ABSTRACT

Title of dissertation: DECENTRALIZED DATA CONSISTENCY PROTOCOLS FOR MOBILE AND WIDE-AREA ENVIRONMENTS

Uğur Çetintemel, Doctor of Philosophy, 2001

Dissertation directed by: Assistant Professor Peter J. Keleher
Department of Computer Science

Replication is essential to systems that attempt to efficiently support shared data in mobile and wide-area environments. Existing approaches for managing replicated data are often ill suited for these environments because of implicit assumptions of high availability, strong connectivity, and static environmental and application-specific characteristics. This thesis presents replication protocols that address these concerns by integrating decentralized update commitment, peer-to-peer information propagation, and light-weight adaptation mechanisms.

The first part of the thesis covers Deno, an object-replication system specifically designed for use in mobile and weakly-connected environments. At the core of Deno lies a

decentralized update commitment protocol that is implemented through peer-to-peer, epidemic information flow. Unlike previous protocols, this combination allows Deno to support strong consistency without sacrificing availability.

The second part of the thesis covers decentralized, semantics-based data consistency protocols for wide-area environments. It first presents an experimental evaluation of data partitioning and replication approaches in the context of wide-area commodity distribution applications. It then presents ReBound, a middleware system that supports a general numerical divergence control framework for deploying distributed applications that can tolerate and benefit from bounded inconsistency. ReBound enables clients to specify continuous or one-time (i.e., per-read) quantitative precision constraints on the data they read. ReBound incorporates peer-to-peer, decentralized server-side protocols that efficiently ensure that these constraints are met by refreshing client caches as necessary. In both Deno and ReBound, light-weight adaptation is facilitated through the use of system-wide fixed weights and pair-wise weight redistribution.

The thesis also describes the architectures of the Deno and ReBound prototypes, and presents prototype- and simulation-based experimental results that demonstrate the performance advantages of our protocols over other decentralized approaches.

DECENTRALIZED DATA CONSISTENCY PROTOCOLS FOR MOBILE AND WIDE-AREA ENVIRONMENTS

by

Uğur Çetintemel

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2001

Advisory Committee:

Assistant Professor Peter J. Keleher, Chair/Advisor
Associate Professor Louiqa Raschid, University Representative
Assistant Professor Bobby Bhattacharjee
Assistant Professor Sudarshan S. Chawathe
Assistant Professor Chau-Wen Tseng

DEDICATION

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

Chapter 1 Introduction

Recent years have witnessed dramatic changes in the way people access data and services. This is mainly due to the emergence of Internet-based applications and technologies that enable mobile and ubiquitous computing. Data and computing services will continue to become increasingly decentralized in order to meet the performance, availability, and scalability requirements of the next generation environments and applications. This trend requires a critical rethinking of many issues that arise in the design and implementation of the underlying computing infrastructures and protocols. In this thesis, we focus on the issue of efficient, consistent access to distributed replicated data for distributed applications that run on top of mobile and wide-area environments.

Data replication is essential for performance, availability, and mobility. Data is replicated across sites in a distributed system mainly for enhancing fault tolerance (i.e., availability) and performance. Fault tolerance can be achieved by accessing available replicas. Performance can be enhanced by accessing *nearby* replicas and distributing work across sites. Obviously, these advantages do not come for free. In addition to the redundant storage of replicas (which has ceased to be a concern with the ever-decreasing storage costs), replicas at different sites need to be kept consistent for application correctness. This consistency requirement calls for data consistency protocols that commonly require proper propagation of updates to all replicas.

There has been considerable research devoted to data replication, and a wide variety of replication protocols have been proposed [34]. Existing replica control protocols, however, are ill suited for mobile, weakly-connected, and wide-area environments because of implicit assumptions of high availability, strong connectivity, and static envi-

ronmental and application-specific characteristics. Characteristics of our target environments —in particular, severe restrictions in connectivity and frequent disconnections in mobile and weakly-connected environments; network partitions and unpredictable delays in wide-area environments; and the dynamic nature of the systems and applications in both environments— render these assumptions unrealistic.

In this thesis, we present distributed data consistency protocols that eliminates these assumptions by integrating:

1. Decentralized, asynchronous update commitment,
2. Peer-to-peer information propagation, and
3. Light-weight adaptation mechanisms

Decentralized asynchronous update commitment, in our context, enables each entity (i.e., server) in the system to make commit decisions independently, asynchronously, using only local information, and without the need to have tight synchronization with multiple other servers. This feature eliminates the need for primary-copy [12, 65, 66] and ROWA-type approaches [4, 12], which are well known to suffer from low availability. The ability to work with *peer-to-peer information propagation* is especially desirable in weakly-connected, dynamic environments because of minimal connectivity and communication support demand from the underlying communication infrastructure. *Light-weight adaptation mechanisms* enable adaptation to changing environmental and application-specific factors in a light-weight fashion, i.e., without the need to have tight global synchronization among the servers in the system.

In the rest of this chapter we present an overview of the thesis and discuss our main contributions.

1.1 Decentralized Replication in Mobile and Weakly-Connected Environments

1.1.1 Overview

Replication is an essential mechanism for any system that attempts to efficiently support shared data in mobile and weakly-connected environments. The reason is that limitations in bandwidth and connectivity force a heavy reliance on local resources. Traditional synchronous solutions for managing replicated data are ill suited for these environments because they typically require strong connectivity and availability. In this respect, asynchronous solutions have inherent advantages over their synchronous counterparts, namely that they can operate with less than full connectivity, easily adapt to frequent changes in group membership, and make few demands on the underlying network topology. However, this functionality comes at a price: existing asynchronous solutions generally either require reconciliation and thus fail to provide formal correctness guarantees, or have low availability because they typically rely on primary-copy or ROWA-type protocols. *Deno*, a decentralized, highly-available shared-object infrastructure that we describe in this thesis, addresses these concerns.

Deno retains the advantages of previous asynchronous protocols through its *peer-to-peer* information and control flow mechanism, which divorces protocol requirements from communication requirements. Unlike previous protocols, however, Deno provides formal correctness guarantees, and achieves high-availability through a novel decentralized commitment protocol. This protocol enables each server to commit or abort transactions autonomously, based on locally available information. Furthermore, the system is truly decentralized in that no specific server owns any data item, and no

server is specifically charged with serializing updates to any given piece of data. Besides the high-availability enabled by decentralization, Deno supports the following features:

1. *Light-weight replica management and system reconfiguration operations*: Deno enables replicas to be created or retired, and their weights to be modified via the synchronization of only two servers, without requiring any global consensus or synchronization;
2. *A transactional replication framework that supports multiple consistency levels*: Deno supports updates that involve multiple items and multiple read/update operations. Deno provides two formal consistency levels: A weak consistency level that globally serializes individual updates only, and a strong consistency level that serializes reads as well as updates.
3. *Security against internal attacks*: Deno incorporates security extensions to provide protection against a specific class of internal threats, malicious actions by authenticated entities that misrepresent protocol-specific information. The proposed solution requires the use of conventional cryptographic primitives as well as modifications to the commitment protocol.

We characterized the performance of the basic Deno protocol and its extensions first by using a detailed simulation model and then using the Deno prototype. Overall, the performance results revealed that Deno indeed performs better than asynchronous ROWA-type protocols and comparable to asynchronous primary-copy protocols, while achieving higher availability than both types of protocols.

1.1.2 Contributions

In terms of data replication in general and epidemic protocols in particular, the thesis makes numerous contributions, which we outline as follows:

- We propose a new replication protocol that combines weighted voting and asynchronous epidemic information flow. The protocol is completely decentralized: there is no primary server that *owns* an item or serializes the updates to that item. Any server can create and commit new updates, and servers need only be able to communicate with a minimum of one other server at a time in order to make progress.
- We propose *bounded* voting, a new voting scheme that, unlike traditional voting schemes, bounds the total weight aggregated across *all* replicas to a system-wide, static value. This unique feature enables several desirable operations, such as replica creation, retirement, and weight redistribution, to take place in a light-weight manner in the system; i.e., these operations can be performed while maintaining the correctness of the protocol with the participation of at most two servers. These operations are especially useful in mobile and weakly-connected environments due to the need to quickly adapt to changing environmental and application-specific factors and efficiently modify system configuration.
- We present the design and evaluation of the Deno decentralized object-replication system that combines bounded voting with pair-wise epidemic information flow. Deno is the first decentralized system proposed for supporting replicated objects in mobile and weakly-connected environments that provides formal consistency

guarantees (unlike optimistic systems) without sacrificing availability (unlike systems that are based on primary-copy or ROWA approaches).

- We present a detailed performance study of the epidemic replication protocols that appeared in the literature using both simulation and Deno implementation. Our study provides fundamental insight into the tradeoffs between centralized and decentralized epidemic protocols and different aspects of epidemic protocols. One particularly interesting result of our investigation is that the presumed performance advantage of the centralized primary-copy approach over a decentralized voting approach is not significant with asynchronous epidemic protocols.
- We propose and evaluate extensions (i.e., *speculative update propagation*, and *directed synchronization*) to Deno's basic replication protocol, and qunatitatively demonstrate that these extensions significantly improve the performance of Deno under many workload scenarios and system configurations.
- We described the first treatment and evaluation of secure update commitment protocols for peer-to-peer, decentralized databases in the context of Deno. We investigate internal threats, primarily focusing on malicious attacks that involve misrepresentation of protocol-specific information. To handle such malicious insiders, we extend Deno's non-secure replication protocol with cryptographic techniques and modifications to the update commit criteria. A unique aspect of our solution is that it not only enables a tradeoff between performance and the degree of tolerance to malicious servers, but also allows for individual servers to support non-uniform degrees of tolerance without adversely affecting the performance of the rest of the system. We then characterize the performance of our internal security extensions

using the Deno prototype. We also describe protocols and mechanisms that provide protection against external security threats using conventional cryptography-based mechanisms.

1.2 Decentralized, Semantics-based Data Consistency Protocols in Wide-Area Environments

1.2.1 Overview

In the second part of the thesis, we address wide-area applications and services and present decentralized data consistency protocols that exploit relaxed consistency requirements to improve overall system performance.

We first present an experimental study that investigates the inherent tradeoffs between data partitioning- and replication-based approaches for wide-area inventory-based commodity sales applications (i.e., applications that allow globally distributed purchasing of commodities and merchandise such as books, CDs, travel tickets, etc., over the Internet). Semantics of these applications commonly allow commodities of interest to be represented on line by tokens, which can then be distributed among servers to enhance the performance and availability. There are two fundamental approaches for distributing such tokens — partitioning and replication. Partitioning-based approaches eliminate the need for tight quorum synchronization required by replication-based approaches. The effectiveness of partitioning, however, relies on token redistribution techniques that allow dynamic migration of tokens to where they are needed. We propose pair-wise token redistribution strategies to support applications that involve wide-area commodity distribution. Using a detailed simulation model and

real Internet message traces, we investigate the performance of our redistribution strategies and a previously proposed replication-based scheme. Our results reveal that, for the types of applications and environment we address, partitioning-based approaches perform superior primarily due to their ability to provide higher server autonomy.

We observe that for many distributed applications running in wide-area, it is neither necessary (due to their relaxed correctness semantics) nor practical (due to limitations in communications and large-scale) to maintain strict consistency of replicated data at all times. In fact, a common theme of many popular distributed applications, including the distributed commodity sales applications we study, is that they maintain numerical shared data, and that they can continue to operate with stale, inconsistent data *as long as the divergence from the accurate, up-to-date data is properly bounded*. Such applications include mobile sales and inventory, wide-area resource monitoring, allocation, and management applications, wide-area network management, and e-commerce commodity sales applications.

Based on these observations, we then extend and generalize the data partitioning- and replication-based solutions that we studied in order to provide efficient support for deploying such applications in wide-area environments. *ReBound*, a middleware system that provides a general numerical divergence control framework, captures the consistency requirements of many popular distributed applications such as those mentioned above. In the ReBound framework, servers update distributed, replicated numerical data, which are cached by clients. Clients submit read operations that specify numerical divergence bounds on the data items they cache. These bounds indicate

limits on the acceptable quantitative deviation of the data read by clients from the accurate, up-to-date values maintained at the servers. By varying the divergence bounds they specify, clients can trade off the degree of the consistency of the data they read with the efficiency of reads. We propose peer-to-peer, decentralized server-side algorithms, which are based on partitioning and replication of the numerical bounds across the servers in the system, that efficiently ensure that these bounds are met by refreshing client caches with new updates as necessary.

The ReBound framework is general in that it captures the consistency requirements of several mobile and wide-area applications by enabling an arbitrary set of clients to specify reads with (1) continuous/one-time, (2) soft/hard quantitative consistency constraints, which are then enforced cooperatively by an arbitrary set of servers. The framework is adaptive in that it employs light-weight mechanisms to dynamically readjust the constraints to be enforced locally at individual servers, based on application-specific and environmental factors. We also describe the design of the ReBound prototype and present preliminary experimental results that demonstrate the practicality of the model and significant performance benefits that ReBound can provide for applications that can tolerate bounded numerical divergence.

1.2.2 Contributions

Our major contributions in this part of the thesis can be summarized as follows:

- We study data partitioning in wide-area environments by proposing decentralized, pair-wise token redistribution strategies that dynamically migrate tokens to where they are needed in the network, and by evaluating their performance under a range

of workload scenarios using a detail simulation model and real wide-area message traces.

- We provide valuable insight into the fundamental tradeoffs between data partitioning- and replication-based approaches for an increasingly important class of distributed applications that involve wide-area inventory-based commodity sales and distribution. In particular, we conduct the first performance study that directly compares these two fundamental approaches using a detailed simulation model and real Internet message traces. Our results reveal that, for the kinds of applications and environments we address, partitioning-based approaches perform superior mainly because they enable higher site autonomy when executing transactions.
- We propose a numerical divergence control framework, ReBound, that enables a data quality (i.e. precision) vs. performance tradeoff when accessing replicated data that are updated at multiple peer servers. The framework is general in that it captures the consistency requirements of several distributed applications by enabling an arbitrary set of clients to specify read operations with continuous/one-time, and soft/hard quantitative divergence constraints, which are then enforced cooperatively by an arbitrary set of servers. The framework is adaptive in that it employs light-weight mechanisms to dynamically re-adjust the divergence constraints to be enforced locally at individual servers, based on application-specific and environmental factors. We also argue that several previous existing proposals that address the numerical divergence problem for replicated data are instances of our framework.

- We propose two server-side, decentralized divergence control algorithms that efficiently maintain divergence constraints by partitioning or replicating them across the servers in the system, and by limiting the total weight of the committed updates that are unknown to clients by refreshing client caches as necessary.
- We present preliminary experimental results, based on our ReBound prototype, that demonstrate the practicality, adaptability, and the performance advantages of our divergence control algorithms under various workloads and system configurations.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows:

Chapter 2 presents a survey of the previous literature in the related general areas of data replication and consistency protocols in general, and replication in mobile, weakly-connected, and wide-area environments in particular.

Chapter 3 presents the Deno decentralized object-replication system, which is the primary focus of this thesis, in detail. Chapter 4 addresses security issues for decentralized peer-to-peer databases in the context of Deno.

Chapter 5 discusses our work on token redistribution for a class of distributed e-commerce applications that involve wide-area commodity sales and distribution. Chapter 6 presents the ReBound system that is designed to provide effective divergence control for numerical replicated data that can updated at multiple wide-area locations.

Finally, Chapter 7 concludes the thesis by presenting a summary of the results of our work and outlining avenues for future research.

Chapter 2 Literature Survey

Data replication is crucial for performance, availability, and mobility. In this section, I present a brief survey of data replication, focusing on the solutions proposed for mobile and wide-area environments.

2.1 Data Replication and Consistency in Distributed Systems

There has been considerable research devoted to data replication and a wide variety of solutions have been proposed [34]. Data is replicated across sites in a distributed system mainly for enhancing fault-tolerance (i.e., availability/reliability) and performance. Fault tolerance can be achieved by accessing available replicas. Performance can be enhanced by accessing *nearby* replicas and distributing work across sites. Obviously, these advantages do not come for free. In addition to the redundant storage of replicas, which has ceased to be a concern with ever-decreasing storage costs, replicas at different sites need to be kept consistent for application correctness. This consistency requirement calls for data consistency protocols that typically require the same update to be propagated to and applied on all replicas.

Many applications require the concept of a *unit of execution* containing multiple operations on (possibly) several items. A *transaction* refers to the execution of a sequence of operations with the *ACID* properties:

- *Atomicity*: Either all or none of the operations of a transaction are executed.
- *Consistency*: A transaction takes a consistent database state to another consistent state.

- *Isolation*: A transaction is executed in isolation from other transactions.
- *Durability*: When a transaction commits, its effects are permanently reflected to the database.

Application programmer is responsible for consistency and the system guarantees atomicity, isolation, and durability. A *serial execution* of transactions is one in which all operations of a transaction is executed one after another, without being interleaved by operations belonging to other transactions. Consistency, therefore, ensures that an execution is correct if it is serial. An execution is *serializable* if it is equivalent to a serial execution in terms of the resulting database state and the output it produces. Concurrency control protocols deal with restricting operations from different transactions to achieve serializable executions. These protocols can roughly be categorized into two [12]: (1) pessimistic protocols (e.g., two-phase locking (2PL)) that disallow conflicting operations typically by blocking, thereby eliminating the need to undo the effects of operations later; and (2) optimistic protocols that detect conflicts after they occur and abort a subset of involved transactions before their effects are finalized.

The notion of serializability is extended to specify correctness for replicated databases as *one-copy serializability* (1SR) [12]. Replica control protocols are typically used to ensure 1SR and present the most current state of the database under failures. Atomicity control protocols guarantee the atomicity property of transactions by having all involved sites either commit or abort a transaction. The Two-Phase commit protocol (2PC) is one of the most popular atomic commit protocols [12]. 2PC commits a transaction only after *all* the involved sites declare that they are ready to commit. The

three-phase commit protocol (3PC) improves upon 2PC by avoiding blocking under a wider range of failure modes [12].

Read-One-Write-All (ROWA) protocols [12, 34] are one of the simplest and perhaps the only transactional replication schemes employed by commercial database systems. In ROWA protocols, an update operation is applied to all the replicas. Reads, on the other hand, can be performed on any single replica. One popular ROWA variant is the *primary-copy* (aka., master copy or monarchy) protocol [65] where a specific replica is designated as the primary copy of the data item. Reads are performed at the primary copy, whereas writes are first performed at the primary copy and then propagated to all other sites. The fundamental shortcoming with ROWA-based approaches is that the unavailability of even a single site may stall update operations. Read-One-Write-All-Available protocol [12, 34] was proposed to overcome this limitation of ROWA.

Quorum Consensus (aka., voting) schemes generalize ROWA schemes by trading off read and write availability [12, 34]. The fundamental idea underlying quorum systems is to synchronize a quorum of servers to perform an operation prior to performing it. In other words, servers are asked to vote for or against performing a certain operation (hence the term voting). Quorums are formed such that conflicting operations require overlapping quorums (i.e. quorum intersection property), ensuring that no two conflicting operations can be executed concurrently. For instance, in the uniform majority quorum consensus protocol [67], a write operation can be executed only after a majority of servers vote for performing that operation. It is thus easy to see that no two conflicting writes can be executed simultaneously since at most one write operation can gather votes from the majority of servers. The primary benefit of quorum-based

systems is that they enable higher availability and fault-tolerance than ROWA approaches.

There has been significant research in quorum-based systems (e.g., [2, 27, 67]). These proposals typically differ in how they specify quorum membership. Uniform majority voting [67] technique is one of the earliest quorum schemes. In this scheme, each server is assigned a unit vote and a majority of votes is required before a read or write is operation is performed. This approach is resilient to a single network-partition and independent failures of at most half of the sites. However, multiple partitions can not be tolerated. A drawback of this approach is that multiple sites need be contacted even for a read operation.

Weighted majority voting generalizes uniform majority voting by assigning a non-negative weight to each replica [27]. Assume that the sum of the weights of replicas equals u. Every object is assigned a read threshold r and a write threshold w such that (1) $r + w > u$ and (2) $w > u/2$. Rule (1) guarantees that any successful read operation returns the current value of the object and rule (2) guarantees that no two conflicting updates can be performed simultaneously.

Further work extended these basic quorum schemes by imposing *logical* structures on the set of sites replicating the object. For instance, the Grid protocol [20] logically arranges the set of sites into a grid structure. By registering, for instance, reads at a set of sites that lie on a single column and the writes at a set of sites that lie on a single row, quorum intersection property is guaranteed for read-write conflicts. Other organizations such as multi-dimensional weighted-voting [5] and tree quorum schemes [2] were also explored.

The cost of assembling a quorum is highly dependent on the distribution of weights across replicas in the system. Weight can be distributed and redistributed to optimize overall performance, availability, or a combined metric. The *optimal* weight distribution depends on the non-trivial interplay among several factors such as expected availability of individual servers, interconnectivity, and application characteristics. In general, replicas that are more reliable or better interconnected should receive more weight [10]. Amir and Wool studied optimal weight distributions that based on individual failure probabilities at sites [8]. Barbara *et al.* [11] proposed a dynamic weighted-voting scheme in which the weights at sites are changed dynamically. In order to increase availability, weights are continuously reassigned such that more available sites hold more weight.

We note that recent work [70] has investigated why quorum systems have yet to become widespread in real-world applications. One of the conclusions is that quorums do not enhance availability because either failures are positively correlated (when servers are on a single LAN) or network partitions occur (when servers are distributed across multiple LANs). In the latter case, a quorum constructed on a single LAN has higher availability than quorums constructed across multiple LANs. However, mobile environments fit neither category. Most failures (e.g., disconnections) are likely to be independent, and partitions, while possible, are not the dominant cause of unavailability.

Gray [29] categorized replication protocols in two dimensions: update regulation and update propagation. Two types of update regulation are possible: Group and master regulation. In group update regulation, an object can be updated at any server. In

master regulation, on the other hand, every object has a master node and an object can only be updated at its master. All other (non-primary) replicas are read-only. There are two ways to propagate updates: eager propagation and lazy propagation. In eager update propagation, updates are propagated to all replicas of the same object synchronously as part of the same transaction. In lazy propagation, updates are propagated asynchronously, typically as separate transactions.

2.2 Replication in Mobile and Weakly-Connected Environments

Traditional replication protocols are typically designed based on the (implicit) assumptions of high availability and strong/complete connectivity. In mobile and weakly-connected environments, restrictions in connectivity and frequent disconnections from the communication network render these assumptions somewhat unrealistic. It is, therefore, impractical to support strong consistency and 1SR in these environments [29]. Fortunately, weaker notions of consistency suffice for many real-world domains and applications (e.g., shared file systems, distributed collaborative applications). Many replicated systems, thus, employ weak-consistency replication protocols, trading off replicated data consistency with performance and availability.

The basic approach to weak consistency is to execute and possibly commit updates locally, and then either *optimistically detect and resolve* or *pessimistically prevent* update conflicts that may arise because of concurrent execution of conflicting updates at different parts of the system.

2.2.1 Optimistic Approaches

An example of the conflict detection and resolution is the protocol used by Coda system [43], which is based on a client-server model. Coda clients are allowed to update

the data in their caches during a network partition or disconnected operation. It is possible that a client update a data item while it is disconnected and then during reintegration find out that the same data item is already updated at the server (using per-data-item timestamps that count the number of updates committed on an item). Having detected a conflict, Coda executes a conflict-resolution procedure to resolve the conflict and somehow *merge* the conflicting updates so that the effects of all the updates are reflected to the state of the involved data item. If automatic conflict resolution is not possible, the conflicting updates and the involved data are marked for *manual* resolution. Ficus [56], a peer-to-peer distributed filing system intended for wide-area, Internet-based use, and Lotus Notes [40], a commercial epidemic system that has widespread usage in practice, also employ similar optimistic update policies along with mechanisms for detecting and resolving conflicts automatically.

2.2.2 Pessimistic Approaches

The main problem with the optimistic approaches is that automatic conflict reconciliation is only viable in certain restricted domains such as file systems, and manual reconciliation, as Gray argues [29], is not scalable and leads to *system delusion* where databases at different servers/clients diverge from each other such that there is no straightforward way to make them consistent anymore. Optimistic approaches therefore inherently cannot provide any formal consistency guarantees (e.g., 1SR), restricting their applicability to only a certain of applications.

Pessimistic approaches address these problems by *preventing* the commitment of update conflicts, at the expense of performance and availability, and by making formal consistency guarantees (typically using some notion of serializability [12]). Bayou

[66], a weakly-consistent storage system that is designed for mobile and weakly-connected environments, uses such a pessimistic approach. Bayou allows all replicas to be read and updated, and propagates the updates using epidemic information flow [23]. Bayou guarantees a unique global serialization order on all updates determined by the primary-copy; i.e., all updates are serialized in the order they are received and executed the primary-copy server. Since updates are typically observed at different orders at different Bayou servers, servers employ mechanisms that undo the effects of previously executed updates and redo them according to the global serialization order designated by the primary copy.

An alternative to the epidemic primary-copy approach used by Bayou is an epidemic ROWA approach such as the one proposed by Agrawal *et al*. [4]. Agrawal's algorithm supports transactional updates, and strongly-consistent executions, ensuring 1SR. The basic idea is to have an update transaction to be pre-committed at *all* the servers before it is allowed to commit globally.

It has long been known that primary-copy and ROWA approaches inherently suffer from low availability because the unavailability or inaccessibility of even a single server (i.e., the primary server in primary-copy approach and any server in the ROWA approach) is sufficient to halt update commitment in the system. Voting protocols based on epidemic information flow were proposed to achieve improved availability over epidemic primary-copy and ROWA approaches. Availability is improved by requiring only a subset of the servers to be available for update commitment. The Deno protocol we present in this thesis is the first epidemic voting approach that appeared in the literature. Based on the early Deno protocol, Holliday *et al.* [35] also proposed an

epidemic voting approach for transactional updates. Holliday's work assumes a more traditional distributed database environment and a traditional majority-voting scheme applied implemented with epidemic information flow. The main difference of our Deno work from Holliday's work, among many others, is that our emphasis is on making progress under incomplete system information in dynamic environments.

2.3 Semantics-based Distributed Data Consistency Protocols

Traditional replicated data management protocols providing strong consistency of replicated data are typically prohibitive in wide-area environments: these protocols commonly require multi-server synchronization for each commit, thereby incurring a significant communication overhead as synchronization of multiple servers in wide-area is costly due to the error-prone, highly unpredictable nature of the underlying communications medium.

Fortunately, semantics of many popular distributed applications, including inventory-based commodity sales and distribution, network management, and replicated stock quotes services, do not require strong consistency of replicated data. System performance and availability can be significantly improved if the relaxed correctness requirements of such applications are effectively exploited.

2.3.1 Escrow-based Distributed Data Partitioning and Replication

There has been significant work that focused on exploiting application semantics to improve performance in certain classes of applications. O'Neil [55] proposed *escrow transactions* to enable concurrent access to high traffic *aggregate fields* on which only a restricted class of operations (such as incremental updates) are allowed. This technique utilizes the commutativity property of such operations; i.e., two operations can

run in any order and still produce the same final result provided that they do not violate upper/lower bound constraints on the involved data items. Escrow transactions execute a special escrow operation that attempts to *put in escrow* (i.e., reserve) some of the resources that it plans to acquire. All escrow operations that succeed are logged. Transactions consult this log before executing an escrow operation and see the total amount of resources that are escrowed by all uncommitted transactions. If the total quantity of un-escrowed resources is sufficient, the transaction proceeds; otherwise it aborts.

