; however, once created, they cannot be revoked before they expire.

Both predictive treaties and warranties leverage compositionality to ensure that enforcement checks recompute only the subcomputations possibly affected by an update. Recomputing only the affected subcomputations to update a result has been explored in work on incremental self-adjusting computation [1, 18]. Incoop [18] applies this technique to Hadoop clusters. RDDs [112] use a similar tech-

nique for a more limited class of distributed applications in a cluster. In databases, this technique is used for incremental view maintenance [44], and TxCache [82] offers similar functionality for web applications. However, these techniques were not designed for high-latency, geodistributed settings.

5.8 Discussion

Predictive treaties and metrics are new abstractions for constructing low overhead system-scope warranties to enforce assertions over geodistributed state. Building upon the computation warranties design in Chapter 4, predictive treaties are designed to avoid synchronizing recomputation as much as possible by automatically selecting stable subexpressions to monitor. To support this, predictive treaties are defined in terms of metrics that can be maintained locally, computed hierarchically, and are associated with a prediction model for projecting how their values will evolve. Predictions help the system determine the lowest synchronizing subexpressions to monitor to enforce a system-scope warranty. Furthermore, predictive treaties support *time-varying* expressions, which can sometimes be much more stable than static expressions. Our results show that these new abstractions permit programs to be straightforwardly built in terms of predictive treaties and metrics, with significant performance benefits.

CHAPTER 6

CONCLUSIONS

In this dissertation I discuss a new abstraction for providing strong consistency with improved performance over traditional methods: warranties. Warranties provide time-dependent guarantees on the system’s state, which the application can use to avoid synchronization between nodes in the system. This reduced synchronization helps to reduce load on the system and reduce transaction latencies, providing improved scalability without sacrificing consistency.

6.1 Public State Warranties

Public state warranties demonstrate how time-limited guarantees can generalize optimistic concurrency control by allowing clients to optimistically use locally cached data without validating the value read at commit time. This leads to reduced load on stores serving popular data which is infrequently updated.

To ensure strong consistency, stores block updates that would invalidate an issued public warranty’s guarantee until the warranty expires. By using an adaptive model, warranties’ time limits are set to benefit as many readers as possible while avoiding blocking writes which would invalidate the guarantee.

We demonstrated that public state warranties improve throughput and latency on standard transaction benchmarks as well as a port of the Cornell CS department’s course management web service.

6.2 Computation Warranties

Warranties can be generalized to arbitrary predicates, which allows applications to check high-level statements such as “Who is currently winning?” and “Are there at least 5 seats on this flight?” rather than explicit stored values, such as vote

counts at each station or the currently booked seats on a plane. These high-level statements can be used as a form of distributed strongly consistent memoization—applications can use the asserted predicate result in place of performing a distributed computation.

Computation warranties, unlike state warranties, do not require an effective *freeze* of the underlying state to remain true. As long as changes to the underlying state do not affect the asserted predicate result, they need not be delayed by the system. This avoids contention in scenarios where the underlying data may be updated frequently but not affect application checks over the data, such as relationships between frequently modified objects. To support this behavior, the store must check the *effect* of an update to data used by a computation and determine if the result has *visibly* changed. These update checks can be done incrementally when computation warranties are constructed compositionally using other subcomputation warranties.

We demonstrated on a simple microbenchmark that this can help improve throughput and reduce latency in applications where state warranties would not be extremely beneficial due to frequent writes. By supporting abstract predicates across multiple values, computation warranties help applications ensure they are enforcing consistency guarantees that actually matters to the application rather than low-level restrictions on reads and writes of individual values.

6.3 Predictive Treaties

Computation warranties are enforced by monitoring subexpression results for changes after updates to related state. When a computation warranty asserts an expression over state across multiple stores, synchronization may be required to compute an updated result. Predictive treaties build on the original computation war-

warranties design to provide system-scope warranties that avoid synchronizing during enforcement. The design uses novel techniques to ensure that recomputations of multistore assertions, which requires synchronization, are rare.