Haerder [33] extended the escrow transactional model for centralized database environments to DB-sharing systems in which multiple DBMSs share the database at the disk level. He proposed the use of a hierarchical escrow scheme that consists of a global escrow and local distributed escrows. He discussed a technique that enables the hierarchical scheme to behave like a purely global scheme for only a critical margin of aggregate values.

Kumar and Stonebraker generalized the notion of escrow transactions for replicated data [48]. Each server is assigned an escrow quantity that can be used to execute transactions locally. The escrow quantities at servers are readjusted by the use of a periodic global snapshot algorithm. This algorithm has to be executed sufficiently frequently for sites to have an up-to-date view of the global state. On the other hand, frequent execution of such a costly algorithm may itself degrade performance. Both DVP and GSE employ mechanisms that eliminate the need for a global state algorithm.

Kumar discussed several *borrowing policies* for escrow transactions in a replicated environment [46]. He devised four simple borrowing policies that (1) select lender

sites either randomly or according to a pre-specified order, and (2) borrow either the exact amount they need or borrow an amount such that the final escrow quantities at the involved sites become equal. Kumar's borrowing policies do not make use of the knowledge of the global state of the system to improve its effectiveness and adapt dynamically to workload.

Golubchik and Thomasian discussed demand-driven token allocation schemes in the context of a fractional data allocation method [28]. One such scheme they describe enables token partitioning between the involved sites based on demand. In [68], Thomasian further discussed FDA and proposed an abstract model for optimal initial token distribution. It is worth noting that no previous work has explored the fundamental tension between replication and partitioning for token-based resource distribution.

2.3.2 Distributed Constraint Maintenance for Numerical Replicated Data

There has been significant research on maintaining consistency constraints on numerical replicated data. Recently, Yu and Vahdat [73] proposed algorithms for numerical error bounding for replicated network services. These algorithms are used to bound numerical error in the continuous consistency model [72]. The algorithms proposed, like the ones we present in this thesis, use the sum of the weights of the updates committed at individual servers but not seen at other servers. The error bounding model proposed in [73] can be seen as a specific instance of the ReBound framework (see Chapter 6) where the system configuration is restricted to the multiple server/single client, and only continuous and hard divergence bounds are supported. Furthermore, the proposed error bounding algorithms do not consider bound redistribution and are not adaptive.

Olston and Widom [54] recently proposed tunable algorithms that provide precision vs. performance tradeoff for aggregation queries over replicated data. Each query comes with a quantitative precision constraint, and is satisfied by a combination of cached data and accurate master data maintained at a single server. The model they describe is an instance of our framework where the configuration is limited to the single-server/multiple clients, and only soft constraints are supported. Olston and Widom do not address how to support data updated at multiple servers, which is our primary focus in this work in this thesis.

Gupta and Widom [32] discuss an algorithm that enables local verification of global integrity constraints. The algorithm exploits the idea of *covering tuples* such that the insertion of a new tuple cannot invalidate a constraint if there already exists a tuple *covering* the new one with respect to the constraint. It is not possible to find any such *covering* data for numerical data.

The *demarcation protocol* [9] was proposed for maintaining linear arithmetic inequalities in traditional distributed database systems. This work is closely related because a divergence bound on an item is essentially an arithmetic constraint. The demarcation protocol is not designed to support fine-grained, one-time reads with divergence bounds. Furthermore, the demarcation protocol does not utilize many unique properties of specific to numerical divergence control case: First, it does not exploit server views that store the approximate states of other nodes in the system. This view mechanism enables servers to set local divergence bounds based on the updates committed by other servers in the system. Second, after all clients are refreshed, the bounds at all servers can be reset. Even though the demarcation protocol also exploits

local constraint readjustments, it only enables explicit server-server adjustments. As we discuss in Chapter 6, Rebound allows explicit bound redistribution via practical server-server weight redistribution, as well as implicit readjustments that automatically happen during a view advance.

Epsilon-serializability [71] is a generalization of conventional serializability [12], where a limited amount of inconsistency is allowed by multiple reads in a query. Later work [36] extended epsilon-serializability to distributed databases. Similar to the demarcation protocol, epsilon-serializability addresses a much more general problem than the numerical divergence control problem we address here. It therefore cannot exploit many features specific to our problem. The protocols proposed for epsilon-serializability, being designed for general transaction processing, heavily rely on locking-based techniques, which are prohibitive for the types of environments and applications we address in this thesis.

Bounded ignorance [44] also allows bounded violations of integrity constraints by allowing a transaction to be ignorant of the results of at most a limited number of transactions. Bounded ignorance, however, provides no bounds on the data item values themselves. *Divergence caching* [37], *quasi-copy caching* [6], and several materialized view maintenance algorithms (e.g.,[31, 38, 74]) all address numerical error bounding. One limitation of these proposals is that they assume that only the master database accepts updates to data items. In this respect, the model we investigate in this thesis is more general because we efficiently support the case where data items are updated at multiple servers.

Chapter 3 Deno: A Decentralized Object-Replication System

Recent advances in hardware and software technologies have made mobile computing feasible and practical. Mobile device usage is increasing, as the devices become smaller, cheaper, and more powerful. Mobile users often carry their laptops, PDAs, and other portable devices wherever they go.

Mobile environments differ from typical desktop environments in many ways, including power availability, resources such as CPU, memory, secondary storage, and, above all, in their communication behavior. Mobile systems usually lack continuous connectivity, and typically possess limited communication bandwidth even when they are connected. As a result, mobile and weakly connected operations rely heavily on replication mechanisms in order to deliver good performance.

Replicas are useful for many reasons, including efficiency, availability, and fault tolerance. Replicas increase efficiency by allowing a local rather than a remote copy to be accessed, much in the same way that accessing a processor's memory cache is much faster than accessing memory over the computer's I/O bus. Replicas improve availability by making it possible for applications to make progress even when one or more replicas become temporarily unavailable. Fault tolerance is achieved by ensuring that object data is kept consistent. Loss of any one replica does not result in committed updates being lost if other replicas have copies of the same updates.

The problem with having replicas is that they must be kept consistent. Consistency is problematic in distributed systems because updates of multiple sites are generally non-atomic operations. Different sites usually take differing amounts of time to access,

meaning that competing tentative updates may be seen in different orders at different sites. However, consistency requires that any competing updates to the same shared object be committed in the same serial order at every replica [12].

There has been considerable research devoted to data replication and a wide variety of solutions have been proposed [34]. Traditional replication mechanisms, however, are ill suited for mobile environments and weakly-connected environments [29]. Mobility and weak connectivity require a critical reassessment of the assumptions underlying traditional replication mechanisms [7]. For instance, one assumption made by master-copy replication schemes [65] is that a single server is always available and accessible by the rest of the system. Clearly, such an assumption may become invalid in mobile environments. Server machines may be disconnected, and therefore inaccessible, at any given time. Furthermore, master-copy replication often assumes that the master server has complete and up-to-date knowledge of system membership, which is difficult to obtain in a mobile environment. As another case in point, consider replication in traditional voting schemes. Such schemes typically work by requiring a quorum of simultaneously connected servers to agree on an operation prior to performing it. If such a quorum cannot be established, the operation is aborted. However, mobile replication protocols should ideally allow progress to be made, and updates to be committed, even if a quorum of servers is not simultaneously available. Mobile replication protocols should therefore be decentralized and asynchronous wherever possible.

Asynchronous solutions (e.g., [14, 40, 49, 66]) for managing replicated data have a number of advantages over traditional synchronous replication protocols in large-scale, mobile, and weakly-connected environments. They can operate with less than

full connectivity, easily adapt to frequent changes in group membership, and make few demands on the underlying network topology. However, this functionality comes at a price. Asynchronous solutions are generally either optimistic and require reconciliations [43], or have lower availability because they rely on primary-copy schemes [65].

In this chapter, we present the design, implementation, and performance evaluation of Deno, a highly-available replicated object storage system intended for use in mobile and weakly-connected environments. Deno achieves high availability while providing formal correctness guarantees through its decentralized, asynchronous replication protocol. Deno differs from previous approaches in that it completely decentralizes all control and information flow. Innovations of Deno include the extension of voting schemes [27, 67] through fixed per-object weights and the use of pair-wise epidemic protocols [23] with voting schemes. Deno's replication protocol is highly available, and is able to make progress and eventually commit updates even if there is never a majority of replicas connected to each other simultaneously. No server ever needs to have complete knowledge of group membership, and a given server only needs to be in intermittent contact with at least one other server to take full part in the voting and commitment process. As such, the protocol is highly suited for environments with weak connectivity.

The rest of the chapter is organized as follows. Section 3.1 describes our system model. Section 3.2 provides a high-level description of Deno's decentralized replication scheme. Section 3.3 discusses Deno's basic replication framework in detail. Section 3.4 extends the base Deno protocol to provide stronger consistency semantics.

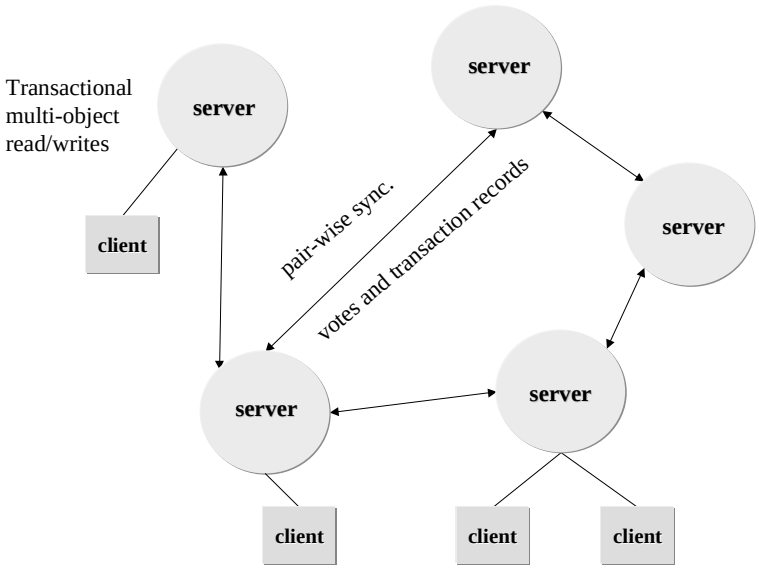

Figure 1: Deno system model

Section 3.5 argues the correctness of the proposed protocols by providing correctness proof outlines. Section 3.6 describes the weight management mechanisms and policies used in Deno, and Section 3.7 presents several extensions we proposed to improve the availability and performance of Deno. Section 3.8 presents the design of our Deno prototype system and Section 3.9 presents our experimental evaluation and results. Finally, Section 3.10 concludes the chapter by summarizing the main results of our work.

3.1 System Model and Features

Deno's system model is illustrated in Figure 1. One or more client processes link to each Deno server, which communicates through pair-wise information propagation. The Deno servers are not necessarily *ever* fully connected.

The protocol's strengths result from a combination of weighted voting and epidemic information flow [23], a process where information flows pair-wise through the sys-

tem like a disease passing from one host to the next. The protocol is completely decentralized. There is no primary server that owns an item or serializes the updates to that item (as in Bayou [66]). Any server can create new object replicas, and servers need only be able to communicate with a minimum of one other server at a time in order to make progress. Instead of synchronously assembling quorums, which has been extensively addressed by previous work (e.g., [27, 39, 67]), votes are cast and disseminated among system servers asynchronously through pair-wise propagation. Any server can commit or abort any transaction unilaterally, and all servers eventually reach the same decisions.

The use of voting allows the system to have higher availability than primary-copy protocols. The use of weighted voting allows implementations to improve performance by adapting weight distributions to site availabilities, update activity, or other relevant characteristics [17]. Each server has a specific amount of weight, and the total weight in the system is fixed at a known value. The advantage of a static total is that servers can determine when a plurality or majority of the votes have been accumulated *without complete knowledge of group membership*. This last attribute is key in dynamic, wide-area environments because it allows the protocol to operate in a completely decentralized fashion, eliminating performance bottlenecks and single points of failure.

The use of epidemic protocols divorces protocol requirements from communication requirements. First, an epidemic algorithm only requires protocol information to move throughout the system eventually. The lack of hard deadlines and connectivity requirements is ideally suited to mobile environments, where individual nodes are rou-

tinely disconnected. Second, epidemic protocols remove reliance on network topology. Synchronization partners in epidemic protocols can be chosen randomly, eliminating the single point of failures that occur with more structured communication patterns such as spanning trees.

3.2 Decentralized Data Consistency Protocols

3.2.1 Overview

For exposition purposes, we first briefly give a high-level description of Deno's asynchronous weighted-voting protocol. We formalize our discussion and give a detailed description of the protocol in Section 3.3.

We assume a model in which the shared state consists of a set of objects replicated across multiple servers. In general, objects do not need to be replicated at all servers and multiple objects can be replicated at the same server. Objects are modified by updates, which are issued by servers. Updates do not commit globally in one atomic phase. Instead, each server independently commits updates on the basis of local information. However, we show that if an update commits at one server, it eventually commits at all servers, and in the same order with respect to other committed updates.

A clean way of thinking about update commitment is as a series of *elections*. In the election framework, a server is analogous to a voter, creating an update is analogous to a voter deciding to run for office, and a committed update is analogous to a candidate winning the election. A candidate wins an election if and when it corners a plurality of the votes. Votes are weighted and the sum of all votes in the system is bounded to 1.0. Any election may have multiple candidates, which represent logically concurrent, conflicting updates. As we describe in the following sections, we use version numbers and

logical timestamps to detect logically concurrent updates. Candidates from different elections might be alive at the same time due to the asynchronous nature of the system.

Voting information and updates flow from voter to voter through anti-entropy (i.e., synchronization) sessions. For our purposes, an anti-entropy session from server s_i to server s_j is a uni-directional flow of information that specifies the elections that have been won and the votes in the current election. More specifically, an anti-entropy from v_i to v_j involves propagation of the unknown votes and updates from v_i to v_j. Voter v_i applies the propagated committed updates to its local database, and votes for the new tentative updates if it has not already voted for conflicting updates.

A voter v keeps track of the votes of all individual votes and summarizes this election information in two main statistics:

- $|votes(\{k\})|$, which is the sum of votes that have been cast in favor of candidate k in v's current election, and
- *unknown*, which is the sum of weights of voters whose vote for v's current election is currently unknown to v.

Voter v gathers election information until either it can award its current election to a candidate k, or learns from another server that the election has already been committed. Voter v awards the election to k when v finds out that k has won a plurality of votes, that is, if and only if, for all candidates $k \neq j$:

$$|votes(\{k\})| > |votes(\{j\})| + unknown$$

The above condition implies that candidate update k can commit only when it is guaranteed that no other candidate update j can gather more votes than k. The proto-

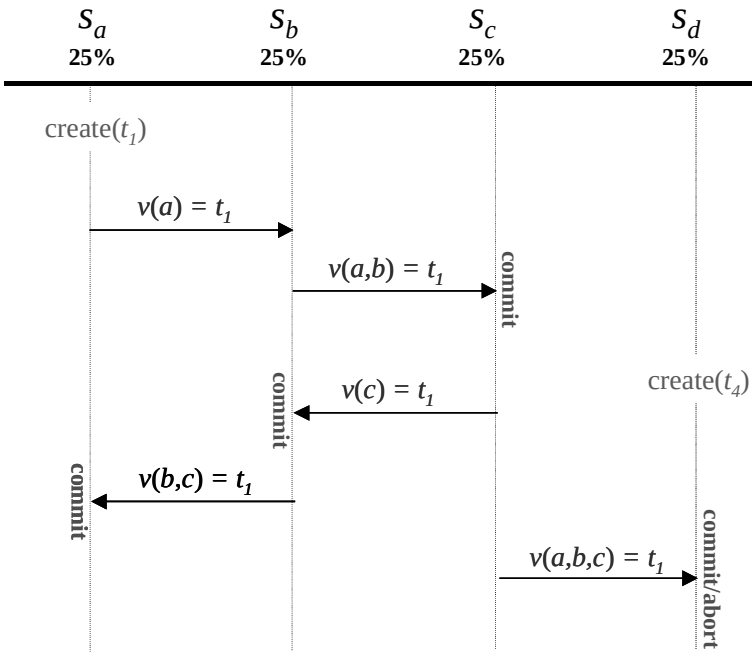

Figure 2: Illustrating update commitment

col, therefore, ensures mutual exclusion by guaranteeing that no two conflicting updates can both commit during the same election.

Each individual voter counts votes locally and deduces election outcomes independently. As a result, *voter v can commit an update without knowing all the votes, without complete knowledge of which voters have seen the update, and even without knowing which voters replicate the object*. After voter *v* has awarded election *i* to *k*, it will move on to election *i*+1.

3.2.2 Illustration

We illustrate our protocol in Figure 2 with an example scenario. The system has four servers, all with weight of 0.25. Server s_a creates a new update, t_1, votes for it, and sends a message describing t_1 and its vote to s_b via a synchronization session. Server s_b votes for t_1, and then later transfers notice of t_1 and both votes to s_c. After adding its

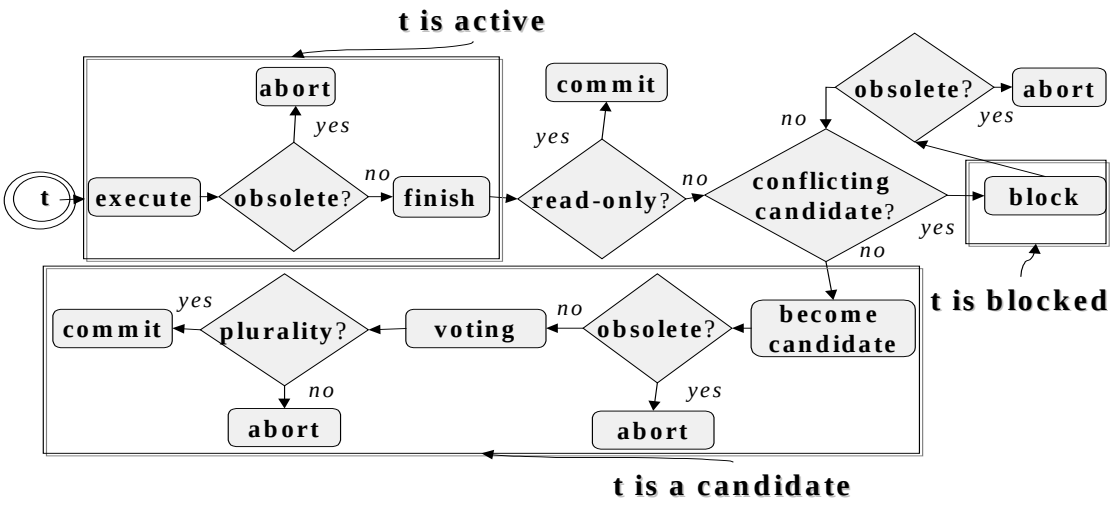

Figure 3: An update's life

own vote, s_c can commit t_1 because it has gathered a plurality. Later synchronization sessions move the votes back to s_b and s_a, which also reach the same commit decision.

Meanwhile, s_d has created a *conflicting* update t_4. Eventually, s_d learns of t_1 (and the corresponding votes for t_1 from s_a, s_b, s_c), commits t_1 (since $|votes(\{t_1\}| = 0.75$), and aborts t_4 (since t_4 becomes obsolete by the commitment of t_1).

3.3 Generalized Replication Protocol

Before delving into the fine detail, we give a quick overview of the *life* of a Deno transaction in Figure 3. A transaction is submitted by a client to any server, which executes it locally. Upon completion, the transaction either blocks (if the local server has seen a conflicting transaction) or becomes a *candidate*, which means that the update can become visible to other servers. Candidates are voted on, and eventually either committed if they corner a plurality of the system weight, or aborted. We now describe Deno's replication framework employed in detail.

3.3.1 Transaction Model

A transaction consists of a sequence of read and write operations on replicated data items. A transaction reads a set of *read items*, and updates a subset of the read items called *update items*. Our model assumes no blind-writes, i.e., data items are always read before being updated. Current values are tracked by associating a version number with each database item.

Each Deno server maintains an *active transaction list* that contains *active* transactions; i.e., transactions that are being executed. While a transaction is executing, it constructs a *transaction record* that summarizes the transaction's execution state. A transaction record for a transaction *t* enumerates the read items of *t* (along with the version numbers of the items *t* recorded when it read the items), the update items of *t*, and the new values of the update items written by *t*. A transaction does not perform any locking before it performs an operation; it simply performs the operation and records it in its transaction record. Furthermore, update transactions do not perform *in-place* updates; i.e., a transaction does *not* update the local copy of the item stored in the database. Instead, it simply writes the new value in its transaction record. When a transaction *re*-reads an item it has already updated, the transaction reads the value it has most recently written. Likewise, when a transaction *re*-writes a new value, it overwrites the value it has previously written in its record.

The items in the local copy of the database are modified, and their version numbers incremented, only when update transactions commit. In other words, transactions only access committed values. Depending on application semantics, however, this requirement can be relaxed, and transactions may be allowed to see new values written by

uncommitted transactions. Such a model may especially be useful in facilitating disconnected operation, and has been investigated in the context of Bayou [66].

We distinguish between two types of transactions, *queries* (i.e., read-only transactions) and *update transactions*. Both types of transactions execute entirely locally. However, queries are more light-weight in that a query can commit without further processing immediately after it successfully finishes its execution. Update transactions, on the other hand, must participate in a distributed commitment process after finishing execution. When an active transaction successfully completes its execution, it takes one of the following two paths: either (1) the transaction becomes a candidate transaction at its local server and participates in a distributed voting process that determines whether it commits or aborts; or (2) the transaction blocks and waits for the termination of other preceding, conflicting transactions before becoming a candidate. The blocked transactions are later considered for becoming candidates. In Section 3.7.4, we eliminate blocking in order to exploit commutativity.

3.3.2 Voting

We define the vote set V_i as the set of votes observed by server s_i. A vote $v \in V_i$, is a 4-tuple <*voter, trans, curr, tstamp*> where:

- *v.voter* denotes the server that casts the vote,
- v.trans denotes the transaction the vote is cast for,
- *v.weight* denotes the amount of weight *v.voter* voted for *v.trans*,
- *v.tstamp* is the value of *v.voter*'s local timestamp when *v.trans* was created (for our purposes, a timestamp is a monotonically increasing local counter that is incremented each time the server casts a vote).

Notice that the combination of "*voter*" and "*trans*" fields uniquely identifies a vote. In the rest of our discussion, we omit the other fields when they are not relevant or necessary to the understanding of the discussion.

Let $t_i.read$ be the read items and $t_i.update$ be the update items of t_i. Furthermore, let $t_i.vers(d)$ denote the version of item d observed (and recorded) by transaction t_i when t_i first accessed d.

Definition 1 (Conflicting transactions) Two transactions, t_i and t_j, are said to conflict if (1) their common read items have the same version numbers, and (2) one of the transaction's read items overlap with the other's update items. More formally, t_i and t_j conflict if:

$$t_i.vers(d) = t_j.vers(d), \forall d \in t_i.base \cap t_j.read \text{ , and}$$

$$(t_i.read \cap t_j.update) \cup (t_i.update \cap t_j.read) \neq \phi$$

A server, s_i, votes for a transaction by creating a vote, v, assigning a weight to v, and inserting it into V_i. The weight for a vote can be set in two distinct ways based on the state of the vote set. Server s_i votes with all its weight for transaction t_i if it has not already voted for a conflicting candidate transaction. Such a vote is called a *yes* vote and is an indication of the support of the server for the corresponding transaction. Otherwise, s_i votes with 0.0 weight, in which case the vote is called a *no* vote.

We now describe the voting process from the perspective of a single server. Each Deno server s_i maintains the following major data structures:

- A vote list, V_i.

- A *candidate* transaction list, C_i,. C_i consists of those update transactions that are known to s_i, have finished execution either locally or remotely, but have yet to be either committed or aborted at s_i.
- A *blocked* transaction list, B_i, consisting of locally completed transactions waiting to become candidates.
- A commit log, which contains a list of committed transaction records.

A server may create a *vote* for a candidate or locally completed transaction that does not conflict with any other candidate transaction for which it has also voted. If the server votes for a blocked transaction, the transaction becomes a candidate transaction and is moved from the blocked list to the candidate list. *Once cast, votes can never be retracted*.

As explained below, a transaction *t commits* at s_i when it is guaranteed that no conflicting transaction can obtain more votes. Transactions can be committed even without knowledge of complete group membership because the total amount of weight in the system is always 1.0. The protocol guarantees that all servers eventually reach the same commit decisions.

(*Voting Rule*) Server s_i considers voting for a transaction in the following three cases:

1. *When s_i learns about a new candidate transaction t after synchronizing with another server*; it votes yes for t if it has not already voted for a conflicting transaction; Otherwise, s_i votes no.
2. *When s_i commits or aborts a candidate transaction*; it considers all transactions *t* in the blocked list (i.e., all transactions waiting to become candidates) in inser-

tion order. For any such transaction that does not conflict with an existing candidate transaction; s_i votes *yes*.

3. *When s_i completes the execution of a local transaction t*; if there is no candidate transaction that conflicts with *t*, s_i votes *yes* for *t* and inserts *t* into its candidate list, C_i. Otherwise, s_i blocks *t* and inserts *t* into its blocked list, B_i.

There are two important implications of the cases stated above. First, there cannot exist *yes* votes from the same server for conflicting transactions. Second, locally completed transactions are blocked until the termination of existing *conflicting* candidate transactions.

3.3.3 Update Commitment

Given a server s_i, and its vote set V_i, we make the following definitions:

Definition 2 (Votes of a transaction) We define the sum of votes cast for a transaction t as:

$$votes(t) = \Sigma\ v.weight, \text{ where } v \in V_i, \text{ and } v.trans = t.$$

Definition 3 (Unknown votes of a transaction) We define the *unknown votes* of a transaction *t* as:

$$unknown(t) = 1.0 - \Sigma\ s.weight,$$

where s is a server who has already voted for t.

In other words, $unknown(t)$ is essentially the sum of weights of those servers whose votes for transaction *t* are *not* available. We now define the commit rule that server s_i uses to decide which candidate transactions to terminate (i.e., commit or abort) on the basis of local information (i.e., votes that are locally available). The fundamental idea is to commit a transaction when it is guaranteed that no other conflicting transaction

can gather more votes. Note that *server*(*t*) is defined as the server on which transaction *t* was executed.

(*Commit Rule*) A transaction $t \in C_i$ commits when: $\forall t' \in C_i$ such that t' and t conflict:

1. $votes(t) > votes(t') + unknown(t)$, or (*Plurality case*)
2. $votes(t) = votes(t') + unknown(t)$ and $server(t) < server(t')$ (*Tie-break case*)

The commit rule states that candidate transaction *t* can commit if it gathers the *plurality* of votes. Case 1 enforces mutual exclusion, as indicated before, by ensuring that no other conflicting transaction, which may or may not be known to server s_i, can gather more votes. Case 2 states that ties are broken using a simple deterministic comparison between the indices of the servers that created the transactions.