Predictive treaties use a predictive model and estimated model parameters on associated *metrics* that represent computations over the system state. Metrics predict the trends in the system state using a simple probabilistic model, Brownian motion with drift, whose parameters can be learned by tracking updates in the system. Unlike computation warranties, predictive treaties can express predicates which depend on *time* in addition to the system's state. Depending on time allows predictive treaties to track and assert *trends* on the system state discovered by metrics. These time-varying treaties can be used to enforce distributed treaties for much longer without requiring synchronization than previous static approaches for similar designs.

Predictive treaties have been demonstrated to improve performance over prior work on standard benchmarks, simple applications, and using data collected from a distributed web application. By reducing the occurrence of synchronization events during the lifetime of a warranty, predictive treaties provide lower operation latencies for distributed applications.

6.4 Future Work

I believe there are a number of directions for future work to better understand how to provide efficient, expressive, intuitive warranties that help improve distributed system performance. Here, I outline some potential opportunities.

6.4.1 Combining Leased and Public Warranties

Warranties in Chapters 3 and 4 demonstrated the value of *public* warranties, which do not require tracking who holds and uses warranties but require blocking invalidating updates. However, there are many cases where it may be better to use *leased* warranties which support revocation by tracking holders of the warranty, such as statements that are intended to be broken by users of the warranty (e.g., balance checks in banking) or rely on data for which updates are less predictable.

Ideally, the system could use workload data and a well-designed policy to automatically determine when a warranty should be public or leased to ensure good performance. Automatically switching between leased and public warranties help the system find the best tradeoff between benefits for warranty users and enforcement overheads.

Furthermore, there may be potential benefits of mixing leased and public warranties when composing warranties for enforcement. Perhaps there are cases where it might be useful to use leased warranties to enforce a system-scope public warranty: allowing more flexibility at each store while supporting greater sharing and lower tracking overhead at for system-scope guarantees.

6.4.2 Fault Tolerance and Failure Recovery

There are questions about how these abstractions can be designed to work well in the presence of failures such as network partitions, servers crashing, and byzantine behavior by some subset of the system nodes. In the designs presented, we did not consider design opportunities and overheads in making the system resilient to crashes of stores hosting and enforcing treaties. Using replication to tolerate failures of treaty hosts would help but is likely to create additional latency overheads. On the other hand, it may also be possible that warranties and treaties can be

used to *enhance* fault tolerance techniques by providing more general abstractions for capturing what is and isn't currently true about the system's status.

6.4.3 Techniques for Warranty Search and Discovery

Computation warranties and predictive treaties are designed to be broadly applicable by capturing high-level abstract guarantees about the system state. However, these techniques still rely on finding and using exact matches for the predicates used within an application.

For example, it's possible for the current design to simultaneously use and enforce two predicates ϕ and ψ without recognizing and leveraging an implication between the two such as $\phi \implies \psi$. A better design would probably enforce ψ using ϕ even if it would normally not be the best choice for avoiding synchronization—the system will be enforcing ϕ and so there's no need to create overheads for a separate enforcement strategy in the meantime.

6.4.4 Population Statistics for Metrics

Predictive treaties performance improvements are derived from having reasonably accurate predictions of future update behavior in an application. Unfortunately many real applications create new data for which there is no past behavior to construct predictions from, such as new user accounts or new items stocked in a warehouse. In cases where there is no past behavior to base predictions on, it is reasonable to assume that behavior for objects in a collection is predictive of the behavior of newly created or newly added objects.

Population statistics, predicting update behavior based on the trends seen across a group of related metrics, would allow predictive treaties to provide improved performance for cases where there aren't necessarily continuous trends.