When a candidate transaction *t* commits at server s_i, s_i incorporates the effects of *t* into its database by installing the new values of the update items of *t* (available from *t*'s transaction record), and incrementing the version numbers of the local copies of those items. Finally, the transaction record of *t* is appended to the commit log. Note that servers must eventually garbage-collect their commit logs, as otherwise these logs will grow indefinitely. The candidate transactions and votes, on the other hand, do not need to be garbage-collected because they are discarded as soon as they become obsolete.

All active and candidate transactions whose read items are modified become *obsolete* and need to be aborted. More formally:

(*Abort Rule*) An active, blocked or candidate transaction, $t \in C_i$, is aborted at s_i when there exists a data item *d*, such that $d \in t.read$, and $s_i.vers(d) > t.vers(d)$, where *vers*(*d*) is the version of *d* observed by a transaction or server. Additionally, committing a

transaction causes all votes cast for an obsolete transaction to be discarded, i.e., a vote $v \in V_i$ is discarded if $v.trans = t$ and t has become obsolete.

3.3.4 Synchronization

We now discuss how two Deno servers synchronize their states. A pair-wise synchronization session essentially involves the propagation of (1) committed transactions, (2) candidate transactions, and (3) votes that are known to one server and unknown to the other.

In Deno, synchronization is controlled via version vectors [53]. In our model, each server s_i maintains an n-element vector, vv_i, where n is the number of servers, that describes the number of events of each other server *seen* by s_i. Element $vv_i[j]$ is a scalar count of the number of j's events that have been seen at s_i. For example, if $vv_1 = \{2, 3, 1\}$, then s_i has had two local events, has seen three events of s_2, and has seen one event of s_3's. There are three types of events: transaction commits, transaction promotions, and votes. A commit event is created whenever the local process commits a transaction, a promotion event is created whenever a transaction becomes a candidate on the server where it executed, and a vote event is created whenever a vote is cast.

In more detail, server s_i maintains a serial order, called *local ordering*, on all local commits, promotions and votes. We denote the j^{th} such event as e_i^j. As information about events is always propagated in local order, we can maintain the following invariant:

(Synchronization *invariant*) If s_i's version vector is vv_i, s_i has seen all events $e_j^1 \ldots e_j^{vv_i[j]}$, for all $j = 1 \ldots n$.

Synchronization is then straightforward. For purposes of exposition, we assume a unidirectional *pull* synchronization, although other modes are possible [23, 42]. When s_i pulls information from s_j, the following actions take place:

a. Server s_i sends vv_i to s_j.

b. Server s_j responds with all events e_k^l s.t. $l > vv_i[k]$ and $l \leq vv_j[k]$,

   for all $k = 1 \ldots n$.

c. Server s_i incorporates the new events in the same order that they originally occurred by:

   i. processing new commitments, candidates, and votes,

   ii. applying the voting rule, the commit rule, and the abort rule for all relevant transactions,

   iii. updating vv_i to the pairwise maximum of vv_i and vv_j

For purposes of exposition, we assumed *n*-dimensional vectors in the above description. As we described in the introduction, however, we do not assume that the number of servers is known to any server. Our implementation uses a set representation for the version vector, i.e.:

$$vv_i = \{(j, c_j), (k, c_k) \ldots\}$$

where each pair consists of a server id, *j*, and a count specifying the number of *j*'s events seen by s_i. The lack of a pair in vv_i describing some server *k* would be treated as an implicit pair (*k*, 0), meaning that no events from that server have been seen.

3.3.5 Protocol Illustration

Let $D(t)$ represent the set of read items of t, with the update items marked by a "*". For the following two examples, assume transaction records and initial vote sets as follows:

$D(t_1)=\{d_1, d_2^*\}$, $D(t_2)=\{d_1, d_2^*\}$, $D(t_3)=\{d_1, d_4^*\}$, $D(t_4)=\{d_2, d_3, d_4^*\}$;

$V_1=\{<s_1, t_1, 0.2>, <s_1, t_2, 0.0>, <s_1, t_3, 0.2>, <s_1, t_4, 0.0>,$

$<s_2, t_2, 0.2>,$

$<s_3, t_2, 0.25>, <s_3, t_3, 0.25>,$

$<s_4, t_2, 0.0>, <s_4, t_4, 0.25>\}$;

$V_4=\{<s_2, t_2, 0.2>,$

$<s_4, t_2, 0.0>, <s_4, t_4, 0.25>\}$.

Let $c(t_i, t_j)$ mean that transaction t_i conflicts with t_j. Then, $c(t_1, t_2)$, $c(t_1, t_4)$, $c(t_2, t_4)$, and $c(t_3, t_4)$.

Example 1: Consider server s_1. Using commit rule 1, s_1 commits t_2 because $votes(t_2)=0.45$, $unknown(t_2)=0.10$, and no conflicting transaction has a vote of more than 0.25, i.e. the maximum any conflicting transaction could gather is guaranteed to be less than t_2's current votes. Thus, s_1 aborts the candidate transactions, t_1 and t_4, and discards the votes that became obsolete at s_1 by the commitment of t_2. Afterwards, $V_1=\{<s_1, t_3, 0.2>, <s_3, t_3, 0.25>\}$.

Example 2: Suppose that s_4 now pulls information from s_1. The events propagated from s_1 to s_4 consist of the commit of t_2, candidate t_3, and votes $<s_1, t_3, 0.2>$ and $<s_3, t_3, 0.25>$. The arrival of the commit decision causes s_4 to commit t_2, and to discard t_4. At the end of the synchronization, $V_4=\{<s_1, t_3, 0.2>, <s_3, t_3, 0.25>\}$. At this point, s_4 votes

yes for t_3, and adds <s_4, t_3, 0.25> into V_4. It then applies the commit rule, and commits t_3 since *votes*(t_3)=0.70. Server s_4 then discards all relevant votes. Finally, V_4={}.

3.4 Providing Serializability: Extended Protocol

The base protocol ensures that queries always access transactionally-consistent data, and that update transactions are globally serialized with respect to each other. However, as the following example illustrates, the base protocol does not serialize update transactions with respect to all queries:

Example 3: Consider update transactions t_1 and t_2, where $D(t_1)=\{d_1*\}$ and $D(t_2)=\{d_2*\}$; and queries q_1 and q_2, where $D(q_1)=D(q_2)=\{d_1, d_2\}$. Suppose server s_1 commits t_1, q_1, and t_2; and server s_2 commits t_2, q_2, and t_1, in the order presented. This scenario is perfectly valid using the base protocol since t_1 and t_2 do not conflict and, thus, they are not necessarily serialized globally. The local serialization graphs at s_1 and s_2 will then be, $t_1 \rightarrow q_1 \rightarrow t_2$ and $t_2 \rightarrow q_2 \rightarrow t_1$, respectively. The global serialization graph, therefore, has a cycle.

This section extends the base protocol to provide strong consistency, which provides serializability to queries as well. We define strong consistency as follows:

Definition 4 (*Strong Consistency*) A query sees strong consistency if it is serialized with respect to both queries and update transactions. Strong consistency is characterized by an acyclic serialization graph, prohibiting both *update transaction cycles* and *multi-query cycles* (i.e., cycles involving multiple queries and one or more update transactions). This form of consistency guarantees globally-serializable executions [12, 13, 26].

The base protocol fails to provide strong consistency because non-conflicting update transactions are not necessarily globally serialized with respect to each other. We address this problem by forcing all update transactions to commit in the same order at all servers[1] by providing mutual exclusion among *all* transactions, rather than just among conflicting transactions as the base protocol does. We accomplish this by modifying the voting process such that each server votes *yes* for all candidate transactions (whether or not they conflict), but specifies a total order on all of it's votes. The commit process is then restricted so that only the *top* transactions, which are the candidate transactions that come first in any server's ordering, are considered for commitment.

More specifically, the protocol works as follows. Instead of choosing among conflicting transactions, a server votes *yes* for all candidate transactions as soon as they are received. The result is that V_i contains a *yes* vote by s_i for each candidate transaction, differing only in the votes' timestamps. The timestamps impose a total ordering on all votes created by s_i. A transaction may be committed if it gains a plurality of the *top votes*, where a top vote is the earliest vote in the vote set from a specific server.

More formally:

Definition 5 A vote $v \in V_i$ is said to be a *top vote* at server s_i if:

$$v.tstamp < v'.tstamp, \ \forall v' \in V_i \ s.t. \ v.voter = v'.voter.$$

[1] In fact, there are at least two other approaches to provide strong consistency. One approach is to include queries in the global voting process, which is clearly not desirable in our target environments. A second approach is to commit an update transaction after it is certified by all servers (similar to Agrawal's protocol [4]). This latter approach always requires contacting all servers in the system, which may be a serious restriction during times of low availability.

Definition 6 A candidate transaction $t \in C_i$ is said to be a *top transaction* at s_i if:

$$\exists v \in V_i \text{ s. t. } v.trans = t \text{ and } v \text{ is a top vote at } s_i.$$

Definition 7 We define the sum of votes cast for a *top* transaction t as:

$$votes(t) = \Sigma\, v.weight, \text{ where } v \in V_i \text{ s.t. } v.trans = t.$$

Definition 8 We define the unknown votes as:

$$unknown = 1.0 - \Sigma\, votes(t'), \text{ where } t' \text{ is a top transaction.}$$

Any server s_i may have up to n top votes and n top transactions, one of each for each of the n servers in the system. Notice the difference between Definition 3 and Definition 8. Definition 3 indicates that each transaction may have a different "*unknown*". In the modified definition, the "*unknown*" value applies to the vote set of an entire server. We now state the modified commit rule a server s_i employs:

Commit Rule (*Strong Consistency*): A *top* transaction $t \in C_i$ commits when, $\forall t' \in C_i$ such that t' is a top transaction:

1. (*Plurality case*) $votes(t) > votes(t') + unknown$, or
2. (*Tie-break case*) $votes(t) = votes(t') + unknown$ and $server(t) < server(t')$.

Aborts are handled as in Section 3.3.3. The following example illustrates how the extended protocol works:

Example 4: Consider server s_1 with the following transactions and votes as follows:

$D(t_1)=\{d_1, d_2^*\}$, $D(t_2)=\{d_3, d_4^*\}$, $D(t_3)=\{d_3, d_4^*\}$, $D(t_4)=\{d_1, d_3^*\}$; and

$V_1=\{$**$<s_1, t_1, 0.2, 6>$**, $<s_1, t_3, 0.2, 7>$,

$<s_2, t_2, 0.2, 4>$, $<s_2, t_1, 0.2, 5>$,

$<s_3, t_1, 0.35, 8>$, $<s_3, t_2, 0.35, 9>\}$.

The top votes are marked above using bold fonts, and the top transactions are t_1 and t_2 (recall that only the top votes and transactions are considered in the commitment decision at any stage). Server s_1 commits t_1, since *votes*(t_1)=0.55. No candidate transactions become obsolete in this case, as t_1 does not conflict with any other candidate. Therefore, V_1={**<s_1, t_3, 0.2, 7>**, **<s_2, t_2, 0.2, 4>**, **<s_3, t_2, 0.35, 9>**}. Transaction t_2 is *still* a top transaction and t_3 has also become a top transaction. At this point, t_2 commits since *votes*(t_2)=0.55. Server s_1 then aborts t_3, which has become obsolete, and discards the corresponding votes. Finally V_1={}.

3.5 Correctness Issues

3.5.1 Correctness and Consistency Issues

We now discuss the consistency level provided by the base voting protocol. First, we present the following lemma:

Lemma 1 (*Update Consistency*) If an update transaction t commits at one server, then t eventually commits at all servers.

Proof: Assume that transaction t_i committed at server s_i. Let *yes*(t_i) denote the set of servers that voted *yes* for t_i. Now consider another server s_j and another transaction t_j that conflicts with t_i. If all the votes cast by the servers in *yes*(t_i) are known at s_j, then s_j cannot commit t_j. Even if s_j may not know the votes cast by some of the servers in *yes*(t_i), that amount will be reflected in *unknown*(t_j), preventing t_j from committing at s_j. Therefore, s_j will eventually deduce the same outcome as s_i and commit t_i itself, or be told of the commitment of t_i by another server.

Lemma 2 (Update *Serializability*) The base voting protocol ensures global update serializability.

Proof: Assume that the protocol generates a non-serializable global schedule involving update transactions. Then, by the previous lemma, there exists a cycle in the global serialization graph[2] of the form $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$ where $t_1, t_2, \ldots, t_n$ are update transactions. Consider t_1 and t_2. Since $t_1 \rightarrow t_2$, t_1 and t_2 must conflict on some data item, d. Suppose t_1 commits before t_2 at server s. Assume now that t_2 committed at s' before t_1. We consider the three possible types of conflicts between t_1 and t_2 at s':

1. *rw* (t_2 *writes an item d which is then read by* t_1): Since t_2 updated d when it committed at s', the version number of d recorded by t_1 will be strictly smaller than the version number of the copy of d at the database of s'. This establishes t_1 as an obsolete transaction at s' and leads to t_1 being aborted.

2. *wr* (t_2 *reads an item written by* t_1): This case is the opposerver of the *previous* case. This time, t_2 cannot commit at s, as it is based on a version of d that has already been updated by t_1.

3. *ww* (t_2 *writes an item written by* t_1): This conflict type implies both *rw* and *wr* conflicts among t_1 and t_2. It is, therefore, subsumed by the previous two cases (since we do not allow blind-writes).

We therefore conclude that t_1 must have committed before t_2 at all servers. A straightforward induction based on the transitivity of the conflict relation asserts that t_1 commits before t_n at all servers. This eliminates the possibility of a cycle in the serialization graph, thereby producing the contradiction that completes the proof.

---

[2] A serialization graph [12] consists of nodes that represent transactions and directed edges that represent conflicting operations. A path from t_i to t_j indicates that t_i has to precede t_j in any equivalent serial ordering.

So far, we have addressed only update transactions and showed that our protocol guarantees the serializable execution update transactions alone. The protocol thus prohibits serialization graph cycles that contain only update transactions. We now extend our discussion to include queries, and demonstrate that the protocol supports a weak form of consistency, which we define below:

Definition 9 *(Weak Consistency)*: A query sees weak consistency if it serializes with respect to all update transactions, but possibly not with other queries [12, 13, 26].

In weak consistency, each query observes a serial order of update transactions, which is not necessarily the same order observed by other queries. However, weak consistency does ensure that queries always observe transactionally-consistent database states. In other words, a query does not see partial effects of any update transaction. Weak consistency prohibits both *update transaction cycles* (i.e., cycles involving only update transactions), and *single-query cycles* (i.e., cycles involving a single query and one or more update transactions).

Lemma 3 (Query-*Transaction ordering*) Let q be a query and t be an update transaction that respectively reads and updates item d. The dependency $q \rightarrow t$ implies that q commits before t commits, and $t \rightarrow q$ implies that t commits before q commits, at the execution server of q.

Proof: First consider $q \rightarrow t$. Query q reads d before t updates d. Query q must have committed *before* t committed. Otherwise, q must have been active when t committed, and the commitment of t would have aborted q (as q would have become obsolete). Now consider $t \rightarrow q$; q reads d after t updates d. In this case, q must have

read *d* and committed *after t* since any update transaction (including *t*) installs its updates and commits atomically.

Theorem 1 *(*Weak Consistency*) The base protocol described above provides weak consistency.*

Proof : Assume that there is a single-query cycle, involving query *q* and update transactions $t_1, t_2, \ldots, t_n$, of the form $q \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow q$. Consider $q \rightarrow t_1$. By Lemma 3, *q* must have committed before t_1 at the execution server of *q*, say *s*. By Lemma 1, t_1 commits before t_n at all servers. Therefore, *q* must have committed before t_n at *s*, prohibiting the single-query cycle assumed initially. Moreover, we know by Lemma 2 that there can not be any update transaction cycles. Therefore, we conclude that the protocol provides weak consistency.

3.5.2 Termination Issues

(*Eventual Termination – Base Protocol*) A candidate transaction eventually terminates (i.e., commits or aborts).

Proof (*Sketch*): Suppose there exists a candidate transaction *t* that never terminates. We can partition the set of servers into three disjoint subsets as (1) the servers that voted *yes* for *t*, *yes*(*t*), (2) servers that voted *no* for *t*, *no*(*t*), and (3) servers that have not yet observed *t*, denoted *unknown_servers*(*t*). Assuming that information eventually propagates to all servers, *unknown_servers*(*t*) will eventually become empty. Let the conflict set of *t*, *CS*(*t*), denote the set of candidate transactions that conflict with *t*. When *unknown_servers*(*t*) becomes empty, *CS*(*t*) can not grow further due to the voting rule (see Section 3.3.2), since all servers voted for either *t* or another transaction that conflicts with *t*. Now consider the case where all candidate transac-

tions $t' \in CS(t)$ are observed at all servers. At this point, *votes*(t) and *votes*(t') for all $t' \in CS(t)$ are determined. As a result, *unknown*(t) and *unknown*(t') for all $t' \in CS(t)$ are all 0.0. Therefore, the commit rule will commit the transaction with the most votes (or in the case of a tie the one executed at the server with the smallest id) and the rest will be aborted, thereby contradicting our initial claim. Moreover, a deadlock situation due to vote dependencies can not exist. Such a deadlock has to involve a cycle of the form *votes*(t_1) > *votes*(t_2) > …. > *votes*(t_n) > *votes*(t_1) where t_1, t_2,…, t_n are candidate transactions. Since both *votes*(t_1) < *votes*(t_n) and *votes*(t_n) < *votes*(t_1) can not be true at the same time, we conclude that such a deadlock cannot exist.

Now consider a blocked transaction *t*. Transaction *t* will eventually become a candidate since 1) the set of candidate transactions that *t* is blocked after will all eventually terminate (see earlier discussion), and 2) the blocked transactions are considered in the order they are entered into the blocked list, so *t* is not going to wait indefinitely before being considered for candidacy.

Lemma 4 (*Eventual Termination – Extended Protocol*) A candidate transaction eventually terminates (i.e., commits or aborts).

Proof : We consider two cases. First consider a top transaction, *t*. Assume eventually that the top votes cast by all servers are known (i.e., *unknown*=0.0). Let *U* denote the non-empty set of top transactions that obtained more votes than the remaining top transactions. If *U* contains a single transaction, then that transaction commits. Otherwise, *U* contains a set of transactions and the top transaction whose execution server has the smallest id commits. If *t* is the transaction that commits, then we are

done. Otherwise; if *t* becomes obsolete, then it gets aborted, or else *t* remains a top transaction. The process repeats and *t* either becomes obsolete and is aborted, or eventually gets enough top votes and commits. Now consider a *non*-top transaction, *t*. Transaction *t* will either become obsolete and get aborted by the commit of a top transaction, or eventually become a top transaction itself and terminate (by the discussion in the first case). Therefore, we conclude that a candidate transaction always terminates.

3.5.3 Correctness of the Extended Protocol

Lemma 5 (*Global Update Consistency*) The protocol presented above ensures a unique global commit order on the set of update transactions.

Proof : In particular, we show that each server commits the same update transactions in the same order. Assume that t_i is the very first transaction that committed at server *s*. Extending the discussion presented in the proof of Lemma 1 by treating the top transactions to be the only conflicting transactions in the system, we can conclude that t_i is the first transaction to commit at all servers. A straightforward induction on the sequence of committed transactions concludes the proof.

Theorem 2 *(*Strong Consistency*) The protocol presented above provides strong consistency and serializability.*

Proof: Lemma 1 ensures that there are no update transaction cycles. Without loss of generality, assume that there is a multiple-query cycle of the form

$$q_1 \rightarrow t_1 \rightarrow q_2 \rightarrow t_2 \rightarrow \ldots \rightarrow q_n \rightarrow t_n \rightarrow q_1.$$

Consider $q_1 \rightarrow t_1$, which implies that there is an item *d* read by q_1 and then updated by t_1. By Lemma 3, q_1 commits before t_1 at the execution site of q_1, say s_1. Now

consider $t_1 \rightarrow q_2$ and $q_2 \rightarrow t_2$, which together imply that t_1 commits before q_2 and, therefore, before t_2 at the execution site of q_2, say s_2. Therefore, by Lemma 1, t_1 commits before t_2 at all sites. Using a straightforward induction, we can say that t_1 commits before t_n at all sites. However, $t_n \rightarrow q_1$ implies that t_n commits before q_1 at s_1, creating the contradiction that concludes the proof.

3.6 Weight Management

Timely update commitment depends on being able to assemble a majority to vote on updates. The cost of assembling a majority is highly dependent on the weight distribution of the object replicas. The best weight distribution depends on the non-trivial interplay among several factors such as expected availability of individual servers, interconnectivity, and application characteristics. In general, replicas that are more reliable or better interconnected should receive more weight [10].

In this section, we discuss mechanisms that enable the implementation of arbitrary weight distribution policies while still maintaining the correctness of the voting protocol. Note that the issue of finding *optimal* weight distributions is outside the scope of this thesis and has been addressed by several previous work (e.g., [8, 10, 11, 39, 47]).

We first describe how replicas are created and weight is initially allocated. We then discuss protocols for dynamically reallocating weight while maintaining the mutual exclusion properties of our voting protocol. We also investigate the cost of migrating weight distributions towards target distributions when initial allocations are not ideal.

3.6.1 Replica Creation and Retirement

Objects are initially created with a total weight of 1.0 (or any other system-wide fixed amount), held by the creating server. A new replica is created through a request to a

server that already has a replica. The response to such a request contains both an object replica and some amount of weight that is subtracted from the weight held by the responding server. A replica can be retired using a similar pair-wise mechanism in which the weight held by the retired replica is transferred to another replica.

Initial weight allocation is non-trivial because not only servers do not have complete knowledge of the size of the anticipated set of servers, but also there is generally not even a central location that can be expected to receive all weight requests. Instead, each server receives an initial block of weight from the server who responds to its initial request to create a replica. This respondent can be any server, so we can clearly not guarantee to achieve a given distribution merely by allocation.

However, Deno applications can direct initial weight allocation by providing a hint at object creation as to how many replicas are expected to be created. This hint allows Deno to allocate weight to replica requests in a way that provides a uniform level of weight for the expected number of replicas. For this to work, new replicas must be created from the original replica. This choice can also be controlled through runtime hints.

3.6.2 Weight Redistribution Mechanisms

Without any restricting assumptions, it is not likely that initial weight allocations will approach the target distributions. Furthermore, the *optimal* distribution in dynamic environments and systems may change continuously. It is crucial, therefore, to provide mechanisms to redistribute weight dynamically throughout the lifetime of the object.

Deno uses peer-to-peer *weight exchanges* to incrementally change existing weight distributions into arbitrary target distributions. A peer-to-peer weight exchange in-

volves a pair of servers communicating and redistributing their total weight according to some redistribution policy.

Requirements

We now describe in detail how to implement peer-to-peer weight exchanges while maintaining the correctness of the voting protocol. Let s_i and s_j be two servers that exchange weight, and, without loss of generality, let x be the weight to be transferred from s_i to s_j. Further, let e_i denote the most recent election in which s_i voted, and e_j denote the current election of s_j. For correctness, the protocol has to guarantee that:

1. x is not used more than once in any election, and
2. x is available to every election.

Requirement (1) is needed in order to prevent servers from reaching different conclusions on the outcome of a single election. The need for requirement (2) is less obvious. Any amount of weight that effectively *disappears* from an election can prevent an election from closing. In the case of server failures, the rest of the system cooperates to reallocate the lost server's weight. However, in this case no server has failed, and without restriction (2), a loss of weight could halt the entire system.

In order to satisfy the two correctness requirements presented above, we define e, the election in which s_i decreases the amount of weight it uses by x and s_j increases the amount of weight it uses by x, as:

(i) if $e_i < e_j$, then $e = e_j$

(ii) if $e_i \geq e_j$, then $e = e_i + 1$

Case (i) indicates that s_i last used x in an election which has already been completed by s_j. Server s_j can immediately use x in its current election since the requirement (1)

presented earlier is guaranteed. In fact, s_j *must* use x in its current election in order to satisfy requirement (2). Case (ii) complements the former case by handling the situation in which s_i last used x in an election which has *not* yet been completed by s_j. In this case, s_j cannot use x before it completes all those elections in which s_i has used x in order not to invalidate requirement (1).

Case (i) also implies that it is possible that s_j increase its vote during an election for which s_j has already cast a vote. Therefore, a server that observes two different votes from the same server for the same election uses the vote with more weight, since cases (i) and (ii) together guarantee that it is not possible for a server to decrease its weight in an election it has already voted. On the other hand, s_j *cannot* change the candidate update for which it has already voted in a given election.

Weight Representation

The above weight exchange protocol requires that servers maintain extra information regarding the amount of their weight they can use in a given election. Specifically, each server, s_i, maintains a set of <*weight, election number*> pairs as follows:

$$\{< w_{i1}, e_{i1} >, < w_{i2}, e_{i2} >, \ldots, < w_{im}, e_{im} >\},$$

where e_{ij} represents the election number in which s_i starts voting w_{ij} of its weight, $\forall j = 1 \ldots m$. The weight voted by s_i at election e is then:

$$\sum w_{ij} \text{ , s.t., } e_{ij} \leq e$$

Weights transferred during exchanges are also represented using the same set representation. It is, therefore, possible for a server to exchange the weight that the server

has not even started using yet without sacrificing correctness. The protocol guarantees that for every election *j*:

$$\sum_{i=1}^{n} w_{ij} = 1.0 ,$$

where *n* is the number of servers in the system. Note that the protocol presented above also applies to the weight transfers performed during replica creation and retirement.

An important feature of peer-to-peer exchanges is that the final weight distribution does not have to be known by any participating server. Rather, each server indicates a target weight and receives weight proportional to this weight. More formally, let w_i and w_i' denote the weights that s_i holds before and after a weight exchange, respectively. Assume that two servers, s_i and s_j, that desire to eventually hold target weight levels of t_i and t_j respectively, perform a weight exchange. In this case, the new weight values after the weight exchange will be:

$$w_i' = (t_i / (t_i + t_j)) (c_i + c_j)$$

$$w_j' = (c_i + c_j) - c_i'$$

3.6.3 Weight Redistribution Policies

Given any initial distribution, randomized peer-to-peer weight exchanges can be used to converge to *any* target distribution, even without complete knowledge of the servers in the system. For example, consider the optimal availability weight distribution given by Amir and Wool [8], where weight is distributed proportionally to the individual availability of servers. Without complete knowledge of all availabilities in the system, it is not possible for any individual server to determine its own target weight. How-

ever, two servers participating in a peer-to-peer weight exchange can converge to these unknown targets by redistributing their own weights proportionally to their own availabilities (i.e. t_i is set equal to the availability of s_i). Therefore, it is sufficient for each server to have knowledge of only its own availability. For instance, servers can converge to a uniform distribution without knowing the total number of servers; during a pair-wise weight exchange, two servers can simply share their total weight equally (i.e., t_i and t_j are set equal).

In synchronous quorum systems, it is known that when all servers have the same independent failure probability f, then availability is maximized using a primary-copy system if $f > ½$, or using a majority system if $f < ½$ [8]. If the failure probabilities are different and all of them are smaller than ½, then optimal weights are defined as:

$$w_i = \log_2\left(\frac{1-f_i}{f_i}\right),$$

where w_i is the weight at s_i, and f_i is the failure probability of s_i.

On the other hand, measuring the availability of an epidemic protocol is not necessarily well defined. The availability of a typical quorum protocol is the percentage of time that a quorum is simultaneously connected and able to communicate. However, epidemic protocols do not require any server to be able to talk to more than one other server in order to make progress. While this implies that availability might be a poor metric, we can capture the affects of disconnections by looking at its effect on commit performance.

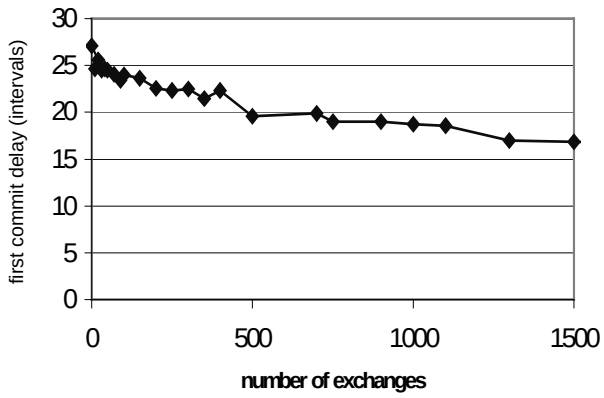

Figure 4: Effects of availability-based weight redistribution (100 servers)

We conducted a serious of experiments to quantify the potential performance improvements attainable through weight redistribution based on individual server availabilities. For purposes of these experiments, servers disconnect at any synchronization period with some probability. The disconnection probabilities are assigned to servers using a zipf distribution. During each synchronization session, two servers redistribute their total weight proportional to their availabilities; where availability of a server is defined to be 1.0 less the disconnection probability of the server. As in the earlier experiments, weight is initially uniformly distributed in the system, and servers perform synchronization once every synchronization period on average.

Figure 4 shows the change in the commit performance of the system as the system performs weight exchanges. Commit delays decrease as the number of weight exchanges performed in the system increases. Initially, i.e., when the weight is distributed evenly across servers, the commit delay is about 27 intervals. The commit delay decreases down to about 24 intervals after 50 exchanges, 20 intervals after 500 exchanges, and 17 intervals after 1500 exchanges. These figures correspond to, respec-

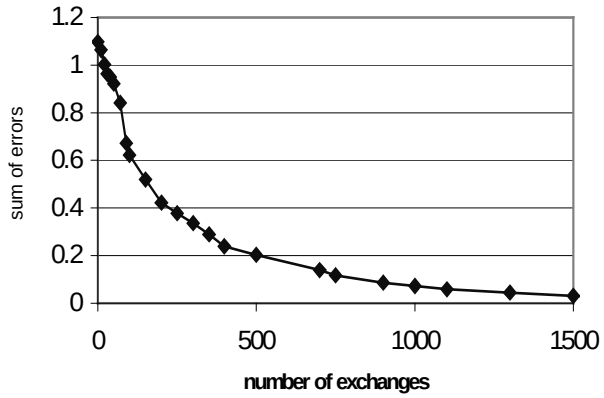

Figure 5: Difference between existing and desired weight distribution (100 servers)

tively, 9%, 28%, and 38% improvements with respect to the initial commit speed of the system. After 1500 weight exchanges (not shown), we do not observe more improvement in commit performance; the performance essentially remains the same. Figure 5 investigates the reason for this behavior by plotting the sum of differences between the existing weight distribution and the distribution where weight is distributed proportional to relative availabilities at servers as the system performs pair-wise weight exchanges. The figure reveals that this difference diminishes very fast. In fact, the difference becomes after zero after approximately 1500 exchanges, implying that the system converged to its target distribution.

Note that an existing weight distribution can be migrated to a target distribution without the need for any server to have global information (e.g., number of servers, current weight distribution, etc). The ability to achieve global goals with only local information is one of the reasons that this mechanism is especially suited for highly-dynamic environments and systems.

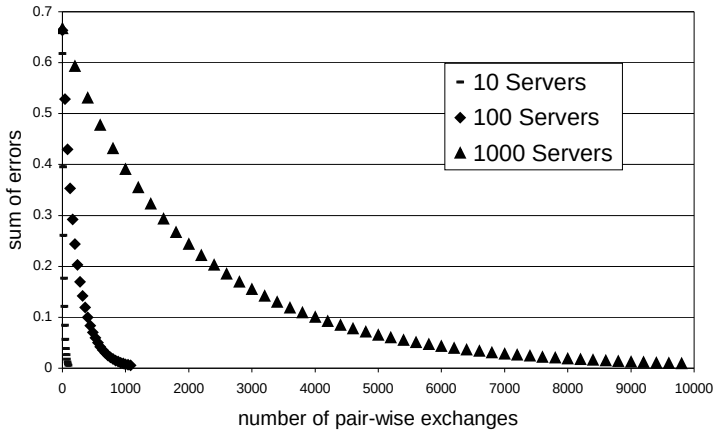

Figure 6: Converging to a target distribution with randomly selected peer-to-peer weight exchanges

3.6.4 Convergence Rates

We further investigated the convergence speed of our pair-wise weight redistribution mechanism. We observed that randomly-selected, pair-wise weight exchanges allow an existing weight distribution to converge *exponentially* fast to any target distribution. We showed this result analytically for three servers, and the experimental results suggest that the proposition generalize when there are more than three servers.

Figure 6 shows the mean difference between thousand pairs of randomly chosen initial and target weight distributions versus the number of (randomly-selected) pair-wise weight exchanges performed in the system for different system sizes. Each weight exchange involves the two servers redistributing their weights proportional to their weights in the target distribution as described previously. As expected, the larger the number of replicas, the more the number of weight exchanges required to converge to the target distribution. However, the shapes of the plots in the figure demonstrate that the difference between the target and the existing distributions diminishes quickly.

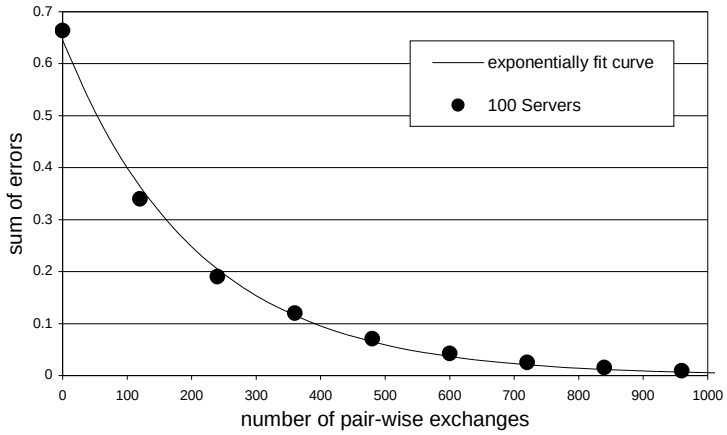

Figure 7: Exponentially diminishing error (100 servers)

In fact, the convergence to target distribution happens *exponentially* fast. We illustrate this result in Figure 7, which plots the corresponding data points for 100 servers along with an exponential curve fitted to our data. The exponential curve shown in the figure has the form $y = ae^{bx}$ where (a,b) = (0.6453, -0.0047). The corresponding exponentially fit curves for 10 and 1000 servers (not shown) can be obtained, respectively, using settings of (0.6107, -0.0514) and (0.6431, -0.0004).

It is also worth noting that Barbara and Garcia-Molina demonstrated that autonomous, incremental methods for determining new weight distributions, while being more flexible, can yield as much availability as those methods that require having complete knowledge of the state of the system [11].

3.7 Extensions

3.7.1 Fault-Tolerance Issues

In this section, we discuss approaches to detect and handle failures in Deno. We first describe how failures can be detected using general techniques (e.g., timeouts), or protocol-specific techniques. We then describe how to handle failures using a proxy

mechanism. When the failure is benign (e.g., a voluntary disconnection), a server transfers its voting rights to another server that votes in place of the disconnected server. Since it is not possible for the failed server to assign a proxy server in the case of involuntary disconnections or long-term network partitions, we propose that the rest of the system collectively elect a proxy for the failed server in those situations. Proxies, therefore, represent *failed* servers in the system and are selected either by the failed server itself (in the case of expected disconnections) or through proxy elections.

Failure Detection

Detection of failures is problematic in mobile, weakly-connected, and wide-area environment. Different failures with different characteristics can occur, and detection of some of these failures modes (e.g., network partitions) in general is known to be undecidable. It is thus essential that we treat all kinds of failures uniformly, rather than trying to capture and solve each independently. For this reason, we treat all kinds of failures (including network partitions) as independent, unexpected/involuntary disconnections or failures of a set of servers. In this way, we reduce the problem of general failure detection to that of detecting failed servers.

The standard approach to detect failures in distributed systems is to use timeouts. Whenever a server attempts to contact another, and fails to do that within a certain time period (i.e., timeout threshold), the server may deem the other as failed. This straightforward mechanism requires an effective setting of the timeout threshold. Even if an optimal threshold setting is found, the dynamic nature of the environment requires dynamically changing timeouts.

In addition to the standard techniques that are applicable by all distributed systems, we can also make use of schemes that are specific to the Deno protocol. For instance, one way to detect a failed Deno server is to count the anti-entropy requests that were not responded by the server. In order to implement this scheme, each server can keep a count of such unresponded requests and pass this information to others during anti-entropy sessions. When it is determined that a server has not responded to a certain number of anti-entropy requests, the server can be declared as failed.

A second Deno-specific approach is to count the number of updates that commit without a vote from the server. Only the servers that are unexpectedly out of touch with the rest of the system cannot vote since the votes of other servers are cast either by themselves or by their proxies. This approach also requires some extra information to be maintained in the system for some time. More specifically, each site keeps track of which servers (that voted previously) have not voted for some number of elections. If a site is observed not to have voted for some predetermined number of elections, then it can be declared as failed.

Note that in a distributed environment, it is always possible that a server be falsely declared as failed. It is also possible that a server may correctly be identified as failed, but by the time some action is taken it may already have rejoined the rest of the system. It is, therefore, crucial that the protocol function correctly under such subtle situations.

Proxy Assignment

Deno transparently handles voluntary, planned-for disconnections by having a *primary* server engage a proxy server to vote in its place while the primary is disconnected. A

vote cast by a proxy server is then indistinguishable to other servers from the situation where a server votes and disconnects. The use of proxies can prevent degradation in the overall commit rate when devices have expected, planned-for disconnections. In fact, proxies can even improve commit latency because weight is concentrated in fewer servers, and fewer rounds of communication are required to establish a quorum.

We now illustrate the proxy assignment process by a simple example. Consider a server, s_i, planning to disconnect from the network. Server s_i contacts another server, s_j, assigns s_j as its proxy, and disconnects from the network. Now suppose s_j sees a new candidate update. When processing the candidate, s_j not only casts a vote for itself, but it also casts a vote in place of s_i using s_i's weight. This vote is then no different from the vote that would be cast and disseminated by s_i if s_i were connected. Other servers have no way of knowing about the proxy assignment, and therefore the whole proxy mechanism is transparent to the rest of the system.

Proxy Elections

Once a permanent disconnection is detected, action must be taken to recoup the weight held by the disconnected server. Loss of this weight can either slow down or completely prevent updates from being committed. As mentioned above, unplanned, unexpected disconnections may block system progress. To illustrate this case, consider the following example: Assume that there are three servers s_i, s_j, and s_k that share the total weight equally (i.e., each server holds about 0.33 weight). Assume further that s_i and s_j initiate conflicting updates on the same item. Concurrently, s_k disconnects unexpectedly or partitions away from the others. In such a scenario, s_i and s_j in contact cannot make any progress until s_k rejoins and its weight becomes available again. The funda-

mental problem is that some amount of weight is not available anymore for voting purposes. Such a situation can typically occurs when (1) a server makes an unplanned disconnection without designating a proxy, or (2) a network partition occurs and some servers get partitioned away from the rest of the system.

In the case of unexpected disconnections, failures, or network partitions, Deno servers collectively elect a single server to act as a proxy to the unavailable, failed server(s). Proxy elections are performed similarly to coordinator elections protocols widely used by many distributed protocols [12], using the decentralized voting scheme described earlier.

After detecting a failure, a server initiates a *proxy election* (PE) update that indicates the server's intention to become the proxy for the failed server. As with other changes to objects, a proxy election update is a special type of operation on an object. The election update, therefore, must be committed before it can take effect. When a PE update is committed, the server that initiated the update becomes the proxy for the failed server.

Electing a single server to act as a proxy has several desirable properties:

- The commitment of a PE update only affects the processing of the server that initiated the update;
    - The problem of blocking (i.e., the case where voting can not proceed to completion due to unavailable votes) does not occur, because the server that initiated the PE update can retract it anytime without sacrificing correctness (such a unilateral retraction is not possible for regular updates due to decentralization); and

When the failed server rejoins, it can simply contact the elected proxy and get back its weight; recovering to its state immediately before failure. This may especially be valuable when a server is *falsely* declared as failed.

When the failed server rejoins, it becomes aware that it has been assigned a proxy using related logs that it obtains from other servers. The failed server then contacts the elected proxy to get back its weight just like it would have done if it had used the regular proxy mechanism.

Deno treats all updates, including PE updates, uniformly and uses its voting scheme to commit them. One implication is that a proxy election can occur only if a majority of the current weight is available. Such a restriction is necessary to prevent parallel proxy elections in multiple partitions after a network failure. When a failed server rejoins the computation and learns about the proxy election, the server resets its weight to zero. The server may then request its weight back from the elected proxy server or obtain weight from other servers through peer-to-peer weight exchanges.

If a failed server does not request its weight back for a long time, the proxy may decide to stop acting as the proxy of the failed server (mainly for garbage collection purposes). The proxy server can then declare the weight of the failed server as its own, or distribute it to other sites as in the case of weight revaluation.

3.7.2 Synchronization Mechanisms

We have so far assumed that the targets of anti-entropy sessions are chosen at random. This assumption gives us good expected-case performance with a variety of levels of connectivity, with a maximum slowdown from the optimal directed strategy of only

1.7 [58]. Additionally, this approach does not require any knowledge of current weight distributions.

It is, however, possible to improve commit performance by more intelligently choosing synchronization partners. One way to improve upon random target selection is to exploit weight information. When committing an update, necessary votes can be gathered faster if *rich* servers (i.e., servers holding relatively more weight) are favored when selecting anti-entropy partners. Since servers typically can hold different weights for different objects, skewing target selection based on weight weights is only useful when weights for all objects are skewed similarly or when we desire to focus on a particular object. Secondly, this approach fails to account for weak connectivity, which is one of the main characteristics of our target environment. We can, thus, instead choose to skew target selection based on known server availability information. Note that the weight information needed in the weight-based selection scheme is already available within the information maintained for voting (see Section 3.3). For the availability-based scheme, server availability statistics can be gathered over time.

3.7.3 Exploiting Application-Specific Commutativity Information

Disconnected and weakly-connected environments can be characterized by disconnections, high communication latencies, and incomplete connectivity. Therefore, applications running on top of these environments and systems need be designed so as to minimize conflicts among updates in order to avoid high abort rates [29]. One approach to accomplish this is to have the applications export domain-specific semantic information that can be used by the underlying system to modify application's consistency requirements [66]. To this end, Deno's extended protocol supports *commutativ-*

ity procedures to exploit application-specific commutativity information. A commutativity procedure is a simple query over the database specifying an acceptance criterion [29]. If the query is satisfied, the transaction is considered as valid with respect to the current state of the database (event though it may already become obsolete in the traditional sense).

Deno executes a transaction's commutativity procedure (if it exists) if and when it becomes obsolete. If the acceptance criterion implemented by the procedure is satisfied, the transaction is kept alive and not aborted. Note that the use of commutativity procedures does not in any way change the consistency guarantees provided by the Deno's protocols. The protocols still ensure a consistent global ordering of transactions. This guarantee is the reason that commutativity procedures are only supported by the extended protocol. The base protocol does not guaranteee a global ordering of all transaction commits, so the view of the database seen by the commutativity procedures, and the answer returned by any such procedure, would be site-dependent.

3.7.4 Speculative Voting and Update Propagation

Recall from Section 3.3 that a transaction that completes its execution is blocked until the local server has decided whether to commit or abort all conflicting candidate transactions. Blocked transactions can only proceed and participate in the global voting protocol after the conflicting transactions are terminated. Such a conservative blocking approach is used by many pessimistic distributed protocols [12, 34].

An optimistic alternative is for the blocking phase to be skipped by having servers immediately vote for all transactions as soon as they finish their local execution. These transactions are immediately candidates to be added to subsequent anti-entropy ses-

sions. The advantage of such speculative voting is that transactions can make progress, in terms of gathering votes, while the system is still deciding the fate of prior transactions. Speculative votes are useful when previous conflicting transactions are aborted. As shown below, the advantage conferred by this technique is larger when there are commuting transactions in the system.

Example 5: In order to illustrate the potential benefits of speculation, consider a system of three servers, s_1, s_2, and s_3, and two conflicting transactions, t_1 and t_2. Assume that both transactions complete execution at s_1 in the order presented. In the (standard) non-speculative protocol, t_2 blocks until t_1 gets propagated through the system, either gets committed or aborted, and this termination information is received (or made independently) at s_1. If t_1 gets committed, then no sacrifice is made because t_2 is aborted in any case. On the other hand, if t_1 gets aborted, t_2 will resume and the whole process will be repeated to decide on the fate of t_2. If speculative propagation is used, however, t_2 will be propagated along with t_1 and will gather votes that will be immediately available for use in case t_1 gets aborted. In this case, speculation will cut down the commit delays significantly by making progress during an interval otherwise wasted waiting for other transactions to terminate. The cost of speculation is that some transactions that will eventually get aborted will also be propagated through the system unnecessarily, resulting in a waste of communication bandwidth.

In order to support speculative transaction propagation and voting, we extended our protocols such that servers vote for locally completed transactions immediately (without blocking after any other transaction) and propagate them through the system. In a sense, the system makes available the transactions and votes that may be required in

future commitment processes speculatively, without knowing whether they will be necessary or valid.

In order to support speculation in Deno, we made the following modifications to the voting rule: Whenever a transaction locally completes execution, the server immediately votes for it as if it is a candidate transaction received from another server, inserting the transaction to the candidate list. More specifically, when a transaction *t* locally completes its execution, the server votes *yes* for *t* if there is not another conflicting candidate transaction. Otherwise, the server votes *no* for *t*. Notice that this change eliminates the need for a blocked transaction list, as all local transactions become candidates as soon as they locally complete execution.

Somewhat surprisingly, ww do not need to make any modifications to the commit rules to implement speculative voting and update propagation. The two commit rule versions, both the weak- and the strong-consistency versions, transparently decide on which votes should be considered in the current commitment decision, thereby ignoring the speculated votes for the current commit decision. The strong-consistency rule utilizes timestamp information (already available at the votes) to take into account the only the *top* votes as current votes. The weak-consistency rule implicitly utilizes *yes* and *no* votes. For each transaction *t*, only the votes from a given server that have lower timestamps than the timestamp of that server's vote for *t* are used in the current commit decision.

3.8 Deno Prototype

We now describe the design and architecture of the Deno object replication system. The overriding goal of the Deno project is to investigate replica consistency protocols.

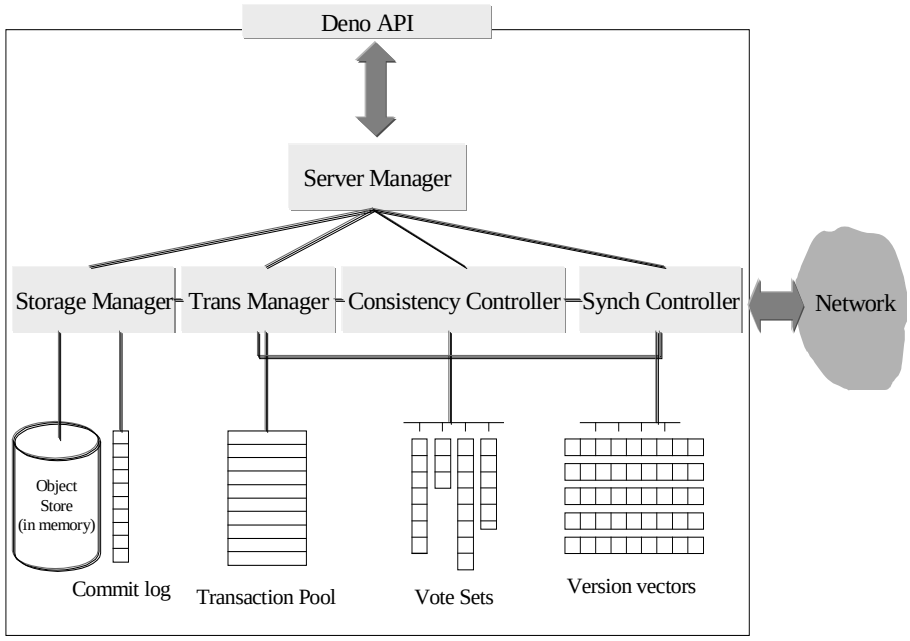

Figure 8: Basic Deno architecture

We are therefore not motivated to build large and complicated interfaces to the object system. By the same token, we feel that lightweight interfaces are the appropriate choice for many applications, and that more complex services can be efficiently built on top of Deno services if needed.

The basic Deno API calls allow new servers, objects, and object replicas to be created, and object replicas to be updated and destroyed. Servers use proxy calls to delegate voting rights before planned disconnections. Notification calls are used to learn about the termination status of the updates. The sparse interface avoids burdening applications with unwanted or unneeded abstractions and functionality.

Figure 8 illustrates the basic Deno server architecture, consisting of the following components:

- The *Server Manager* is in charge of coordinating the activities of the various components. It handles client requests by implementing the basic Deno API.
- *The* Consistency *Controller* implements the decentralized voting protocols used by Deno. In particular, it maintains a *vote pool* that summarizes the votes known to the server.
- The *Synch Controller* is responsible for implementing efficient synchronization sessions with other Deno servers by maintaining *version vectors* that compactly summarize the events of interests from other servers. This component implements different synchronization policies that specify when and with whom to synchronize. In the current implementation, it implements a naïve policy that chooses synchronization partners randomly at regular intervals.
- The *Trans Manager* is mainly responsible for the local execution of transactions. It maintains a transaction pool that contains all active (i.e., non-obsolete) transactions known to the server.
- The Storage *Manager* provides access to the *object store* that stores the current committed versions of all replicated objects at the server. The object store is currently a simple in-memory database.

The prototype makes relatively few demands on the operating system and is therefore highly portable. All communication is made on top of UDP/IP. The current prototype runs on top of Linux and Win32 (i.e., WindowsNT/CE) platforms. All communication is made on top of UDP/IP. Deno's source consists of ≈20,000 lines of multi-threaded C++ code, and has a footprint of ≈200KB.

Parameter	Description	Setting
Synch Period	Mean synchronization period (uniform)	0 – 5 (secs)
Transaction Rate	Mean transaction generation rate (uniform)	0 – 25 (trans/synch period)
Num Servers	Number of Deno servers	3 – 15
Trans Size	Number of items updated by a transaction (uniform)	0 - 5
Commutativity Ratio	The probability that a transaction is acceptable on a given db state	0 – 1

Table 1: : Primary experimental parameters and settings

3.9 Performance Evaluation

This section describes the performance of the Deno prototype. Note that the primary advantage gained in combining weighted voting with epidemic information flow is in increased availability. Rather than belabor the obvious, we instead investigate the performance impact of such a combination. Our earlier work [41, 42] presented simulation results showing the potential for good performance on single-object updates. This work presents results for multi-object transactional updates on a real system. Additionally, this work introduces and characterizes the performance impact of speculative voting and information dissemination.

3.9.1 Experimental Methodology

We performed the experiments on a cluster of 15 Linux machines, each running a single copy of the Deno server. Each machine has two 400 MHz Pentium II's, and 256 MBytes of memory. However, none of the protocols discussed below ever came close to consuming all of a machine's resources. We have intentionally set our communication rates low in order to reflect the constraints of our expected environments. Instead,

our performance evaluation concentrates on relative performance by comparing representative protocols.

The machines were connected via a 100Mbps Ethernet network and the Deno servers communicated using UDP packets. In order to concentrate on the convergence speed of the protocols, we used a small database consisting of 100 data objects of size 20K each. Each Deno server periodically initiates a synchronization session (with a given synchronization period of 5.0 secs) by sending a *pull* request to another randomly selected Deno server. Our experimental testbed differs from the real world in that the set of servers is constant, and assumed to be well-known. This distinction should not affect our findings on relative performance, but partial information about other servers can hurt performance.

Each server generated transactions according to a global transaction rate (specified relative to a synchronization period). Each transaction accessed and modified up to five data items. Since our focus is on the performance of the global consistency protocol, we did not model any read-only transactions. Weight is uniformly distributed across servers in all of the experiments. All objects are replicated at all servers. The main parameters and settings used in the experiments are summarized in Table 1.

The results presented in the following graphs are the average of five independent runs of executing 1000 transactions in the system. The contributions of the first 50 transactions are excluded to account to eliminate system *warm-up* effects.

3.9.2 Protocols Evaluated

We now investigate the performance characteristics of our protocols using our prototype. We look at two versions of Deno's protocol, `Deno-weak`, and `Deno-strong`.

Additionally, we investigate two representative epidemic replication schemes from the literature. The first, `primary`, is an epidemic primary-copy scheme that uses a specialized primary server to serialize the updates, while propagating the updates using epidemic flow. This protocol is similar to that used in Bayou [66], but does not include advanced Bayou features such as “dependency checks” and “merge procedures”. In our implementation, we modeled this scheme simply by allocating all the weight at a single server. Note that primary-copy protocols trade availability for a presumed advantage in performance. One of our findings is that this performance advantage is not significant in protocols that use epidemic-style communication.

The second scheme, `write-all`, is a “Read-One, Write-All” (ROWA) [12] protocol, where servers can only commit transactions after ensuring that all other servers are ready to commit. Therefore, a transaction has to be propagated to all the servers before it can be committed. Furthermore, when a server receives conflicting transactions, it has to abort all of those transactions in order to ensure global consistency. A similar epidemic ROWA protocol was proposed by Agrawal *et al.* [4].

3.9.3 Commit Delays

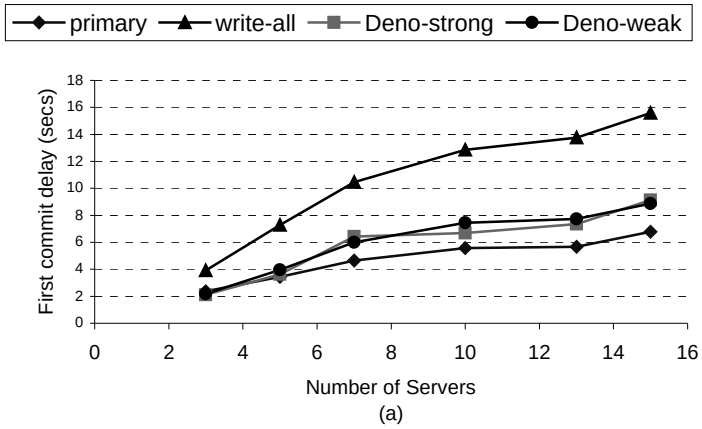

Figure 9: First commit delay vs. system size

We first investigate the update propagation characteristics of different epidemic protocols. We begin by examining the *first commit delay*, which is the traditional commit delay denoting the time between the initiation of a transaction and the time it is *first* committed at a server. Figure 11 shows the first commit delays for the `primary`, `write-all`, and Deno. In `primary`, a transaction commits when the primary server receives the transaction (unless the primary decides to abort the transaction). Therefore, such a scheme clearly (*first*) *commits* a transaction much faster than a uniform voting approach, which requires a transaction to visit at least a majority of servers (assuming a uniform weight distribution). On the other hand `write-all` requires all servers to receive and certify the transaction, and performs poorly compared to either Deno protocol or `primary`.