Consider a boolean value which starts as false and eventually becomes true, never going back to false. This scenario will never exhibit enough updates to predict how much time will pass between creation and being set to true. With population statistics however, we can generalize observations from older boolean values which have already been created and then flipped to true. This scenario may seem contrived but it mirrors a fairly common pattern in distributed computing, found in scenarios like monitoring who has moved to a new version of your software, tracking who has cast their ballot in an election, or marking what seats have sold for a show.

6.4.5 Support for Non-Numeric Metric Data

So far the investigation in predictive treaties has been focused on numeric computations and data, but there are plenty of other kinds of data for which it might be useful to construct predictions for and create treaties over. The boolean example for population statistics is an example of a non-numeric metric data. More broadly, it would be useful to construct predictive treaties for statements on data structures, such as set membership or graph connectedness.

Some of these treaties are ostensibly supported by following the memoized function pattern used in Section 5.3 to build the voting application. However, it is not clear if there is a way to generalize the model-based techniques for choosing enforcement strategies with arbitrarily structured data. Extending metrics and predictive treaties to general data structures would make them much more widely applicable across distributed applications—much of distributed computing is not exclusively concerned with numbers.

6.4.6 Alternative Prediction Models

While linear trends are certainly good for capturing many behaviors, there are some known patterns that show up in distributed systems that are nonlinear and probably worth detecting and leveraging for avoiding synchronization. For instance, many services see patterns in data access that are explained by the sleep patterns of users around the world, with peak activity or uptime at some point each day [74]. One thought is to combine some more long-term predictions of such diurnal patterns with shorter term simple linear predictions to make better decisions about how to allocate and transfer slack over time. Perhaps the longer term prediction could help to make decisions about opportunistic slack negotiation (Section 5.5.2).

BIBLIOGRAPHY

- [1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *35th ACM Symp. on Principles of Programming Languages (POPL)*, pages 309–322, 2008.
- [2] Atul Adya. Transaction management for mobile objects using optimistic concurrency control. Master’s thesis, Massachusetts Institute of Technology, January 1994. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-626.
- [3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [4] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 23–34, San Jose, CA, May 1995.
- [5] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. In *16th ACM Symp. on Principles of Distributed Computing, PODC ’97*, pages 73–82, August 1997.
- [6] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *21st ACM Symp. on Operating System Principles (SOSP)*, pages 159–174, October 2007.
- [7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.
- [8] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *IEEE Symp. on Security and Privacy*, pages 191–205, May 2012.
- [9] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

- [10] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 28–39, New York, NY, USA, 2003. ACM.
- [11] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *PVLDB*, 8:185–196, 2014.
- [12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, April 2012.
- [13] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.
- [14] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*, September 2015.
- [15] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.
- [16] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM CSUR*, 13(2):185–221, 1981.
- [17] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.
- [18] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symp. Cloud Computing*, October 2011.
- [19] Chavdar Botev et al. Supporting workflow in a course management system. In *36th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 262–266, February 2005.

- [20] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.
- [21] Robert G. Brown. Exponential smoothing for predicting demand. *Operations Research*, 5(1):145–145, 1957.
- [22] Heiko Böck. *Java Persistence API*. Springer, 2011.
- [23] Michael Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *9th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 414–426, 1994.
- [24] Chee-Yee Chong and Srikanta P Kumar. Sensor networks: Evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.
- [25] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.
- [26] Graham Cormode. The continuous distributed monitoring model. *ACM SIGMOD Record*, 42(1):5–14, May 2013.
- [27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kukulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.
- [28] Rick Durrett. *Probability: Theory and Examples*. Cambridge University Press, 4th edition, 2010.
- [29] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer New York, 1996.
- [30] EclipseLink. <http://www.eclipse.org/eclipselink>.
- [31] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*,