Unlike traditional synchronous environments where transactions are committed synchronously at every server, commit times typically exhibit wide variability in asynchronous environments. The time at which the *first* server commits a transaction is, thus, not necessarily the quantity that best predicts application performance with epi-

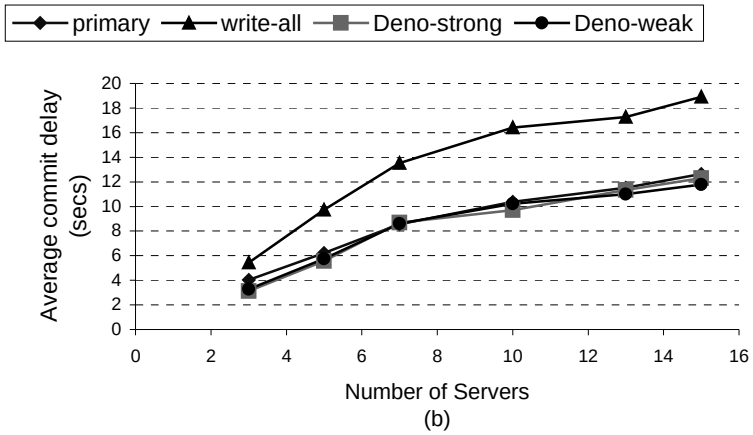

Figure 10: Average commit delay vs. system size

demic information propagation. Since typically all servers have an equal chance of being accessed, an equally useful metric would be the *average commit time*, the time to commit a transaction averaged over all servers. Figure 10 presents the corresponding average commit delays for the `primary`, `write-all`, and Deno protocols. We see that `write-all` still performs significantly worse than the other protocols, and we also observe that the gap between Deno and `primary` does not exist anymore.

We further explore the reason behind the good performance of the Deno protocol by plotting the number of servers that committed the transaction as time progresses (for 15 servers) in Figure 11. Although the *primary* server commits the transaction quickly, this information propagates to other servers relatively slowly. This is because all other servers must learn of the commitment, directly or indirectly, from the primary server. With the `Deno protocols`, on the other hand, distinct servers may either learn the commitment from other servers (as in the case of `primary`), or commit the transaction *independently*. In the presented example, for instance, about seven servers (on the average) committed the transaction independently. The delay between the first and sub-

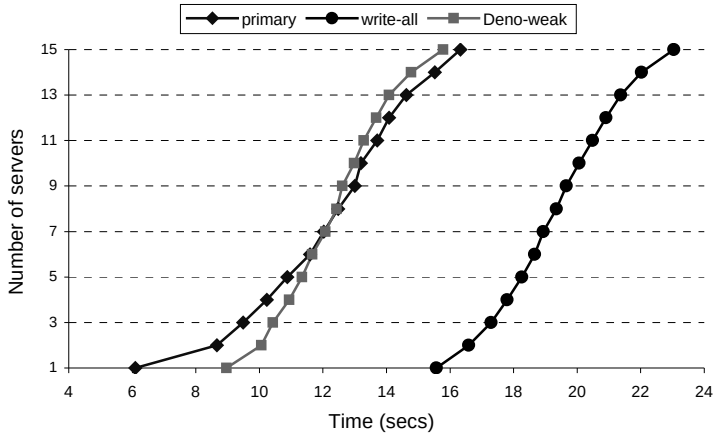

Figure 11: Number of servers that committed the transaction as time progresses (15 servers, TR=0.01, SP=5.0)

sequent commits is thus quite small, as revealed by the high slope of the `Deno-weak` curve in Figure 11. In an environment where updates are propagated asynchronously, average commit delay is as important as the first commit delay because committed transactions are only useful at a server when reflected in the local replicas. One important implication is that the performance penalty of using voting rather than a primary-copy approach is not as large as commonly assumed in the kinds of environments we address.

However, the most interesting results from this series of experiments is that the basic Deno protocol did not appear to have any advantage of the extended version. We had expected the stronger semantics of the extended protocol to exact a performance cost. Instead, the difference between the commit delays of the two with little contention appears to be negligible, and is only an average of 10% with significant contention. The case with contention was where we expected the most degradation in performance, as the requirement of a global ordering effectively increases the number of conflicts. This increase in conflicts, in turn, forces more weight to be inspected before

a winner of a given “election” can be determined. For example, we only need >50% of the weight in order to determine the winner of an election if there are no conflicting transactions, but we may need all of the weight in order to decide between two or more. However, the increase in required *weight* is offset by an increase in *conweight*.

Consider the process of committing a transaction with no contention. Notice of the transaction has to propagate to half of the system servers before it can be committed. With two conflicting transactions that gain votes at the same rate, on the other hand, all of the votes may need to be cast before a winner is identified. However, each transaction can collect server votes independently, with notice of all votes arriving at some intermediate server in approximately the same amount of time as was needed for the single transaction case. Therefore, update contention does not necessarily increase commit delays.

3.9.4 Effects of Contention

The previous subsection focused on the speed of transaction commits protocol performance when there is no update contention. This section studies the effects of transaction generation rate on the overall performance of the system.

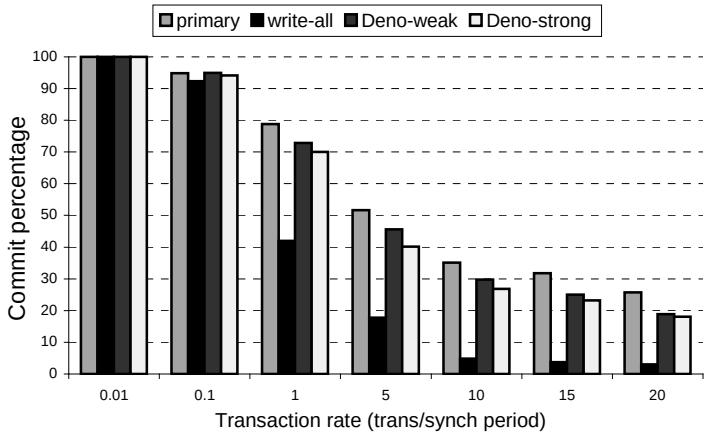

Figure 12: Commit percentages
(15 servers, SP=5.0)

Figure 12 presents the performance results of the protocols *under update contention.* More specifically, the figure shows the *commit percentage* (i.e., the percentage of initiated transactions that are committed) results for different levels of transaction generation rate (for 15 servers) for all protocols. The figure shows that all approaches suffer from the increased transaction rate due to the global update consistency requirement that only one out of a set of conflicting transactions can commit. Under very small transaction rates ([0.0-1.0] trans/synch period), all protocols perform fairly well, achieving commit percentages of around 100%. With increasing transaction rates, however, commit percentages drop for all protocols significantly. We observe the most dramatic fall for `write-all`: at around a transaction rate of 1.0, the commit percentage of `write-all` is less than 50%, whereas the commit percentages of the other protocols are above 70%. Overall, `primary` achieves the best commit percentage, followed closely by the weak and strong versions of Deno. The difference between the two versions of Deno as well as the difference between Deno protocols and `primary` over the whole range shown is small (within absolute 5%). The performance

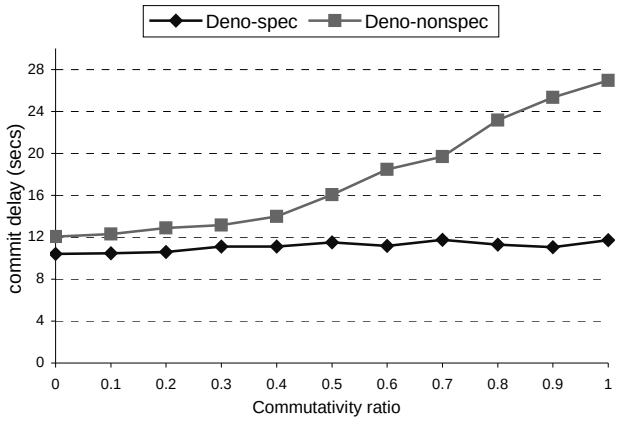

Figure 13: Speculation effects on commit delay
(15 servers, SP=5.0 secs)

of `write-all` is significantly lower than the rest of the protocols. In fact, at (and beyond) a transaction rate of 25.0 (not shown), `write-all` does not commit any transactions. The main reason for this difference is that `write-all` has to abort all conflicting transactions, as it is not equipped with any mechanisms to globally single out a transaction to commit (out of a set of conflicting transactions). The other protocols continue to commit transactions regardless of the transaction rate (not shown).

3.9.5 Effects of Speculation

Figure 13 examines the benefits of speculative update propagation and voting for varying degrees of commutativity by showing the performance of speculative (`Deno-spec`) and non-speculative (`Deno-nonspec`) versions of `Deno-strong`. A commutativity ratio of .25, for example, implies that 25% of transactions made obsolete (in the sense discussed in Section 3.3) run commutativity procedures whose acceptance criteria are satisfied. Somewhat non-intuitively, larger commutativity ratios result in larger commit delays for the non-speculative version of the protocol. The reason is

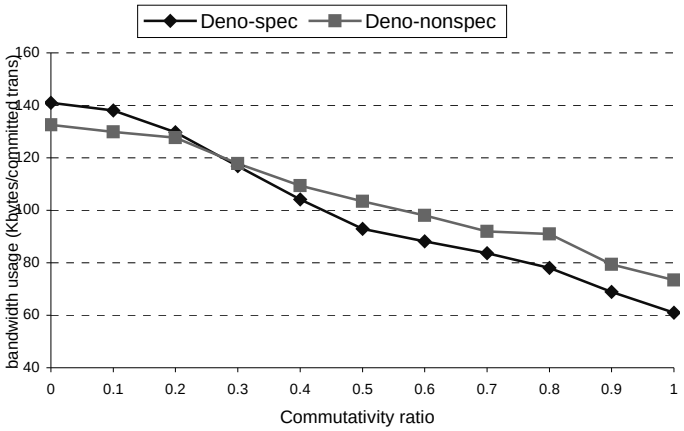

Figure 14: Speculation effects on bandwidth usage
(15 servers, SP=5.0 secs)

that increasing commutativity results in fewer aborted transactions, which in turn increases contention for those transactions that have not been aborted.

By contrast, `Deno-spec`'s commit delay is largely constant across all commutativity ratios. Speculative voting confers a performance advantage of about 15% even with a commutativity ratio of 0.0 (i.e., the default case where no transactions commute). The gap increases with commutativity ratio until `Deno-nonspec`'s commit delay is more that twice `Deno-spec`'s at a ratio of 1.0.

As discussed above, the benefits of speculation come at the expense of propagating more transactions and votes. To this end, we also investigated the relative bandwidth requirements of both approaches. We found out that `Deno-spec` indeed sends about 6% larger messages on the average (which turns out to be about 200 bytes/message more than the non-speculative case on average in our setting). Although the speculative approach uses more bandwidth per message, as demonstrated in the previous experiment, the commit rates for different approaches are quite different and need to be taken into account when comparing the relative bandwidth *utilizations* (e.g., how

much useful work is accomplished per byte transmitted.). Figure 14 shows the amount of information sent across all servers (in KBytes) per committed transaction for `Deno-spec` and `Deno-nonspec`. For low commutativity ratios (up to .10), `Deno-spec` propagates about 4-6% more information per transaction commit. At a commutativity ratio of .20, both approaches propagate almost the same amount of information. Beyond this ratio, the speculative protocol sends less information than the non-speculative version, with the difference increasing as the commutativity increases. At a commutativity ratio of 1.0, `Deno-spec` propagates about 16% less information per committed transaction. In summary, the speculative version not only decreases average commit delays, but also decreases bandwidth requirements per committed transaction.

3.10 Summary

We have presented the design, implementation, and evaluation of Deno, a highly-available object-replication system that supports transactional semantics in mobile and weakly-connected environments. Deno's consistency protocols are based on an asynchronous weighted-voting approach implemented through epidemic information flow. Our voting approach achieves higher availability than primary-copy approaches, and higher availability and performance than ROWA approaches.

Our base protocol ensures weakly-consistent executions where update transactions are serializable and queries always access transactionally-consistent database states. Our extended protocol provides strong consistency and globally serializable executions by providing a unique global commit order on all update transactions. Both protocols allow queries to be executed and committed entirely locally, and without

blocking. Furthermore, neither protocol suffers from local or global deadlocks. We also provided proof outlines for the correctness of all our protocols.

Our detailed performance study revealed several interesting results. First, the presumed performance advantage of the primary-copy approach over a uniform voting approach is not as significant with asynchronous epidemic protocols. The reason is that epidemic voting protocols allow servers to independently arrive at the same conclusions, whereas primary-copy schemes require all commit information to emanate from a single, distinguished server. Second, our extended protocol performs nearly as well as the base protocol, while providing significantly stronger semantics. The result is increased functionality at essentially little cost in performance. Finally, speculative update propagation and voting provides a considerable performance advantage for protocols that use pair-wise communication, and this advantage is magnified when application-specific commutativity information is used to decrease the rate of transaction aborts.

Chapter 4 Decentralized Security Protocols

Asynchronous approaches that provide the most flexibility are peer, decentralized approaches. These decentralized approaches, also called lazy-group [29], can support the "update anytime, anywhere" model, which eliminates restrictions regarding where and how updates are performed, effectively facilitating dis- and weakly-connected operation. With the advent of wireless ad-hoc networking and technologies like Bluetooth, peer-to-peer, decentralized approaches are likely to become increasingly more prevalent in future replicated databases and systems.

Despite all the desirable features that we discussed earlier, however, no such system could be widely deployed in mobile or wide-area environments without ensuring that the infrastructure is secure. Decentralized and asynchronous aspects of these approaches pose unique security challenges, many of which are yet to be addressed. In this section, we present a basic infrastructure security infrastructure for providing strongly-consistent, secure replication support for peer-to-peer, decentralized databases in the context of Deno.

The infrastructure we present addresses both external and internal security threats. The prime *external* threat is of an unauthenticated server attempting to read or modify data. We prevent this through a combination of cryptography-based mechanisms. Our main focus in this thesis is, however, dealing with *internal* threats to security, which is more problematic. Internals threats arise from duly authenticated servers (i.e., insiders) that attempt to cheat by misrepresenting protocol-specific information. As a trivial example, a user of a distributed meeting room scheduler might attempt to falsify votes of other servers in order to ensure that he or she gets a prime reservation. More serious

scenarios could arise in collaborative intranet and Internet applications, such as scheduling and workflow applications. Finally, this work has obvious applications in military scenarios (e.g., consider communication among tanks or mobile command posts). We deal with malicious insiders by using cryptographic techniques, as well as modifications to the update commit criteria. The fundamental idea is to ensure that any protocol-specific information used is correct by using cryptographic techniques when possible, or by explicitly *validating* any such piece of information.

Despite the growing need for security infrastructures for managing replicated data, this topic has yet to be well addressed in the decentralized, asynchronous environments that we target. Prior work mainly investigated security issues in traditional synchronous environments (e.g., [52]), and environments where strong connectivity and atomic reliable multicast primitives are available and replication can be globally coordinated by distinguished master servers (e.g., [16, 22, 25, 61, 62]). Studies that address the restrictions of our target environments [51], on the other hand, ignored consistency issues and made assumptions about where and how updates are generated and initially received. Our work aims to provide flexible security mechanisms that eliminate many restrictions of prior work while allowing for strongly-consistent access to decentralized, replicated data. We note that, although our fault model is comprehensive, it does not handle *fully* Byzantine attacks due to its reliance on digital signatures for authenticating the original source of messages that are forwarded in the system.

In summary, the work presented in this section makes the following key contributions: First, we classify internal attacks and propose a decentralized protocol that is parametric in the degree of tolerance to malicious insiders. This protocol allows serv-

ers to trade off the degree of this tolerance with the performance of update commits. A unique aspect of our protocols is that individual servers can support arbitrary degrees of tolerance without adversely affecting the performance of other servers. This allows each server to set its security level independently based on individual requirements or resources. The protocols we describe support strong-consistency and global serializability. Second, we describe a combination of conventional cryptography-based techniques that can be effectively used to provide protection against external threats in decentralized, asynchronous databases. Third, we evaluate the cost of our security extensions by implementing them on top of the representatives of existing decentralized replication schemes.

The rest of the chapter is organized as follows. Section 4.1 briefly describes relevant security work in distributed replicated databases. Section 4.2 briefly describes how external threats can be handled using a combination of well-known cryptographic techniques. Section 4.3 gives a classification of potential internal security threats and Section 4.4 describes our approaches for handling such internal threats. Finally, Section 4.5 evaluates the performance of our extensions for handling a specific class of internal threats and Section 4.6 concludes the chapter by summarizing our main results.

4.1 Background

Despite the growing need for secure protocols for managing replicated data, this topic is yet to be addressed for the decentralized, asynchronous environments that we target. We now review relevant security work.

Liu et al. [50] addressed the issue of backing out malicious but committed transactions. This work is complementary to the work we present in this thesis, and can be used by a server that incorrectly committed an update by not supporting a sufficient degree of tolerance. Agrawal and El Abbadi [3] used quorums to preserve confidentiality of replicated data despite the disclosure of the contents of a threshold of the repositories. Ray et al. [60] presented a locking-based, advanced secure commit protocol for multi-level secure distributed databases. Malkhi and Reiter [52] investigated Byzantine failures in quorum systems in a synchronous, non-epidemic setting.

Malkhi et al. [51] provided an analytical treatment of epidemic-style update diffusion (i.e., propagation) algorithms that are tolerant of Byzantine faults. This work assumes that less than t replicas fail and each update is initially received by at least t non-malicious replicas. Each non-malicious replica commits an update only it receives the update from t others.

There is also related work in traditional security for group communication systems and protocols. Ensemble [62] addresses only external security threats, whereas Rampant [61] is designed to handle Byzantine attacks. Castro and Liskov [16] described a practical replication algorithm for tolerating Byzantine attacks in asynchronous environments. These protocols are commonly based on primary-copy models to coordinate replica management and require much stronger connectivity and reliable multicast primitives.

Secure election protocols (e.g., [22, 25]) provide voter privacy and rely on a small number of central facilities for counting votes. Unfortunately, these protocols are im-

practical under weak-connectivity as restrictions in connectivity and disconnections make reliance on central authorities untenable.

4.2 External Security Threats

In this section, we describe how a decentralized, peer-to-peer system, such as Deno, can effectively provide protection against external security threats using conventional public-key techniques. We define an *external* security threat as one that is posed by a principal (server) that has not been authenticated into the system. We first discuss authentication, and then integrity and privacy.

4.2.1 Authentication

A principal (server) is authenticated into the system by identifying itself to a distinguished server acting as the certificate authority (CA). We assume a priori that all servers trust the CA, and know the CA's public key. The CA responds with an *access certificate* that specifies the principal's rights in the system. Certificates may provide either *read* or *read/write* permission for a given database, and may contain a timestamp that delimits the certificate's lifetime. Since a certificate is signed by the CA, any server with the CA's public key can verify that the certificate is valid, and certificates can not be forged. Note that we assume a priori that all servers trust the CA, and know the CA's public key.

Access certificates are checked in three situations; (1) a server requesting an initial copy of the DB must present a read certificate; (2) a server performing its periodic *pull* of information from another server must at least provide a read certificate, and (3) servers will not vote for a new transaction unless it is accompanied by a valid read/write certificate from the transaction's creator.

A CA represents a single point of failure in a system that is otherwise completely decentralized. However, this bottleneck only affects *one-time* authentication into the system. The CA is afterwards not needed to arbitrate even between servers that come into contact for the first time. For example, consider three salesmen who meet for the first time on a train and wish to collaborate on a pre-existing document, setting up a local ad hoc network in order to communicate among themselves. The salesmen do not have to have contact with a CA in order to start collaborating. On the other hand, if only one of the salesmen initially has a copy of the data, the others cannot make copies unless they already have certificates, or are currently connected to the CA.

We solve this problem by allowing the CA to issue *ticket-granting tickets* (TGT), analogously to Kerberos [64]. A TGT gives the bearer a limited ability to make and grant new certificates for resources and properties. In our architecture, use of a TGT requires direct confirmation from the user. Note the TGT's can be used to generalize the system to include a hierarchy of CA's. This not only provides load-balancing for access to the CA's, but increases the chances that a CA is available when needed.

We allow certificates to be revoked via the issue of a *certificate revocation list* (CRL) from the primary CA. This presents problems because Deno servers have no notion of simultaneity, unlike secure multicast groups and other analogous systems. In other words, given that a CRL has been issued, when are revoked certificates guaranteed to be denied? We solve this problem by casting the issue of a CRL as just another update transaction. The CRL update competes with other transactions to commit. Once the CRL update has been committed, we can guarantee that no subsequent update will be committed with the aid of a vote authenticated by a revoked certificate. A secon-

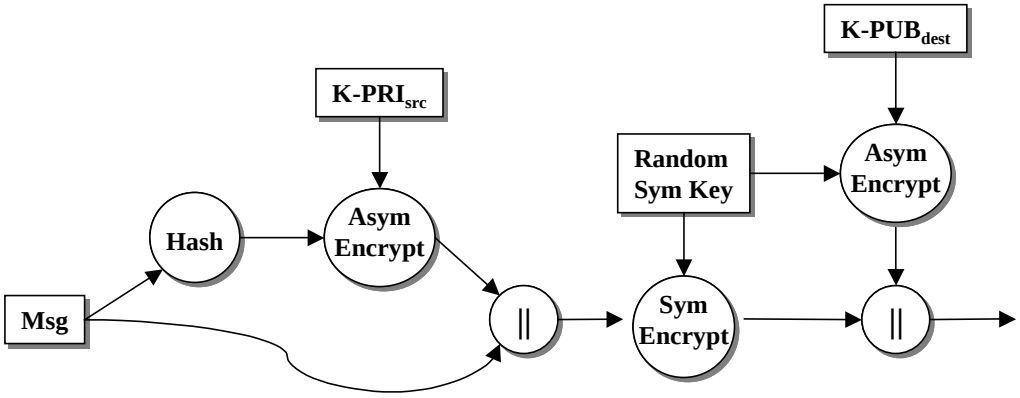

Figure 15: We use symmetric encryption (i.e., Triple-DES) to encrypt the message, and asymmetric encryption (i.e., RSA) to encrypt the symmetric key and sign the message. K-PRIsrc and K-PUBdest are private and public keys of the source and destination of the message, respectively (double bars indicate concatenation).

dary advantage of casting the CRL issue as an update is that it guarantees quick dissemination. Otherwise, knowledge of the CRL might disseminate quite slowly because the CA is not consulted during the normal course of events.

4.2.2 Integrity and Privacy

Figure 15 shows Deno's approach to providing both integrity and privacy guarantees for communicated data. Note that this method is very similar to the method used in PGP. Integrity is provided by appending a message authentication code (MAC) to each message, which in this case is the MD5 hash of the message signed by encrypting with the source's private key. Privacy is provided by encrypting the message and the MAC with a randomly generated, one-time *session key*. The session key is then encrypted with the destination's public key, and the concatenation of the encrypted session key, MAC, and message is sent to the destination.

The use of peer-to-peer one-time session keys allows us to avoid the *key changing* problem incurred by secure multicast trees [62] (note that these peer-to-peer keys need

not be one time; instead they may be cached and re-used later). Secure multicast trees generally use a single session key for the entire group. Any change in group membership requires the session key to be changed. The key must be changed when a new server, s_n, is added to the group because we do not want s_n to be able to read messages that were sent prior to its joining (we assume that s_n might have recorded prior encrypted messages even though it could not read them). Similarly, the key should be changed when s_n leaves the group because we do not want s_n to be able to read messages that are sent after it leaves the group. The use of peer-to-peer session keys, thus, eliminates a similar need in our model.

4.3 Internal Security Threats

We now describe the way that we deal with internal threats. An internal threat is one that results from an authenticated but malicious server. Such malicious insiders misrepresent protocol-specific information, and can cause potentially corrupt objects to propagate throughout the network. Under certain circumstances, even a single malicious insider with arbitrarily small amount of weight can cause different transactions to be committed at different servers. We begin with a discussion of the set of malicious actions a server can undertake, and then discuss our approaches for handling them.

4.3.1 Malicious Actions

Before we classify the actions a malicious intruder can take, we should note that malicious servers can always commit arbitrary transactions to their *local* databases without even advertising the transaction to other servers. Malicious servers can also remain within the protocol framework and issue updates that, if committed, obscure or undo

the effects of other updates. This type of behavior can only be handled in an application-specific manner and is beyond the scope of this work. Under certain circumstances, even a single malicious server can accomplish a denial-of-service attack by refusing to vote its weight. This attack is handled by the Deno's normal *weight revocation* mechanism [17] used to recover from benignly-failed servers.

The goal of this section is to describe the types of damage that malicious servers can inflict on other servers. Malicious insiders can only corrupt the view of other servers by propagating valid but incorrect protocol information. This potentially causes different servers to commit updates inconsistently across the system, which in turn violates any global correctness guarantees and leads to a divergence among the databases at different servers. In our framework, a malicious server can incorrectly report weight values or votes.

Weight Misrepresentation

The problem here is of a server misrepresenting the amount of weight it has available for voting purposes. This is possible since Deno servers can perform *peer-to-peer weight exchanges* to migrate weight allocations towards a target distribution. A peer-to-peer exchange is used by two servers to re-allocate their weight between them. Although this local operation enables light-weight replica creation, retirement, and dynamic weight redistribution [17], and it poses a unique security problem in that it cannot be directly verified by other servers.

We make this operation secure by requiring each weight exchange to be formalized as an update. A weight transfer from s_i to s_j is only considered complete when the corresponding *exchange update* is committed. Note that such exchange updates are com-

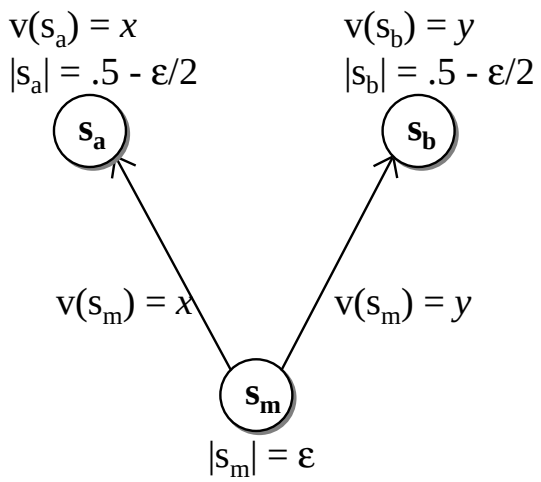

Figure 16: Vote misrepresentation: By telling sa and sb different votes, sm can cause them to commit conflicting updates (|s| is the weight held by s and v(s) is the transaction s votes for).

mutative with respect to all other updates and are generally committed faster than ordinary updates.

Vote Misrepresentation

There are two types of vote misrepresentation:

1. *Misrepresenting non-local votes*: A malicious server s_m misrepresents or forges some other server s_a's vote to a third server s_b. This can happen, for instance, when s_a and s_b are connected through s_m, s_a reports its vote to s_m and s_m forges this vote and reports a different vote for s_a to s_b. This type of malicious behavior is prevented by requiring each server to sign its votes using a suitable digital signature technique. The worst a malicious server can do then is to never report s_a's vote to s_b. Since our symmetric, peer model does not impose any specific con-

nectivity requirements, this behavior can only delay committing of transactions, but cannot affect correctness.

2. *Misrepresenting local votes*: The second vote misrepresentation is more difficult to guard against and can quite easily be used to violate all correctness guarantees. In this case, a server (possibly signs) and illegally votes its own weight more than once for multiple transactions. Consider the example shown in Figure 16. Assume that server s_m is malicious. If s_m tells s_a that it votes for x, and s_b that it votes for y, then both destinations reach the conclusion that their candidates have more than 50% of the vote and can be committed. Furthermore, securely signed votes do not help in this case since s_m can properly sign its own vote for any transaction. In the rest of this section, we investigate approaches to detecting such malicious servers, and develop an algorithm that guarantees correctness at all non-malicious servers.

4.4 Approaches for Handling Internal Threats

We now present a decentralized algorithm that (a) guarantees correctness *even when there are* (*multiple*) *malicious servers*, and (b) allows progress *even when not all votes have been reported.* The idea is to make commit decisions based on votes that are guaranteed to belong to *non*-malicious servers.

4.4.1 Secure Update Commitment

We first distinguish between *validated* and unvalidated votes: formers are votes that are *known* to be correct (i.e., non-malicious), and latter are votes that may or may not be correct (we describe how votes are validated below). Our approach hinges on the following key observation:

1. $\|votes(\{t_i\})\| > 0.5$, or 2. $\|votes(\{t_i\})\| > \|votes(\{t_j\})\| +$ unknown, $t_i, t_j \in C, i \neq j$	non-secure criterion
1. $\|votes(\{t_i\})\| - \|unvalid(\delta, \{t_i\})\| > +0.5$, or 2. $\|votes(\{t_i\})\| - \|unvalid(\delta, \{t_i\})\| > \|votes(\{t_j\})\|$ + unknown, $t_i, t_j \in C, i \neq j$	secure criterion

Table 2 : Commit criteria

"Up to δ *malicious servers can be kept from corrupting the decentralized commitment process if the* δ *largest* unvalidated *votes are not used in any commit decision,"*

where δ is called the *degree of tolerance* to malicious servers ($\delta = 0...n-1$). Consider the following example: if there is a single malicious server, then any single vote may be a duplicate. The server can commit the transaction if the transaction can obtain plurality *without* counting the largest *unvalidated* vote for that transaction. This observation follows since, by definition, (i) validated votes cannot be duplicates and (ii) of the unvalidated votes, at worst the largest unvalidated vote may be a duplicate. Therefore, this worst case duplicate vote cannot be counted towards the commit decision at this server.

In general, *votes*($\{t_i\}$) consists of validated votes, *valid*($\{t_i\}$)**,** and unvalidated votes, *unvalid*($\{t_i\}$). Note that we consider votes cast by the local server to be validated votes. We denote the weight of any vote v in *votes*($\{t_i\}$) by $|v|$. Similarly, we denote the total weight for a set V of votes by $|V|$, e.g., $|votes(\{t_i\})|$ denotes the sum of the weights of all votes cast for $t_i \in T$. Finally, let *unvalid*(δ, T) be the set of δ elements

with the largest weight in *unvalid*(T). If we consider all votes in the base Deno system to be validated, then the base commit criterion for t_i can be stated as in the top row of Table 2, where *unknown* is defined as 1 - |*votes*($\{C\}$)| and C is the set of candidate transactions.

In order to provide resilience against malicious servers, the non-secure commit criterion is modified as in the second row of Table 2. The left hand side of the inequality provides a *lower bound* on the amount of weight that t_i is guaranteed to have by not using the δ largest unvalidated votes cast for t_i. The right hand side of the inequality, as before, provides an *upper bound* on the amount of weight t_j can possibly get. Thus, the amount of weight required to commit t_i must be larger than the total weight for any other transaction t_j even if the largest δ unvalidated votes for t_i are in fact cast by malicious servers, and are thus not valid. If the server knows of no other transactions t_j, but it has not yet seen votes from all other servers, then it simply assumes all unknown votes are cast for some other transaction (analogous to the quantity *unknown* in the base commit criterion). Note that this criterion is equivalent to the base, non-secure commit criterion if we set δ equal to zero (in which case all unvalidated vote sets are null).

In order to validate a vote for transaction t_i from a server s_b, a server s_a must ensure that all other servers in the system have seen the same vote. Thus, server s_a must collect *receipts* of the votes cast by s_b to all other servers. A *receipt* of server s_b's vote from server s_c is a statement of the form "Server s_b votes for transaction t_i", securely signed by server s_c using an appropriate digital signature. Server s_a considers a particular vote valid if and only if it has received receipts for that vote from all other

servers in the system or if the vote is cast by server s_a itself. In order to validate a vote, a server s_a does not need to establish a peer-to-peer connection with all other servers in the system — instead, receipts for votes from any server can be forwarded by any other server in the system. Since strong cryptographic primitives protect the receipts, even malicious servers will not be able to alter the contents of the receipt. Malicious servers may corrupt or discard receipts: corrupt receipts will be detected by the server validating the receipt, while discarded receipts will be treated as any lost message. In the worst case, malicious servers may be able to affect the liveness properties of the algorithm, but once again, we have been able to restore the safety guarantees[3].

When a server detects a malicious vote while performing validation, it marks the corresponding server as malicious, ignores all further votes from that server, and initiates the weight revocation mechanism [17] to cancel the voting rights of the malicious server. If the server already committed an update incorrectly using a malicious vote — which can happen only if the degree of tolerance set by the server is less than the actual number of malicious insiders — the server has to rollback the effects of the update.

4.4.2 Correctness of the Secure Commit Criterion

We now provide a proof sketch for the correctness of the secure commit criterion. The correctness proof for the non-secure commit criterion (i.e., δ=0) can be found in [18].

[3] We stated that we wanted to provide absolute, non-probabilistic, guarantees. Our scheme relies on the integrity of the digital signature used; i.e., our guarantees are only as strong as the underlying digital signature scheme.

Consider n servers $s_1, s_2, ..., s_n$, with weights $c_1, c_2, ..., c_n$. Consider a single server s_i and the case where there is a single malicious server s_m, $i, m = 1...n$ and $i \neq m$. Assume that server s_i commits transaction t_i using the secure commit criterion shown in Table 2. There are two cases: (1) s_i does *not* use c_m towards the votes cast for t_i, and (2) s_i uses c_m towards the votes cast for t_i. In the former case, t_i gathered the plurality of votes by using only the non-malicious votes, so the decision is correct. In this case, the commit criterion is more conservative than required. In case (2), $|votes(\{t_i\})| - c_i$ provides a lower bound on the *valid* votes cast for t_i. This statement follows since $|votes(\{t_i\})| - |unvalid(1, \{t_i\})| \leq |votes(\{t_i\})| - c_i$ as $c_i \geq |unvalid(1, \{t_i\})|$. The commit criterion in this case is conservative if $c_i \leq |unvalid(1, \{t_i\})|$. Therefore in each case s_i uses only the weights that are cast by non-malicious servers towards committing t_i. A straightforward induction on the number of malicious servers concludes the proof.

4.4.3 Examples and Discussion

In this section, we illustrate, via a set of examples, some of the more subtle properties of the secure plurality algorithm. We begin with a simple example of applying the secure protocol to the three server case shown in Figure 16. We had shown earlier that if server s_m is malicious, in the base protocol, under appropriate circumstances, it could cause the committed views of servers s_a and s_b to diverge arbitrarily far *even if it held arbitrarily small amount of weight in the system.* Now we show that even if server s_m is malicious and holds arbitrarily *large* amounts of weight in the system, it cannot cause a *single* incorrect commit at either servers s_a or s_b, as long as servers s_a and s_b operate under the assumption that there are malicious servers in the system. Assume s_m holds an arbitrary amount (say c_m) of weight. Once again, assume the rest of the

weight is distributed equally between servers s_a and s_b (the analysis for the other cases are analogous and is omitted for brevity).

Consider the scenario when both servers s_a and s_b are trying to commit different transactions t_1 and t_2, respectively. Assume server s_m tells server s_a that it votes for transaction t_1: this would be enough under the base commit criterion for server s_a to commit. But under the new commit criterion, server s_a considers its local votes as validated, but the quantity |**unvalid(1,{t_1})**| is non-zero since there is only one other vote and it is unvalidated. The commit criterion is not satisfied and server s_a must delay committing its transaction till it receives a receipt for server s_m's vote from server s_c. Transactions can, therefore, be committed if and only if server s_m votes consistently and correctly.

In the following examples, assume the secure commit criterion is used with the assumption that there is at most one malicious server in the system (i.e., δ=1). The first example shows that even under contention (i.e. when there is more than a single transaction competing for commitment), the commit criterion does not necessarily require any votes to be validated to commit a transaction.

Example 6: Assume five servers, s_1, s_2, …, s_5, in the system, each holding equal (i.e., 0.2) weight, and the following votes at s_1: V_1={(s_1, t_1), (s_2, t_1), (s_3, t_1), (s_4, t_1), (s_5, $\boldsymbol{t_2}$)}. In terms of the new commit criterion: |*votes*({t_1})|=0.8, |*unvalid*(1, {t_1})|=0.2, |*votes*({t_2})|=0.2, and *unknown*=0.0. In this case, s_1 can commit t_1 without validating a single vote!

The second example shows that even when validation of at least one vote is necessary, it is not necessarily the case that all votes have to be validated.

Example 7: Assume servers s_1, s_2, ..., s_4 have weights 0.2, **0.4**, 0.2, and 0.5, respectively. Votes at s_1 are: V_1={(s_1, t_1),(s_2, t_1),(s_3, t_1), (s_4, $\boldsymbol{t_2}$)}. In terms of the new commit criterion: |*votes*($\{t_1\}$)|=0.8, |*unvalid*(1,$\{t_1\}$)|=0.4, |*votes*($\{t_2\}$)|=0.5, and *unknown*=0.0. Server s_1 can not commit t_1 because: |*votes*($\{t_1\}$| - |*unvalid*(1, $\{t_1\}$)|= 0.4, whereas |*votes*(t_2)| + *unknown* is 0.5. Validating s_3's vote would have no immediate utility. However, if s_2's vote were validated instead, the commit could take place. As can be seen from the secure commit criterion in Table 2, validating a vote can only have an immediate effect on a commit decision if it affects *unvalid*(1, $\{t_1\}$). Validating s_2's vote has such an effect; validating s_3's does not.

4.5 Performance Evaluation

4.5.1 Experimental Environment and Performance Metrics

Using the Deno prototype, we now evaluate the cost of the proposed security mechanisms for Deno's decentralized voting protocol and a ROWA-type decentralized protocol (described below). We performed the experiments using the Deno testbed we described in Section 3.9.

The results presented in the following plots are the average of at least five independent runs of executing 1000 transactions in the system. The contributions of the first 50 transactions are excluded to account to eliminate system warm-up effects. The bandwidth requirements for transactional and consistency data were negligible compared to that required for propagating updated values, so we do not consider this question further.

For context, we also show the performance of a second decentralized scheme, `write-all`, which is an epidemic "Read-One, Write-All" (ROWA) [12] protocol

modeling the other peer-to-peer decentralized transactional protocol in the literature. This protocol commits transactions after ensuring that all other servers are ready to commit. Therefore, a transaction has to be propagated to all the servers before it can be committed. In terms of the voting terminology, `write-all` commits a transaction when the transaction gathers all the votes in the system. A similar ROWA-type epidemic protocol was proposed by Agrawal *et al.* [4]. We also implemented a secure version of `write-all`, which also uses a straightforward adaptation of the vote validation technique described in Section 4.4.

The primary performance metric we consider is *average commit delay*, which denotes the time between the initiation of a transaction and average of the times at which it is committed by individual servers in the system. As a measure of scalability, we report the change in commit delay as the number of servers in the system change. We also use *commit percentage*, the percentage of transaction initiated transactions that are committed, when we explore the effects of update contention. In each case, we consider the efficacy of our algorithms by varying the degree of tolerance to malicious servers and where applicable, compare our results to `write-all`.

Before presenting our results, we would like to note that individually signing a large number of votes and receipts using a conventional digital signature scheme such as RSA can be computationally expensive: the execution times for signing a 16 byte hash code using 512 bit keys using the RSAref library from RSA Security Inc. (see www.rsa.com) is approximately 330 msecs in our environment. Instead, any set of votes and receipts can be signed together as a single message. Furthermore, probabil-

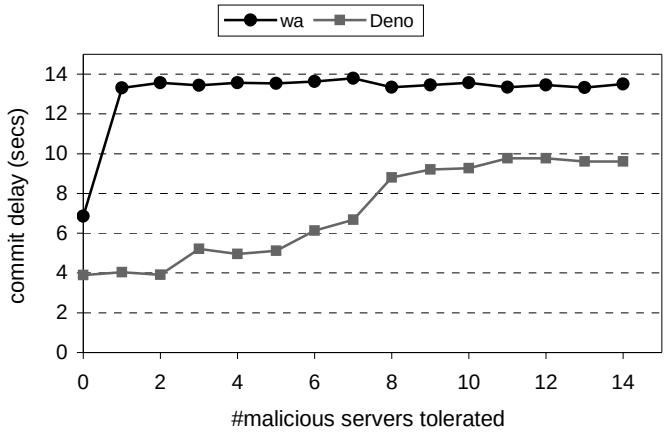

Figure 17: Commit delays vs. degree of tolerance
n=15, TR=0.01, SP=2.0

istic techniques that trade off signature quality to computational overhead can also be used to decrease this computational cost by several orders of magnitude (e.g., [15]).

4.5.2 Commit Delays vs. Degree of Tolerance to Malicious Servers

Figure 17 shows the average commit delays for small transaction generation rates (i.e., almost no update contention), for Deno and `write-all`, with varying degrees of tolerance to malicious servers, i.e., δ. On the x-axis we vary δ from 0 (non-secure system) to n-1 (max-security system). The curve for Deno follows an *S*-shape; initially increasing gradually with increasing δ, making a significant jump in the vicinity of $\delta = n/2$, and then essentially staying flat afterwards. As long as δ is smaller than $n/2$, servers do not need to use validated votes to commit an update; it simply is enough to gather sufficient unvalidated votes. For instance, assuming no update contention and n set to 15, a single update can be committed with 11, 12, 13, 14 unvalidated votes when δ is 3, 4, and 5, respectively. However, when δ is more than half the servers, it is not possible to commit updates without the use of validated votes. Vote validation is a

relatively costly operation, as it involves obtaining receipts from the other servers in the system. This explains the sudden increase in commit delays as δ exceeds $n/2$. After this point, commit delays continue to increase as more validated votes are required for commit. At the point where half of the all votes are validated, updates can immediately commit, which is the reason why commit delays for Deno essentially stays constant for relatively large δ values.

The figure also depicts commit delays for non-secure `write-all` ($\delta = 0$), and secure write-all ($\delta > 0$). Since secure `write-all` requires all votes to be validated for commit, it cannot support intermediate degrees of tolerance as Deno. We observe that for all degrees of tolerance, Deno commits updates significantly faster than `write-all`, reducing the commit delays of `write-all` by 40% and 30% for non-secure ($\delta = 0$) and maximum security cases ($\delta = n-1$), respectively. The most dramatic improvement, 60%, occurs when $0 < \delta < n/2$, since, in this region, Deno commits updates *without* validating any votes, whereas `write-all` has to validate *all* votes before committing an update.

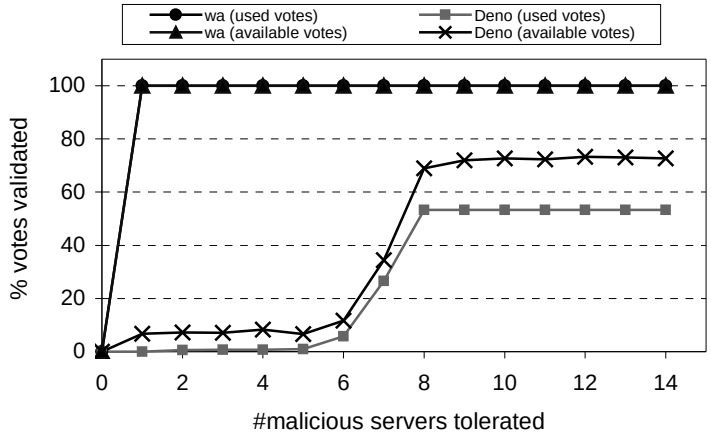

Figure 18: Percentage of validations required/available for commit vs. degree of tolerance, n=15, TR=0.01, SP=2.0

Figure 18 provides more insight by plotting the percentage of validated votes used and those that are available at commit time at each server, averaged over all commits across all servers. As we expected, no validated votes are used at commit by Deno when $\delta < n/2$. In this region, validated votes available at commit time at each server are non-zero, because each server considers its own vote as validated by default. Notice that Deno requires at most 50% of the votes to be validated for supporting any degree of tolerance. On the other hand, `write-all` requires 100% votes to be validated to tolerate any number of malicious servers, thereby incurring relatively large commit delays.

4.5.3 Performance Implications of Supporting Non-Uniform Degrees of Tolerance

We now investigate the performance impact of using different degrees of tolerance at different servers. We expect that the commit performance of each server be independent from the degrees of tolerance supported by others, since each Deno server makes

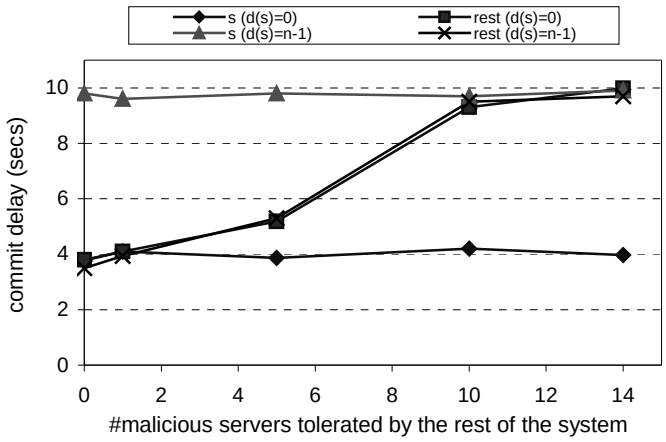

Figure 19: Supporting non-uniform degrees of tolerance at individual servers, n=15, TR=1.0, SP=2.0

all commit decisions entirely *independently* and using only *local* information. To demonstrate the validity of this premise, we conducted an experiment where we let a single server, *s*, use a degree of tolerance, $\delta(s)$, different from that used by the rest of the servers, $\delta(rest)$.

Figure 19 presents commit delay results for *s* and the rest of servers (averaged) for the cases where $\delta(s) = 0$ and $\delta(s) = n\text{-}1$, as we vary $\delta(rest)$ –– note that *d* refers to δ in the figures. When we consider the commit delay curve for *s* when $k(s)=0$, we observe that the curve remains essentially flat regardless of the degree of tolerance used by the other servers. Even when $\delta(rest)$ is set to n-1, the performance of *s* is not affected at all. The same observation holds for the case where $\delta(s) = n\text{-}1$. It is evident that the commit performance of a server is not affected by the performance of the rest of the system. The commit delay curves for the rest of the system for $\delta(s) = 0$ and $\delta(s) = n\text{-}1$ illustrate the complementary case. We observe that the two curves are essentially identical, revealing that the performance of the system as a whole is not affected

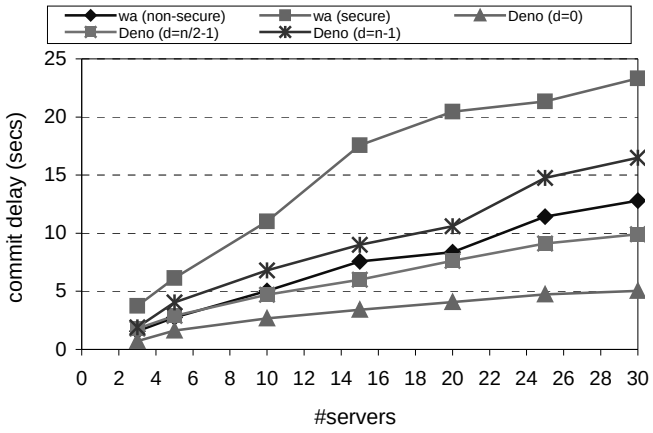

Figure 20: Scalability, TR=0.01, SP=2.0

by the degree of tolerance set by *k*. It is therefore clear that the degree of tolerance used by a server does not adversely affect the performance of other servers and vice versa.

4.5.4 Scalability

Figure 20 shows commit delays for Deno with various degrees of tolerance for malicious servers, and for secure and non-secure versions of `write-all`, as the number of servers is varied from 3 to 30. As expected, the commit delays increase as the system size increases for both Deno and write-all. Non-secure Deno (i.e., $\delta = 0$) demonstrates the best scalability, followed by Deno with δ = n/2-1. Recall that at this tolerance degree, commits do not require any validated votes, eliminating the need to contact every other server in the system. This also explains why Deno with δ = n/2-1 performs better than even the non-secure write-all, which needs to contact all servers to commit an update. We also observe that Deno with max-security (i.e., δ = n-1) commits updates faster than secure `write-all` (since Deno needs to validate at most half the votes), and the difference between the two approaches increases with increasing system size.

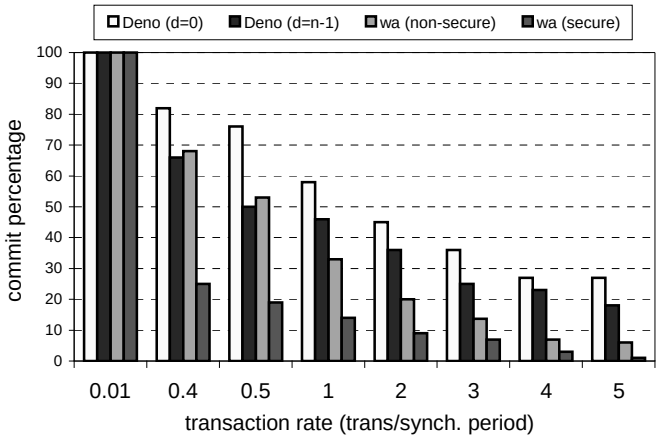

Figure 21: Update contention effects, n=15, SP=2.0

4.5.5 Update Contention Effects

We now investigate the effects of update contention on Deno and `write-all`. Figure 21 plots commit percentage results for varying transaction generation rates. The figure shows that all approaches suffer from the increased transaction rate due to the global update consistency requirement that at most one out of a set of conflicting transactions can commit (the transactions that are aborted can be restarted depending on application semantics). Under very small transaction rates, $TR \in [0.0\text{-}0.01]$, all protocols perform fairly well, committing all updates. With increasing transaction rates, however, commit percentages drop for all protocols significantly. We observe the most dramatic fall for secure `write-all`: at a transaction rate of 0.4, the commit percentage of secure `write-all` is ~25%, whereas the commit percentages of the other protocols are all above 65%. Notice that beyond a transaction rate of 0.5, *max-security* Deno has a higher commit percentage than even the *non-secure* `write-all`. The reason for this interesting result lies in the fundamental difference in the way Deno and `write-all` treat conflicting updates. Deno's voting algorithm can globally pick a

single update out of a set of conflicting updates to commit. The `write-all` protocol clearly lacks such a mechanism and thus has to abort all conflicting updates. Due to this behavior, beyond some update contention, the `write-all` approaches, both secure and non-secure, are not able to commit any updates (beyond 6 and 10 transactions/synch period, respectively). On the other hand, Deno approaches continue to make progress and commit updates regardless of the update generation rate (not shown).

4.6 Summary

Decentralized, asynchronous approaches to replicated data management, while being well-suited for facilitating dis- and weakly-connected operation, also raise unique security challenges not present in their centralized, synchronous counterparts. We presented a complete infrastructure for protecting such highly-available, decentralized databases against malicious attacks, while providing strongly-consistent access to replicated data. We first addressed external attacks and described how to effectively provide authentication, integrity, and privacy using a proper combination of well-known cryptographic techniques. We then classified and addressed potential internal attacks, which cannot solely be handled using cryptographic techniques; requiring modifications to the update commit criterion and explicit validation of protocol information. We proposed a flexible, parameterized protocol that allows servers to set arbitrary degrees of tolerance to malicious insiders.

We evaluated the cost of our security protocols using the Deno prototype replicated system. The experimental results revealed that: (1) protecting against internal threats comes at a cost, but the marginal cost for protecting against larger cliques of malicious

insiders is generally low; (2) our decentralized protocol performs, scales, and handles update contention significantly better than a ROWA-based decentralized protocol; (3) our approach allows servers to trade off performance and the degree of tolerance to malicious insiders, and; (4) individual servers can support different degrees of tolerance without adversely impacting the performance of other servers, allowing servers to set arbitrary degrees of tolerance based on their individual requirements and resources.

Chapter 5 Design and Evaluation of Token Redistribution Strategies for Wide-Area Commodity Distribution

5.1 Introduction

The proliferation of e-commerce and widespread access to the WWW have enabled a new set of applications that allow globally distributed purchasing of commodities and merchandise such as books, CDs, travel tickets, etc., over the Internet. Companies, such as Amazon.com, Expedia, etc., that support these types of applications are drastically increasing in number and scale. As these applications become more popular, centralized implementations will fall short of meeting their latency, scalability, and availability demands, thereby requiring distributed solutions. Conventional distributed implementations, however, are also not viable for these applications: they inherently require tight global synchronization, and, thus, break down in environments where the nature of the communications medium is failure-prone and unpredictable.

More effective distributed solutions can be realized by exploiting two important characteristics of these applications: First, they involve a set of commodity types with a limited inventory (e.g., the latest CD of Santana, economy class tickets for a particular flight, etc.). Second, the operations of interest on these items typically involve incremental updates (e.g., buying two economy tickets for a flight). It is, therefore, possible to achieve distribution by using *tokens* to represent the instances of commodities for sale. Previous work (e.g., [28, 45, 63]) exploited the notion of tokens to enable high-volume transaction processing for distributed resource allocation applications.

There are two fundamentally different approaches for distributing tokens — *replication* and *partitioning*. Token replication typically requires expensive distributed synchronization protocols to provide data consistency, and is subject to both high latency and blocking in case of network partitions or long delays in communications between groups of sites (which are indistinguishable from network partitions). Token partitioning allows many transactions to execute locally without any global synchronization, which results in low latency and immunity against network partitions. The effectiveness of token partitioning, however, relies on *token redistribution* techniques that allow dynamic migration of tokens to the servers where they are needed.