- 19(11):624–633, November 1976. Also published as IBM RJ1487, December 1974.
- [32] Gavin King et al. Hibernate developer guide. Hibernate Community Documentation. <http://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html>.
- [33] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, August 2004.
- [34] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(2):186–213, June 1983.
- [35] Minos Garofalakis, Daniel Keren, and Vasilis Samoladas. Sketch-based geometric monitoring of distributed stream queries. *Proceedings of the VLDB Endowment*, 6(10):937–948, August 2013.
- [36] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 81–94, April 2018.
- [37] Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Izchak Sharfman, and Assaf Schuster. Prediction-based geometric monitoring over distributed data streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 265–276, New York, NY, USA, 2012. ACM.
- [38] David K. Gifford. Information storage in a decentralized computer system. Technical Report CSL-81-8, Palo Alto Research Centers, Xerox Corporation, June 1981. Revised March 1982.
- [39] Cary G. Gray. *Performance and Fault-Tolerance in a Cache for Distributed File Service*. PhD thesis, Stanford University, December 1990.
- [40] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symp. on Operating System Principles (SOSP)*, SOSP '89, pages 202–210, 1989.
- [41] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and de-

- degrees of consistency in a shared database. In *Modeling in Data Base Management Systems*. Amsterdam: Elsevier North-Holland, 1976. Also available in Chapter 3 of *Readings in Database Systems, Second Edition*, M. Stonebraker Editor, Morgan Kaufmann, 1994.
- [42] J. N. Gray. Notes on database operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.
- [43] Paul Greenfield, Alan Fekete, Julian Jang, Dean Kuo, and Surya Nepal. Isolation support for service-based applications: A position paper. In *3rd CIDR*, pages 314–323, 2007.
- [44] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 157–166, 1993.
- [45] T. Haerder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, June 1984.
- [46] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. Technical Report CMU-CS-88-120, Carnegie Mellon University, Pittsburgh, Pa., 1988.
- [47] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. pages 207–216, February 2008.
- [48] Hibernate. <http://www.hibernate.org>.
- [49] Charles C. Holt. Forecasting trends and seasonals by exponentially weighted averages. carnegie institute of technology. Technical report, Pittsburgh ONR memorandum, 1957.
- [50] J. Stuart Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18:203–210, 1986.
- [51] J. Jang, A. Fekete, and P. Greenfield. Delivering promises for web services applications. In *5th ICWS*, pages 599–606, July 2007.
- [52] <http://jmeter.apache.org>.

- [53] Daniel Keren, Izchak Sharfman, Assaf Schuster, and Avishay Livne. Shape sensitive geometric monitoring. *IEEE Transactions on Knowledge and Data Engineering*, 24(8):1520–1535, August 2012.
- [54] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. In *37th ACM Symp. on Principles of Programming Languages (POPL)*, pages 19–30, January 2010.
- [55] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2013.
- [56] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2):213–226, June 1981.
- [57] L. Lamport. Towards a theory of correctness for multi-user data base systems. Report CA-7610-0712, Mass. Computer Associates, Wakefield, MA, October 1976.
- [58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.
- [59] Butler W Lampson. Designing a global name service. In *5th ACM Symp. on Principles of Distributed Computing, PODC '86*, pages 1–10, August 1986.
- [60] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *SIGCOMM*, pages 454–467, 2016.
- [61] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2012.
- [62] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shriram. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 318–329, June 1996.
- [63] Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *10th ACM Symp. on Principles of Distributed Computing, PODC '91*, pages 1–9, August 1991.

- [64] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. on Computer Systems*, 9(2):125–142, May 1991.
- [65] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.
- [66] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *11th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 513–517, April 2014.
- [67] Ying Liu, Xiaxi Li, and Vladimir Vlassov. GlobLease: A globally consistent and elastic storage system using leases. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 701–709. IEEE, 2014.
- [68] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.
- [69] Tom Magrino, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C. Myers. Efficient, consistent distributed computation with predictive treaties. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, March 2019.
- [70] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, USA, 1987.
- [71] K. Marzullo. *Loosely-Coupled Distributed Services: A Distributed Time Service*. PhD thesis, Stanford University, Stanford, Ca., 1983.
- [72] Michael Menth and Frederik Hauser. On moving averages, histograms and time-dependent rates for online measurement. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE ’17*, pages 103–114, New York, NY, USA, 2017. ACM.