In this chapter, we examine the Data-Value Partitioning (DVP) [63], a decentralized approach to token-based commodity distribution. We describe pair-wise DVP strategies that vary in the way they redistribute tokens across the servers of the system. Using a detailed simulation model and real Internet message traces, we investigate the performance of our DVP redistribution strategies by comparing them against each other and a previously proposed scheme, Generalized Site Escrow (GSE) [45], which is based on replication and escrow transactions [55]. GSE generalizes previous escrow algorithms for replicated databases, providing higher server autonomy and throughput.

The remainder of the chapter is organized as follows. In Section 5.2, we briefly describe the system model and the DVP algorithm. In Section 5.3, we propose several token redistribution strategies for DVP. We discuss the GSE approach in Section 5.4. We describe the experimental environment and methodology in Section 5.5 and present our experimental results in Section 5.6. Finally, we conclude in Section 5.7.

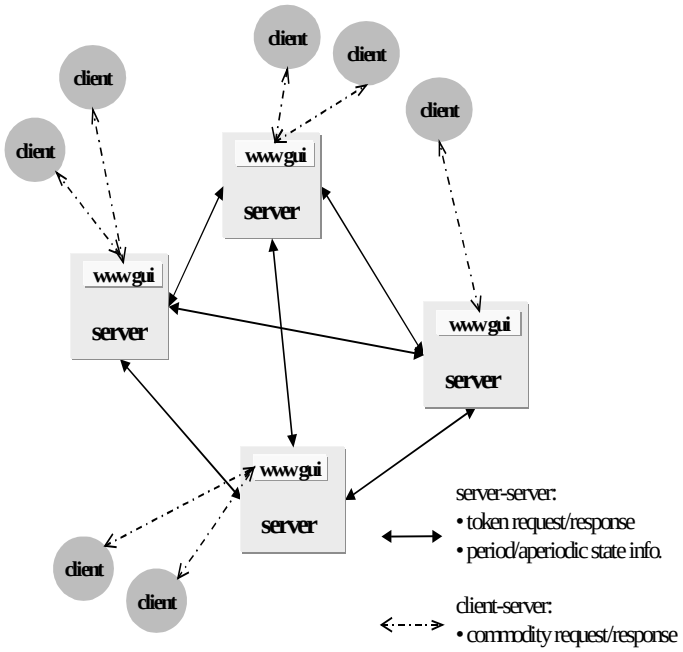

Figure 22: System model

5.2 Overview of Token Partitioning

In this section, we first define our system model. We then briefly give an overview of the basic DVP algorithm [63]. For brevity, we focus only on the performance-related issues, leaving aside many significant properties such as recovery and conweight control. Details of the DVP approach can be found in [63].

5.2.1 System Model

Figure 22 shows our reference system model, which consists of a set of servers and clients that communicate via message exchange over a wide-area network. We assume, for simplicity of exposition, that the system stores a single commodity with a *limited* number of *indistinguishable* instances and we represent each such instance with a single token.

Through a web-based interface, clients submit transactions that *allocate* tokens or *return* (i.e., deallocate) tokens that have previously been allocated. Therefore, the types of transactions that we model involve incremental updates to a data item, *avail*, denoting the number of tokens globally available in the system. We also assume, without loss of generality, that there is a lower-bound constraint on *avail*: the system must ensure that *avail* does not become negative at any time (e.g., tickets for a particular show should not be oversold). Therefore, *all* token return transactions can potentially commit (as they cannot violate the lower-bound constraint), whereas only *some* of the token allocation transactions can commit.

5.2.2 The DVP Approach to Token Partitioning

DVP [63] is a non-traditional approach for representing and distributing data. It essentially applies to data items that can be *partitioned* into smaller pieces such that the pieces can also be regarded as instances of the original data item. The same operators that apply to the original item should also apply to the pieces (e.g., increment, decrement, set to zero, etc.).

The basic idea underlying DVP is to split up the values of database items and store each of the constituent values as tokens at different servers. Transactions are then executed locally at each server using the tokens locally available at that server. In the event that the number of tokens locally available is insufficient to execute the transaction, the server makes requests to other servers only to *borrow* some their tokens. If responses from other servers fail to arrive for any reason within a specified timeout period, the transaction is aborted. Tokens are locked before being accessed, however, only at the server where they reside, i.e., no lock requests are made to other servers.

We refer to the number of tokens available at s_i as tok_i, and the number of tokens required to execute transaction t as $req(t)$, which is a negative value if t is a return transaction.

DVP is a fully *decentralized* scheme that does not require global synchronization. Due to its non-blocking behavior, it is immune to network partitions, and is thus particularly well-suited for environments with unpredictable and failure-prone communications (e.g., many servers on the Internet) due to the high server autonomy it enables.

5.3 Token Redistribution Strategies

Token partitioning enables servers to execute transactions locally as long as they have sufficient tokens. When a server cannot execute a transaction locally due to insufficient number of tokens, it must be able to locate tokens available at other servers and acquire them in a timely fashion in order to continue transaction execution. The performance of DVP relies upon how effectively this *token redistribution* among servers is accomplished. The main questions that a token redistribution strategy needs to answer are:

- when to request tokens,
- which servers to request tokens from, and
- how many tokens to request.

It is possible to construct a cost function with a set of constraints and solve it to find the *optimal* token redistribution. Unfortunately, not only it is difficult to construct a realistic cost function due to the dynamic, distributed nature of the system, and the fact that tokens are perishable resources (i.e., they cease to exist after being used), but also it is long known that even simpler formulations of the problem are NP-hard [24, 28].

Since optimal solutions are impractical, we focus on heuristics-based solutions. In particular, we avoid global strategies that require tight synchronization, and concentrate only on decentralized, pair-wise strategies that make progress using only *pair-wise* synchronization. Note that previous work on DVP [63] focused on the basic features of partitioning and ignored token redistribution issues.

In the rest of the section, we describe several token redistribution strategies for the basic DVP algorithm (Figure 23). We refer to the number of tokens requested by s_b from s_l as $req(s_b, s_l)$, the number of tokens returned by s_l to s_b as $resp(s_l, s_b)$.

5.3.1 Random Redistribution

We begin by describing our baseline strategy, `random`. In this strategy, the borrower contacts a lender server, which the borrower picks *randomly*, and requests the exact number of tokens needed:

$$req(s_b, s_l) = req(t) - tok_b$$

If s_b does not receive the entire amount it needs, it then randomly chooses another lender server and requests the remaining amount. The lender computes the return amount as:

$$resp(s_l, s_b) = \begin{cases} req(s_b, s_l), & if\ req(s_b, s_l) \le tok_l; \\ tok_l, & otherwise. \end{cases}$$

The messages exchanged among servers are minimal, and only include token requests and responses.

5.3.2 Token Count-based Redistribution

Borrower server s_b:

1. Lock tok_b.
2. If $tok_b < req(t)$
   borrow_tokens().
   /* borrow_tokens() gathers tokens from other sites and updates tok_b as it receives tokens. */

Wait until $tok_b \geq req(t)$ or timeout.

3. If timeout
4. Abort; unlock tok_b; and exit.
5. Set $tok_b = tok_b - req(t)$.
6. Commit; and unlock tok_b.

Lender server s_l:

1. Lock tok_l.
2. Calculate the number of tokens, $resp(s_l, s_b)$, $0 \leq resp(s_l, s_b) \leq tok_l$, to be lent to s_b;
   Send $resp(s_l, s_b)$ tokens to s_b.
   Set $tok_l = tok_l - resp(s_l, s_b)$;
3. Unlock tok_l.

Figure 23: The Generic DVP algorithm

The `random` strategy makes *blind* redistribution decisions, since it does not maintain or utilize any information about the states of other servers. The `count-based` strategy attempts to make more intelligent decisions by incorporating knowledge of the *token counts* at other servers into this decision process.

The state maintained by a server s_i is basically a *token table* (*tt*) that stores an estimate of the number of tokens available at all servers; i.e., $tt[j]=(tok_j^i, tstamp_j^i)$, where tok_j^i is the token count at s_j at logical time $tstamp_j^i$, j=1…n, and n is the number of servers.

The `count-based` strategy makes more intelligent redistribution decisions at the expense of maintaining and disseminating token count information. Servers disseminate token count information using *piggybacking* and *broadcasting*. Each server, when

sending a message to another server, also incorporates its token table into the message. In addition to this piggybacking, servers broadcast their states to other servers at the following critical points:

1. when the number of tokens available at a server becomes less than a certain threshold value — so that servers with fewer tokens can be differentiated from the ones with many more tokens;
2. when a server runs out of tokens — so that other servers do not make token requests to this server anymore, and;
3. when tokens are returned at a server that previously had run out of tokens — so that other servers may resume requesting tokens from this server.

A server s_i updates its token table when s_i commits a transaction t:

$$tok_i^i = tok_i^i - req(t) \text{ and } tstamp_i^i = tstamp_i^i + 1$$

and, when s_i receives a token table from server s_k, $k \neq i$:

$$tok_j^i = tok_j^k \text{ and } tstamp_j^i = tstamp_j^k, \quad if\ tstamp_j^k > tstamp_j^i$$

$\forall j = 1...n,\ j \neq i$.

A borrower server, s_b, consults its state table when choosing a lender server. It (rank) orders the servers according to their token counts. The lender servers are then chosen based on their ranks: at each step, a *minimal* set of lenders, L, that together have a sufficient number of tokens is chosen as lenders. Formally, $L \in S - \{s_b\}$ is chosen such that:

$$\sum_{l \in L} tok_l^b \geq req(t) - tok_b \text{ , and}$$

$$\neg \exists L' \; s.t. \; L' \subseteq S - \{s_b\}, \sum_{l \in L'} tok_l^b \geq req(t) - tok_b, \; and \; |L'| < |L|$$

where S is the set of all servers (see Section 5.6.2 for the other lender selection schemes). The borrower then contacts the lenders in *parallel*, without waiting for replies. The borrower makes a pessimistic assumption about the availability of tokens at lender servers and requests $req(t)$-tok_b tokens from each lender. If the estimated token count of a server is zero, then that server is not contacted at all. The redistribution at a lender proceeds similar to that in the basic case.

5.3.3 Token Demand-based Redistribution

In the previous strategies, token redistribution occurs as the result of a token request, which is initiated only when the borrower has insufficient tokens to execute a transaction successfully. The `demand-based` strategy, on the other hand, continually redistributes tokens across servers based on the token request rates at each server. Such a *demand-based* redistribution is likely to be beneficial especially when there is a skew in server workloads.

Each server maintains a simple token request rate value, rr_j, for each server s_j, j=1…n. This value indicates the number of tokens requested from a particular server during a certain period of time. Each server updates its own request rate value and disseminate it along with its token table using piggybacking and broadcasting as described before.

The `demand-based` strategy employs two key techniques that utilize request rate values of the servers:

1. *Token sharing* refers to the redistribution of tokens based on the token request rates as observed by the involved servers. The borrower server s_b sends its token request rate, rr_b, along with its token request. The lender server s_l computes the number of tokens that it should share with s_b as:

$$share(s_l, s_b) = \left\lfloor c \cdot tok_l \cdot \frac{rr_b}{rr_b + rr_l} \right\rfloor$$

aiming to achieve a balanced token redistribution according to relative request rates (where c is the sharing constant). Server s_l then returns:

$$resp(s_l, s_b) = \begin{cases} share(s_l, s_b), & if\ share(s_l, s_b) \geq req(s_b, s_l); \\ req(s_l, s_b), & if\ share(s_l, s_b) < req(s_b, s_l)\ and \\ & \quad tok_l \geq req(s_b, s_l); \\ tok_l, & otherwise. \end{cases}$$

2. *Token* prefetching refers to the periodic, request rate-based redistribution of tokens in the background. Prefetching can potentially eliminate the need to search for tokens *as part of* transaction execution, thereby reducing response time and increasing availability. Periodically, each server contacts every other server in some prefixed order (in the background). When s_i contacts s_j, the tokens available at s_i and s_j are redistributed among the two servers based on their relative request rates as follows:

$$tok_i' = \left\lfloor (tok_i + tok_j) \cdot \frac{rr_i}{rr_i + rr_j} \right\rfloor$$

$$tok_j' = (tok_i + tok_j) - tok_i'$$

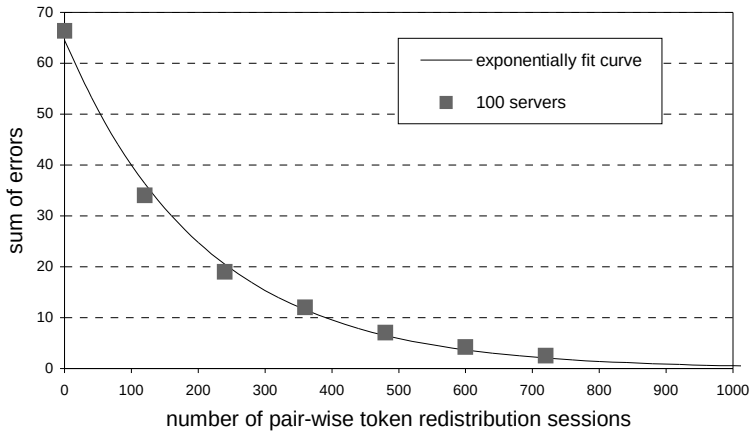

Figure 24: Incrementally converging to global target token distributions using pair-wise token exchanges

where tok_i' and tok_j' are the respective token counts at s_i and s_j *after* redistribution.

Even though the token redistribution mechanism we described operates in a pair-wise fashion and uses *only* the information available locally at the two servers in contact (i.e., tok_i, tok_j, rr_i, and rr_j), it is quite robust in that it *incrementally* migrates the existing token distribution in the system to match the relative request rate distribution. In fact, we experimentally showed that, using this pair-wise mechanism, an existing token distribution converges to any desired global distribution *exponentially* fast.

Figure 24 demonstrates this exponential convergence by plotting the number of pair-wise token exchanges (between randomly selected servers) versus the sum of errors between the global target token distribution and the existing token distribution (averaged over 1000 randomly selected target and initial token distributions for a system with 100 servers).

5.3.4 Hybrid Redistribution

Our preliminary experiments revealed that, as the number of tokens in the system decreases, it becomes increasingly harder to locate and gather the necessary number of tokens in a distributed fashion. Borrower servers typically make several unsuccessful attempts until they are able to gather the tokens they need. Even though there might be sufficient tokens globally available in the system, identifying the servers that have the tokens can be very costly, especially if the system consists of many servers. In such cases, transaction execution costs can become so high that the performance benefits of using a distributed system may be overshadowed.

The `primary` strategy attempts to add the benefits of using a centralized model for situations where only a small number of tokens remain in the system. The system operates in regular, decentralized mode (using the `count-based` strategy) as long as the number of tokens available is above a fixed threshold value, but switches to a *centralized* mode of operation when the total number of tokens drops below this threshold. Each commodity is assigned a primary server that is responsible for satisfying all the requests involving the tokens of that commodity when the system operates in the centralized mode. Each server continuously observes the number of available tokens for the commodities for which it serves as the primary.

If the number of tokens drops below a particular *callback threshold value*, the corresponding primary initiates the switch to the centralized mode by broadcasting *callback* messages. Each server, having received a callback message for a commodity, sends all the tokens of that commodity to the primary. After the callback, the primary executes all requests involving that commodity locally. When a *non*-primary receives a token

request after a callback, it simply forwards the request to the primary, which executes the request and returns the result back to the client. An alternative model, which we do not discuss here, assumes the existence of a set of *forwarding* agents, such as cluster DNSs that translate logical names into the IP-addresses of one of the servers [21], and enables client requests to be submitted directly to the corresponding primary.

If the number of tokens later increases above the callback threshold, the system switches back to its decentralized operation: the primary redistributes the tokens it has to other servers (uniformly or depending on its knowledge of the request rates at servers).

5.4 Algorithms for Token Replication

We now briefly describe the *Generalized Site Escrow* (GSE) scheme [45], an efficient replication scheme based on escrow transactions [55], as the representative replication-based approach to distributed token maintenance.

In GSE, the number of tokens available in the system, referred to as *avail*, is *replicated* at all servers. The escrow quantity at s_i, which represents the number of tokens that s_i can dispense without contacting other servers, is computed as:

$$es_i = \frac{avail_i}{n}$$

where $avail_i$ is s_i's view of *avail*, and n is the number of servers. Each server periodically broadcasts the token allocations it performed to limit the extent to which views of *avail* is out-of-date. The escrow quantity as each server, therefore, dynamically decreases as tokens are allocated. Each server estimates the escrow quantities at other

servers, which are then used to replenish its escrow quantity and allocate tokens without contacting other servers, if possible.

In order to implement this scheme, GSE relies on: (1) *gossip messages*, which are periodic background messages that include the token allocations known by the sender server, in order to limit the global token allocations unknown to a server, and; (2) *quorum locking* to limit the number of token allocations that can be performed concurrently at the quorum servers.

A server s_i can perform token allocations as long as es_i, which s_i updates dynamically it allocates tokens and receives gossip messages, is large enough. In case es_i is not sufficient, s_i needs to contact other servers and form a synchronous quorum of servers such that the combined escrow quantities of the quorum servers are enough to execute the transaction. Forming a quorum of servers involves *remotely* locking the *avail* values at the quorum servers by sending *lock*/*unlock* messages. Such *remote* locking involving multiple servers, as our results will demonstrate, is relatively expensive to perform in wide-area and is the main performance bottleneck of replication-based approaches such as GSE.

1. Lock $avail_i$ and set $Q_t = \{s_i\}$.
2. Set $es_i = \lfloor avail_i / n * |Q_t| \rfloor$.
3. Compute U_i.
4. If $es_i < req(t) + \sum_{t_u \in U_i} req(t_u)$
   If $|Q_t| < n$
   Lock $avail_j$ of a server $s_j \notin Q_t$, and add s_j to Q_t
   Goto Step 2.
   Else abort; and for all $s_j \in Q_t$, unlock $avail_j$.
5. Perform requisite updates
6. Commit; and for $s_j \in Q_t$, unlock $avail_j$

Figure 25: The GSE algorithm
implemented by si to execute transaction t

Figure 25 depicts the basic GSE algorithm. Initially, only s_i is in the quorum of t (denoted by Q_t). Step 2 computes the escrow quantity es_i. The tight synchronization among the servers in Q_t enables s_i to use the escrow quantities of the other servers in Q_t. Since s_i is not up-to-date regarding the token allocations performed by the other servers in the system, it, thus, makes a conservative estimate regarding its escrow size. Server s_i makes this estimate by placing an upper bound on the escrow values assumed by non-quorum servers; thereby guaranteeing that simultaneous token allocations do not cause a lower-bound constraint violation. In order to make this estimate, s_i computes U_i, which denotes the set of token allocation transactions t_u such that t_u is known to s_i, and s_i is not sure that t_u is known to all the non-quorum servers. The sum

$$req(t) + \sum_{t_u \in U_i} req(t_u)$$

provides an upper bound on the number of token allocations that might be unknown to non-quorum servers. The inequality in Step 4 checks whether all but (at most) the last $es_i - req(t)$ allocations known to s_i are known to all the non-quorum servers. If this inequality evaluates to true, it means that es_i is insufficient; additional servers need to be added to Q_t. This is accomplished by making (remote) lock requests to involved servers regarding their views of *avail*, $avail_j$ for s_j). During this locking phase, the views of *avail* at the quorum servers are synchronized (i.e., the *avail* values at the quorum servers become equal). If es_i is still not sufficient and all servers are already in the quorum (Step 4), the transaction is aborted. If es_i is sufficient, the execution of *t* can proceed.

After all the necessary updates and logging are performed, *t* commits and all local and remote locks are released. It is important to emphasize that the performance of GSE heavily depends on how up-to-date each server is regarding the global token allocations performed and the cost of synchronization among servers

5.5 Experimental Environment

5.5.1 Simulation Model

In order to evaluate the performance of GSE and the DVP strategies, we implemented a detailed simulation model using CSIM [1]. The model consists of components that model a distributed set of servers, a population of clients making commodity requests, and communication latencies among servers.

Server Module

The server module is responsible for executing client transactions using either DVP or GSE. It consists of:

a *resource manager*, which schedules the CPU and disk (using a non-preemptive FIFO policy);

a *buffer manager*, which handles transfer of data between the disk and buffer;

a *communication manager*, which handles the passing of messages to and from the network module using a queue to implement ordered message retrieval and processing. Every message sent or received by the server is charged a fixed CPU cost, specified in terms of the number of instructions executed. No per-message-byte cost is modeled since we assume the use of short, fixed-sized control messages; and

a *transaction manager*, which coordinates the execution of transactions.

We now describe the transaction manager in more detail. The transaction manager consists of several components. One component is the *lock manager*, which handles all locking associated with a given conweight protocol. Deadlocks are handled using a timeout mechanism. The transaction manager also contains several algorithm-specific components. The *escrow manager* maintains all necessary information and makes all the quorum-based decisions when modeling the GSE algorithm. It also contains a *gossip manager* component that coordinates the sending of gossip messages to other servers. The other algorithm-specific component is the *DVP manager*, which is responsible for implementing the redistribution strategies that we described. The DVP manager also contains a *gossip manager* component that is responsible for disseminating state information via message exchange. Every message sent or received by a server is

charged a fixed CPU cost. No per-message-byte cost is modeled since we assume the use of short, fixed-sized control messages.

Client Workload Generator Module

The client workload generator models a population of clients submitting transactions to the system. The model generates transactions using exponential inter-arrival times and submits them to the network module for delivery to a server.

We model the transactions based on common characteristics of typical wide-area commodity distribution applications we discussed in Section 0. A client request involves a single commodity with a specific number of tokens (e.g., buying two tickets for a particular show). Note that we do not investigate transactions that involve multiple commodities in this work. Although it is straightforward to generalize our algorithms for that case, such a generalization is beyond the scope of this work.

Infrequently, the tokens obtained from the system are returned (e.g., the return of a book, cancellation of a ticket). The transactions we generate, therefore, specify a particular commodity and a value indicating the number of tokens requested of that commodity.

The commodity to be requested is selected uniformly among all the commodities in the database, and the number of tokens to request is chosen uniformly from a given interval. Each successfully executed transaction potentially has a return transaction that returns the tokens obtained by the original transaction. A return transaction is submitted to the system with a given probability. Return transactions, if submitted, are issued by the client that previously submitted the corresponding allocation transaction.

Network Module

The *network* module models Internet communication latencies among the servers. Rather than using a synthetic model to generate communication latencies and inject certain failure modes (e.g., message losses, network partitions, etc.), we sampled messaging latencies over the Internet. For a period of three days, we continuously collected traces of pair-wise ICMP (i.e., ping) message exchange among four servers, which are located at College Park (Maryland), Murray Hill (New Jersey), Lexington (Kentucky) and Santa Barbara (California).

The traces were then formatted into lists whose entries are the latencies of each message exchange or an indication that the message was lost (approximately 2% of the messages were lost).

The server-server traces are used to drive the network component of the simulations as follows. At the beginning of a simulation run, one trace is randomly (uniformly) chosen for each pair of servers, and one entry is selected randomly (uniformly) on each trace as a starting point. Whenever a server sends a message to another server, the current latency value for the corresponding trace is read and the current entry is incremented. The message is delayed by this latency value and then inserted into the message queue of the destination server. If the entry indicates a message loss, then the message is re-sent after a timeout period of 200 ms (twice the average round-trip time between any pair of servers).

In order to model the communication latencies between servers and clients, we used a similar approach. We used client machines whose IP addresses were obtained from the web-server traces of a large telecommunications company. We chose 400 ma-

Parameter	*Setting*	*Parameter*	*Setting*
Number of servers	10	Database size	200 (commodities)
Local lock timeout	200 (ms)	Number of tokens	200
Remote lock timeout	500 (ms)	Number of clients	400
CPU speed	2000 (ms)	Transaction inter-arrival time	exp(10.0) (msec)
Disk latency	0.0097 (s)	Update transaction probability	1.0
Disk transfer rate	40 (MB/s)	Transaction size	1 (commodities)
Buffer hit ratio	0.95	Tokens requested per transaction	uniform(1,5) (tokens)
Item read cost	1000 (instr.)	Return transaction probability	0.1
Item write cost	2000 (instr.)	Workload skew (θ)	0.0 or 1.0
Message cost	10000 (instr.)	Prefetch period	100 (s)
Control message size	64 (bytes)	Callback threshold	20 (tokens)
Commodity item size	512 (bytes)	Sharing constant	1.0

Table 3: Primary simulation parameters and default settings

chines that were scattered over the Internet. We sent control messages from our four servers to these clients for a period of three days, and logged the latency values observed. The communication between the clients and the servers consists of a short client message that includes the transaction requested and a short response message from the server indicating the result of the transaction submitted. Thus, the sampled latencies can reasonably be used in modeling the client-server communication in our environment. The use of a client-server trace is similar to that described above for the server-server case.

The use of wide-area ICMP traces as described is a reasonable technique for our purposes, since (a) our model consists of servers communicating via message exchange over a wide-area network, and (b) the protocols we study only require the transfer of control messages but not any data messages. Table 3 shows the main simulation parameters and their default settings.

5.5.2 Methodology

All the results reported in the following section are the averaged results of ten independent simulation runs. For each set of runs, the same set of ten workloads (i.e., the same clients submitting the same requests at the same time) are used when comparing different approaches to ensure fairness. The only factor that results in different loads for the different approaches is the existence of return transactions. Since the scheduling of a return transaction is dependent on whether or not the corresponding allocation transaction commits, different approaches will typically see different return transactions. In particular, the higher the commit rate for an approach, the more return transactions that approach will observe.

Note that there are two reasons why a transaction may *not* commit in our environment. First, a transaction may timeout waiting for a local or remote lock and abort. Second, either there may not be sufficient tokens in the system to execute the transaction successfully or there may be sufficient tokens but the system may not be able to locate them in a timely manner.

5.6 Performance Experiments and Results

This section present the results of our performance experiments comparing our DVP strategies and the (extended) GSE approach. We first discuss two modifications to the basic GSE approach that significantly improved its performance in our experiments. Our first modification, which was suggested in [45], involves the construction of a quorum *in parallel* rather than sequentially. Our second modification requires a server to send gossip messages to all others as it learns of a token allocation. All the results reported below are the averaged results of ten independent runs.

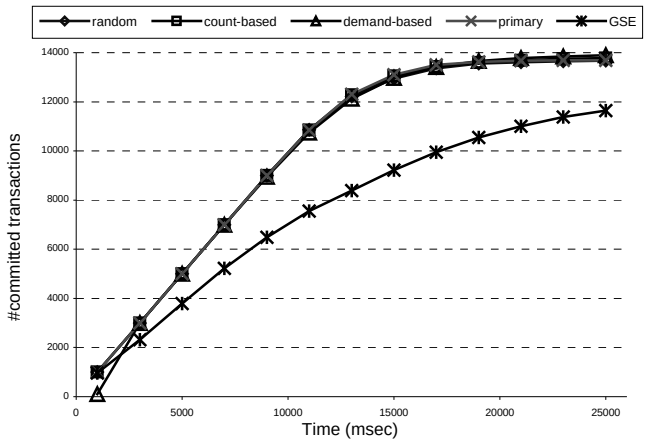

Figure 26: Number of committed transactions
uniform, 10 servers, 100 trans/s

5.6.1 Basic Performance

The graphs we present in this section demonstrate how the performance of the system changes over time. As to be expected, the performance of all the approaches becomes worse as the number of tokens globally available in the system decreases. In all the experiments, we fix the number of servers at 10 and the mean inter-arrival time for client transactions at 10 ms.