- [73] Donald Michie. “Memo” functions and machine learning. *Nature*, 218:19–22, 1968.
- [74] James W Mickens and Brian D Noble. Exploiting availability prediction in distributed systems. In *3rd USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 73–86, Berkeley, CA, USA, 2006. USENIX Association.
- [75] D. L. Mills. Network time protocol (version 3) specification, implementation and analysis. Network Working Report RFC 1305, March 1992.
- [76] P. O’Neil. The escrow transactional method. *ACM Trans. on Database Systems*, 11(4):405–430, December 1986.
- [77] Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st Workshop on Network and System Support for Games, NetGames ’02*, pages 79–84, New York, NY, USA, 2002. ACM.
- [78] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.
- [79] Peter Peinl and Andreas Reuter. Empirical comparison of database concurrency control schemes. In *9th Int’l Conf. on Very Large Data Bases (VLDB)*, pages 97–108, 1983.
- [80] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *17th ACM Symp. on Operating System Principles (SOSP)*, St. Malo, France, October 1997.
- [81] Rico Piantoni and Constantin Stancescu. Implementing the Swiss exchange trading system. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 309–313. IEEE, 1997.
- [82] 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 *9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2010.
- [83] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domin-

- gos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.
- [84] Calton Pu. Generalized transaction processing with epsilon-serializability. In *Proceedings of Fourth International Workshop on High Performance Transaction Systems*, 1991.
- [85] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.
- [86] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, MIT, Cambridge, MA, 1978.
- [87] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.
- [88] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM CSUR*, 37(1):42–81, March 2005.
- [89] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *3rd USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 18–31, Berkeley, CA, USA, 2006. USENIX Association.
- [90] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM*, 29(2):394–403, 1982.
- [91] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2019.
- [92] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

- [93] Marc Shapiro, Nuno M. Prego, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.
- [94] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Transactions on Database Systems*, 32(4), November 2007.
- [95] Sandeep K. Singhal and David R. Cheriton. Using a position history-based protocol for distributed object visualization. Technical Report CS-TR-94-1505, Stanford University, Department of Computer Science, Stanford, CA, USA, February 1994.
- [96] ObjectDB Software. ObjectDB 2.3 developer’s guide. <http://www.objectdb.com/java/jpa/persistence/lock>.
- [97] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.
- [98] Jeremy Sussman and Keith Marzullo. The bancomat problem: an example of resource allocation in a partitionable asynchronous system. *Theoretical Computer Science*, 291(1):103–131, 2003.
- [99] Simon J.E. Taylor, Jon Saville, and Rajeev Sudra. Developing interest management techniques in distributed interactive simulation using java. In *1999 Winter Simulation Conference Proceedings*, volume 1, pages 518–523. IEEE, December 1999.
- [100] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013.
- [101] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *24th ACM Symp. on Operating System Principles (SOSP)*, 2013.
- [102] 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 *15th ACM Symp. on Operating System Principles (SOSP)*, pages 172–183, December 1995.

- [103] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *16th ACM Symp. on Operating System Principles (SOSP)*, pages 224–237, 1997.
- [104] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.
- [105] TPC Council. TPC-C benchmark, revision 5.11, 2010.
- [106] Matt Tracy. How CockroachDB does distributed, atomic transactions, September 2015. <https://www.cockroachlabs.com/blog/how-cockroachdb-distributes-atomic-transactions/>.
- [107] Werner Vogels. Eventually consistent. *Comm. of the ACM*, 52(1):40–44, January 2009.
- [108] Michael Whittaker and Joseph M Hellerstein. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, 2018.
- [109] Yahoo! cloud serving benchmark. <https://github.com/brianfrankcooper/YCSB>.
- [110] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):563–576, 1999.
- [111] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2000.
- [112] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2012.