Uniform workload

Figure 26 presents *num-commits*, the number of committed transactions, by GSE and the DVP strategies under a workload where the transaction requests are uniformly distributed across servers; i.e., $\theta = 0$. The figure reveals that the DVP strategies deliver significantly higher *num-commits* than GSE. For all the DVP strategies, *num-commits* initially increases rapidly. With each committed allocation transaction, however, *avail*, the number of tokens globally available, decreases. *num-commits*, then, settles down as very few transactions can commit due to a lack of tokens in the system. The differ-

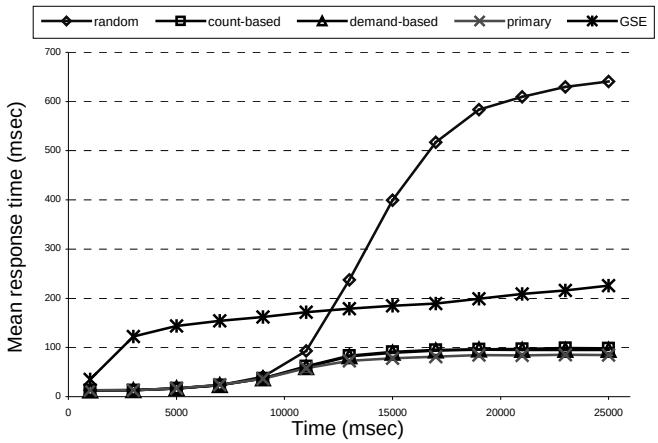

Figure 27: Mean response time
uniform, 10 servers, 100 trans/s

ences among different DVP approaches here are not significant. Figure 27 shows the complementary *resp-time*, mean response time, results. All the DVP strategies, except `random`, outperform GSE throughout the entire execution range. These DVP variants manage to keep their *resp-time* quite low, whereas the performance of GSE deteriorates quickly after several hundred transactions are executed. The *resp-time* of `random` drastically increases as *avail* decreases since it selects the lender servers randomly and cannot tell whether a server has any tokens or not. The other DVP strategies do not suffer from the same problem, achieving much better *resp-time* values.

The GSE approach, as explained in Section 5.4, operates by forming a quorum of servers. It thereby requires tight quorum synchronization, which is quite costly in a wide-area environment. The DVP approaches do not require quorum synchronization: they can continue executing transactions locally as long as they have sufficient tokens at their disposal. Even in the case when a server runs out of sufficient tokens and has

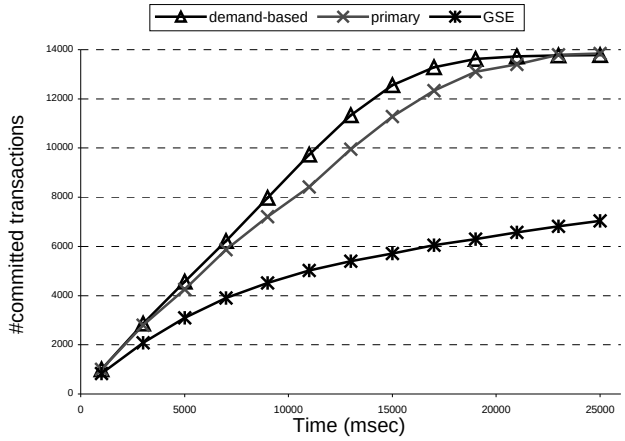

Figure 28: Number of committed transactions

to make requests to other servers, it never has to communicate synchronously with multiple servers.

A close look at the raw results clearly demonstrates the fundamental drawbacks of GSE: (1) GSE cannot execute as many transactions locally as DVP approaches, and (2) quorum sizes increase as *avail* decreases. GSE, therefore, not only has to contact other servers most of the time, but also has to lock more and more servers as *avail* decreases, aggravating its inefficiency. Another problem of GSE is its high abort rate, which occurs mainly due to timeout in lock waits.

Skewed Workload

Figure 28 presents the *num-commits* achieved by GSE and the DVP strategies under a skewed workload where servers receive transactions based on a highly-skewed Zipf distribution; i.e., θ =1. We drop `random` and `count-based` from presentation in the rest of the graphs, as they are consistently outperformed by the other DVP approaches. Comparison of these results with those for the balanced case immediately shows that (1) all approaches are negatively impacted by the high skew in the work-

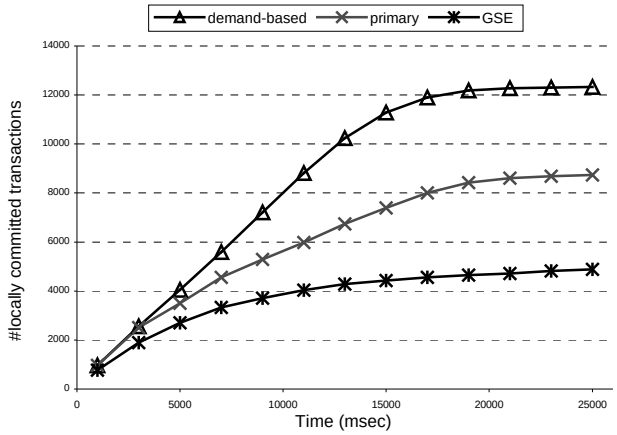

Figure 29: Number of locally executed transactions
skewed, 10 servers, 100 trans/s

load, (2) DVP approaches still significantly outperform GSE, and (3) `demand-based` achieves the highest *num-commits*.

For all the approaches, a few *popular* servers perform most of the token allocations due to the workload skew. In GSE, although the escrow sizes vary dynamically, the popular servers exhaust the tokens in their local escrows rapidly, having to form quorums most of the time. In DVP, the popular servers consume their local tokens quickly, and then have to contact the less popular servers in order to obtain tokens.

Comparing the individual DVP strategies, observe that `demand-based` achieves the highest *num-commits* due to its ability to redistribute tokens dynamically across servers via sharing and prefetching; i.e., by continuously transferring tokens from the less popular servers to more popular ones. The `count-based` and `primary` strategies provide virtually equivalent performance. Figure 29, which shows the number of transactions that are executed *locally* without contacting other servers, further justifies this behavior. In terms of response time (not shown), `demand-based` also outperforms the other approaches.

5.6.2 Other Experiments

In the rest of the section, we summarize the results of the additional experiments that we performed (complete results can be found in [19]).

Scalability and Contention Effects

We conducted experiments where we varied the number of servers and transaction rates. We observed that the DVP approaches scaled and performed better than GSE — `primary` demonstrated the best scalability since its performance is not affected much by the number of servers once the system switches to centralized mode.

Load Balancing

Comparison of the results for the balanced and skewed workload cases reveal that the DVP approaches suffer, although not as much as GSE, from load imbalance. We, therefore, investigated the potential benefits of employing a simple *load balancing* mechanism to distribute the load evenly across servers. In this mechanism each client, rather than submitting the transaction to its *closest* server, *randomly* picks a server to submit the transaction (note that this is a very practical and easy-to-implement scheme). Servers use the `demand-based` strategy — or any other strategy for that matter — without any modifications. The advantage of such load balancing is that the workload seen by each server will (on average) be similar. In such a case, as we observed from the uniform workload results, the DVP approaches will demonstrate significantly better performance. The downside is that such a randomized selection eliminates the potential gains of submitting the transaction to the closest server. This restriction potentially results in increased *client-side* response times. Experimental re-

sults demonstrate that this is indeed the case: Our simple load balancing technique achieves better server-side response times than the *no*-load balancing strategy. In fact, it achieves results quite similar to those of `demand-based` for the uniform workload case. Client-side response times, however, turn out to be similar, or slightly better for the *no*-load balancing case, demonstrating the aforementioned tradeoff.

Non-Deterministic Selection of Lenders

The DVP strategies presented are deterministic in that they use the estimated token counts at servers to choose the lender servers. This deterministic selection may lead to a situation where, for a given period of time, most of the token requests are targeted to the same server, making that server a *hot-spot*. Our experiments showed that, due to the differences among the views at different servers, such situations do not occur very frequently: the probability is no more than 10% and 30% higher than the random selection case on average for the uniform and skewed cases, respectively.

In order to eliminate such situations completely, we studied two randomized schemes. The first scheme chooses the lenders randomly with equal probability (among those with a non-zero token count). The other scheme is based on the idea of *lottery scheduling* [69], where lenders are chosen with a probability proportional to their token counts. Experiments reveal that the lottery-scheduling scheme performs somewhat better than the completely random scheme, achieving commit rates similar to those of the deterministic scheme, while suffering less than a 20% deterioration in response time under both the uniform and skewed workloads (using `count-based`).

5.7 Summary

Token-based commodity distribution can meet the demands of a class of newly emerging Internet-based e-commerce applications. In this work, we experimentally evaluated and compared two fundamentally different approaches to token distribution — partitioning and replication — using real Internet message traces. We proposed several pair-wise token redistribution strategies for the partitioning-based approach and experimentally evaluated them under different workloads.

Our experiments reveal a number of significant results for wide-area token-based commodity distribution. First, replication-based approaches are neither necessary nor desirable for the kinds of applications and environment we address in this study: partitioning-based approaches perform and scale better primarily due to their ability to provide higher server autonomy. Second, the use of information about the system state (e.g., token counts, token demand rates) turns out to be crucial for making redistribution decisions. Third, in the case of non-uniform workloads, demand-based redistribution yields notable performance improvements.

Chapter 6 Decentralized Numerical Divergence Control Protocols

Replication is a crucial mechanism to effectively support applications in mobile and wide-area environments. The cost of maintaining strict consistency in these environments is prohibitive due to several factors such as ever-increasing scale in wide-area, and communication restrictions in mobile environments. Furthermore, many applications do not require strict consistency and can continue to operate with stale data, as long as divergence from the accurate, up-to-date data is properly bounded. Examples of such applications include:

- *Mobile sales and inventory applications*: Mobile sales people sell consumable or financial products using personal digital assistants. The product inventories shared by the sales force are typically limited and the products should not be oversold. It is, however, impractical and/or uneconomical to keep sales people connected and, thus, up-to-date regarding the inventory at all times. On the other hand, a salesperson does not care about the accurate number of products available in the inventory as long as this number is larger than the number of items she is selling at any instant. For instance, if the sales person is selling three units of a particular product, the exact number of units left in the inventory does not matter as long as at least three units are still available for sale.
- *Wide-area network management* [54, 59]: The responsibility of the network manager is to monitor the external traffic on the network, identify performance bottlenecks and then properly reallocate resources to alleviate any problems. Several nodes in the system maintain accurate traffic information local to their domain. Due to the scale of the system, it is neither feasible nor necessary for the manager

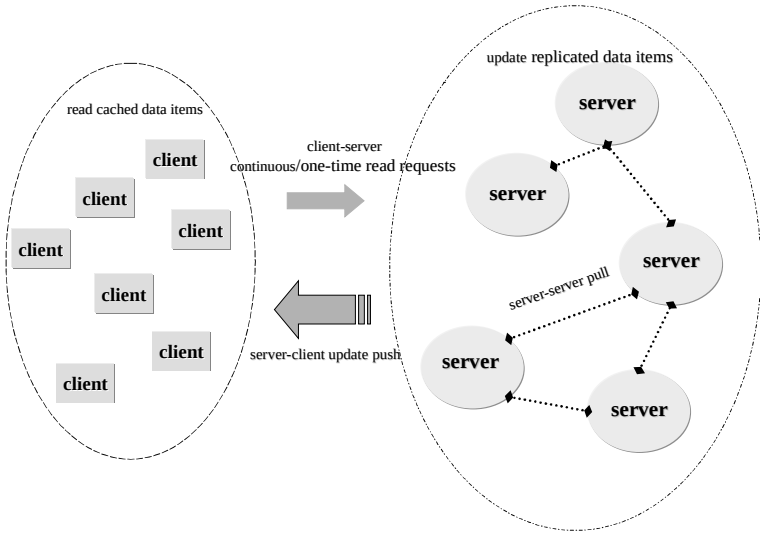

Figure 30: ReBound system model

to retrieve accurate information from each such node at all times. The manager can tolerate inaccurate traffic-related values as long as the *divergence* or *error* is bounded. For instance, a manager monitoring the average traffic latency at a certain portion of the network may desire to observe latency values within ± 5 milliseconds.

- *Replicated stock quotes services* [73]: Users obtain stock quotes from replicated stock quotes servers. Due to the scale of the system, it is very expensive to refresh the cache of all clients after each update. On the other hand, users may want to ensure that the values they observe deviate from the accurate values only by a limited amount. For example, it may be acceptable for a user to observe inaccurate stock values as long as the values she reads are within ± 5 cent/share.

In this section, we present the basic design, implementation, and evaluation of *ReBound*, an adaptable middleware system that provides a flexible framework for deploying such applications in mobile and wide-area environments. *ReBound* incorpo-

rates mechanisms to trade off the degree of consistency of numeric replicated data with the efficiency of access. Figure 30 illustrates the basic ReBound system model. Servers update distributed, replicated numerical data, which are cached by clients. Clients submit read operations that specify divergence bounds on the data they cache. These bounds indicate limits on the quantitative deviation of the data read by clients from the *accurate*, up-to-date values maintained at the servers. To illustrate the basic model more formally, consider a single server *s* and a client *c*. A read issued by *c* to *s* on a data item *r* specifying a divergence bound $[\delta^-, \delta^+]$, where $\delta^- \leq 0 \leq \delta^+$, requires *s* to guarantee that the accurate value of *r* maintained at *s* lies in the range $[cache(r) + \delta_c^-, cache(r) + \delta_c^+]$, where *cache*(*r*) is the cached value of *r* at *c* (see the following section for the generalized definition).

We describe the semantics of our framework along three dimensions: *Configuration*, *Persistence*, and *Hardness.* Configuration refers to the set of valid functionality configurations, the number and selection of servers and clients, supported by the system. ReBound allows any subset of nodes to act as servers, clients, or both. Persistence refers to the temporal aspect of enforcing client-specified bounds. ReBound supports *continuous* bounds, which are registered to servers only once and then continuously enforced, as well as *one-time* bounds, which are specified and enforced only once. Hardness addresses the guarantees regarding the strictness of bound enforcement. ReBound framework supports *hard* bounds, which are strictly enforced at all times, and *soft* bounds, which may temporarily be violated. We argue that many popular mobile and wide-area applications can effectively specify their consistency requirements using our framework.

We describe two algorithms to implement the above framework and maintain the divergence bounds specified by clients. The algorithms maintain continuous bounds by efficiently bounding the updates that are committed at servers and are *unknown* to clients (i.e., not yet propagated to clients). Upon receiving an update, a server checks local criteria to decide whether the commitment of the update violates any divergence bounds. If the criteria are met, the server commits the update locally. If not, the server must perform remote communication and either push the unknown updates to a subset of clients, or pull information from a subset of servers. The algorithms handle one-time bounds by exploiting the already registered continuous divergence bounds, if available, to efficiently select a proper subset of servers whose unknown updates need to be pushed to the clients to satisfy the specified bound.

Despite the importance of divergence bounding of numerical replicated data for many popular distributed applications and services, this topic is yet to be well addressed in the literature. Earlier efforts (e.g., [9, 44, 45, 71]) attacked more general problems, and typically relied on techniques that are impractical and/or inefficient for wide-area, and especially for mobile environments. Recent efforts [54, 73] provided solid results within their scope, but lacked generality, restricting their applicability to a certain class of applications. In this work, our aim is to generalize and further previous efforts on divergence bounding of numerical replicated data into a single, general framework.

The rest of the chapter is organized as follows. Section 6.1 provides an overview of the ReBound numerical divergence control framework. Section 6.2 describes the basic system model that we assume throughout the chapter. Section 6.3 describes the de-

centralized server-side algorithms that we propose to support continuos divergence bounds. In Section 6.4, we describe how and when client caches are refreshed in ReBound and in Section 6.5 we describe the extension of our algorithms to support one-time divergence bounds. Section 6.6 presents the ReBound architecture and Section 6.8 describes the experimental study we performed to quantify the performance of ReBound. Section 6.9 concludes by summarizing the main results of the ReBound work.

6.1 Overview of ReBound

6.1.1 ReBound Framework

In ReBound, clients specify their desired quantitative divergence constraints in terms of numerical bounds on the data items they cache. Servers that replicate and update those items cooperatively maintain the specified constraints. We first define the weight of an update as the amount by which the update changes the value of the corresponding data item. We define a divergence bound on a data item r with respect to a set C of clients and a set S of servers as the sum of the weights of updates, u, on r that the servers in S can commit without ensuring that all the clients in C have observed u. More formally, the algorithms we propose ensure that the following inequality hold at all times:

$$\delta_C^- \leq \sum_{s_i \in S} w(unknown_i) \leq \delta_C^+$$

where $\delta_C^- \leq 0 \leq \delta_C^+$ are, respectively, the lower and upper bounds to be preserved, and $w(unknown_i)$ is the sum of the weights of the updates committed by $s_i \in S$ and not yet reflected to the caches of all clients $c \in C$. Intuitively, the above constraint limits

the total weight of updates the servers can commit without refreshing client caches, essentially limiting the numerical divergence of the values of the data cached at clients.

We use three dimensions to describe the semantics of our framework:

- *Configuration* defines the number and selection of servers and clients over which a divergence constraint can be defined. Many previous proposals restrict allowable system configurations and define protocols that typically support single-server/single-client (e.g., [54]) or multiple-server/single-client (e.g. [73]) configurations. As we argue below, the semantics of many applications require the use of a more general *multiple-server/multiple-client* model that allows a divergence constraint to be defined over any number of servers and clients. *ReBound* supports this more general model, and therefore subsumes the functionality of previous models.
- *Persistence* refers to the temporal aspect of the client-specified divergence constraints. A *continuous* divergence constraint is one that is registered to servers only once and then continuously enforced. A *one-time* constraint, on the other hand, is specified and enforced only once. The availability of continuous constraints is typically sufficient for most applications that we target. One-time constraints, however, are desirable in situations where more accuracy than that provided by the continuous constraints is required (e.g., when a user wants to know the accurate worth of her stock portfolio before making a new trade). ReBound supports both continuous and one-time constraints. Note that the support for continuous and one-time constraints need not be exclusive.

- *Hardness* addresses the guarantees regarding the strictness of bound *enforcement*. *Hard* bounds are those that need to be strictly enforced at all times (through the use of standard locking-based techniques [12, 30]). As we argue below, such hard bounds are typically needed in distributed resource allocation applications such as mobile sales/inventory applications and airline reservation systems, where the availability of resources need to be verified before they are allocated. *Soft* bounds are less strict in that they allow situations where the divergence constraints may be temporarily violated. Soft bounds are especially well-suited for resource monitoring applications such as distributed sensor systems where large volume of updates to base data arrive from outside the system itself. In such cases, support for hard constraints simply become impractical. ReBound supports both hard and soft divergence bounds.

6.1.2 Generality of the ReBound Framework

Having described our framework, we now reconsider the motivating applications discussed in Section 6.1, and argue the generality of our framework by illustrating how their semantics can be specified within our framework:

- *Mobile sales and inventory*: The divergence constraint to be maintained, for each sales item, is a strict, global lower bound on the inventory size (*IS*), which requires that $IS \geq 0$ at all times. Each mobile sales device is a server (since it can generate sales and thus need to update *IS*), and a client (since the constraint is global) at the same time. Our framework captures this semantic requirements by defining a *hard*, *continuous* divergence bound with $[\delta^-, \delta^+] = [0, IS]$, and *multiple-server/multiple-client* configuration defined over all mobile machines.

- *Wide-area network management*: The network monitoring station registers a desired level of accuracy on the traffic data it obtains from several network nodes. Since each network node updates its own traffic data, each acts as a server. Since each node updates its traffic data in real-time, support for hard constraints is prohibitive over wide-area, requiring *soft* constraints. Each constraint is then assigned a *single-server/single-client* (a network node/the monitoring station). If the monitor observes an unanticipated situation, it may further inquire the network traffic nodes using finer-accuracy *one-time* reads.
- *Replicated stock quotes services*: Client nodes specify the maximum tolerable divergence in the particular stock quote values they observe locally. As each server may accept updates to quotes at any time, only soft constraints are practical. These requirements can be satisfied by specifying continuous constraints that are assigned by single clients to multiple replicated stock quote servers.

6.2 ReBound System Model

The system consists of a set S of n servers, $S = \{s_1, s_2, s_3, ..., s_n\}$, and a set C of m clients, $C = \{c_1, c_2, c_3, ..., c_m\}$. Note that S and C are not necessarily disjoint, and we use the term *node* to refer to a server or a client if the distinction is irrelevant. Servers replicate and update numerical data items. An update u changes the value of an item by an amount equivalent to its weight, $w(u)$, which can be positive or negative. Clients cache read-only versions of a subset of these items, and can set *divergence bounds* on the items they cache. Servers that replicate an item cooperate to maintain any divergence bounds defined on that item.

A server commits an update when the server ensures that the update does not violate any client-specified bounds. The server that accepted an update u is called the *initiating* server of u. Updates are always propagated and maintained in the order they are committed at their respective initiating servers. A server reflects the updates it commits to other nodes by propagating the corresponding update records, which are then applied by other servers or clients to their local databases or caches, respectively (note that data item images are never propagated). Servers can perform update propagation using any information propagation mechanism, such as broadcast, multicast, gossip messages [17], or anti-entropy [23, 57], supported by the underlying communications environment. The particular mechanism used does not affect the correctness of our algorithms; only the performance is affected. Since our framework is general and is designed specifically for mobile and wide-area environments, we utilize *pair-wise* synchronization sessions for update and information propagation.

For simplicity of exposition, we assume that there is a single data item replicated by all servers and cached by all clients in the system. Each server s_i maintains a *view* that summarizes the updates that other nodes have seen. We represent the view of s_i as a vector v^i such that $v^i[j,k]$ gives the number of updates committed by node k and known to node j in s_i's view. Views are updated when a server commits a new update locally, or during update propagation. When s_s commits a new update locally, it sets

$$v^i[i,i] = v^i[i,i] + 1$$

When a server s_l propagates its view, v^l, to s_i, s_i updates its view such that:

$$v^i[j,k] = \max(v^i[j,k], v^l[j,k]), \text{ for all nodes } j, k.$$

Each server s_i maintains a *commit log*, which is a sequence of updates either committed by s_i, or committed by another server and propagated to s_i. We represent the sequence of updates committed by s_k in s_i's log by

$$U_k^i =< u_k^i[1], u_k^i[2], u_k^i[3], ..., u_k^i[v^i[i,k]] >$$

We now make several definitions that we will use throughout our discussion. We first discuss what we mean by an unknown update:

Definition 10 (Unknown update) We say that an update u is unknown with respect to a client set C if u is not yet observed by all clients $c \in C$.

Based on the definition of unknown updates, we now define an unknown update sequence as follows:

Definition 11 (Unknown update sequence) We define a sequence of updates, *unknown*^{i}(*j*,*k*), as the sequence

$$< u_k^i[x+1], u_k^i[x+2], u_k^i[x+3], ..., u_k^i[y] >,$$

where $x = min(v^i[c,k])\ \forall c \in C$, and $y = v^i[j,k]$.

Informally, *unknown*^{i}(*j*,*k*) is the sequence of unknown updates committed by s_k as seen by node *j* in s_i's view.

Definition 12 (Weight of a sequence) The weight of a sequence of updates $U =< u_1, u_2, u_3, ..., u_n >$, $w(U)$, is the sum of the weights of the updates in the sequence; i.e., $w(U) = \sum_i w(u_i)$, $i = 1 \ldots n$.

We now define the minimum and maximum suffix of a sequence:

Server s_i:

1. Lock r
2. While (*local_commit_criteria* (r, u, $[\delta^-, \delta^+]$) not satisfied)

   / * based on some divergence control policy */

   push-to-clients(), and/or

   /* push unknown updates to clients (to advance v^i) */

   pull-from-servers()

   /* redistribute bounds (to relax local commit criteria) */
3. Set $r = r + w(u)$
4. Unlock r

Figure 31: Basic algorithm executed by server s_i to commit an update u (on a data item r)

Definition 13 (*Minimum suffix of a sequence*) We define the minimum suffix of a sequence U as $\min(U) = < u_k, u_{k+1}, u_{k+2}, ..., u_n >$ where $\sum_{i=k}^{n} w(u_i) < \sum_{i=j}^{n} w(u_i), \forall j \neq k$, $0 < k < j \leq n$.

Definition 14 (*Maximum suffix of a sequence*) We define the maximum suffix of a sequence U as $\max(U) = < u_k, u_{k+1}, u_{k+2}, ..., u_n >$ where $\sum_{i=k}^{n} w(u_i) > \sum_{i=j}^{n} w(u_i), \forall j \neq k$, $0 < k < j \leq n$.

Informally, min(U) and max(U) are suffix subsequences of U with minimum and maximum weights.

6.3 Algorithms for Numerical Divergence Control

In this section, we describe two server-side algorithms for enforcing the continuous divergence bounds as described above. Both algorithms work by efficiently limiting the sum of the weights of updates that can be committed by the servers without the

clients knowing about them. The key insight to both algorithms is to treat the allowed divergence bounds as global resources to be consumed by the servers.

In the first algorithm, *Share-Bound* (SB), each divergence constraint is *shared* (i.e., replicated) among servers, whereas in the second algorithm, *Partition-Bound* (PB), each divergence constraint is *partitioned* among servers. A server, before committing a new update, must ensure that the update does not violate the shared constraint (in the case of SB) or the server's local constraint (in the case of PB). Servers are assigned per-bound weights[4] that define the size of the share of the server in the divergence constraint and the size of local constraints in, respectively, SB and PB.

In both algorithms, each server initially attempts to commit a new update locally based solely on local information. If this is not possible, the server must bring clients up-to-date and/or somehow relax its local constraint. Clients can be brought up-to-date simply by pushing the updates unknown to them. A server can relax its local constraint by tightening local constraints of other servers. This is facilitated through pair-wise weight redistribution, which involves one server pulling some amount of weight from another. While the pulling server's weight increases (causing its local constraint to be relaxed), the pulled server's weight decreases by the same amount (causing its local constraint to be tightened). Figure 31 illustrates the basic algorithm executed by a server s_i to commit a new update u.

---

[4] Note that each server is assigned a *bound weight* per each divergence bound they maintain. These bound weights should not be confused with the weight of an update. When we talk about the weight of a server, we always imply the server's bound weight.

In the rest of the chapter, we use the term *global* range to indicate the client-specified divergence range. We discuss, for each algorithm, how a single divergence range, $[\delta^-, \delta^+]$, that is registered at a set of servers S by a set of clients C is maintained.

We first present the local update commit criteria used by each server to decide whether it can commit an update *locally*. We then address the case when the update cannot be committed locally by presenting formulas for (1) *minimum quantitative view advance*, and (2) *minimum bound weight increase*, needed to commit an update. We also discuss the mechanics of server-server weight redistribution in detail.

6.3.1 The Share-Bound Algorithm

In SB, the global divergence range, $[\delta^-, \delta^+]$, is shared by all servers in the system. Upon accepting a new update u, a server s_i checks whether the commitment of u will violate the shared global range. For this purpose, s_i computes a local range, $[\delta_i^-, \delta_i^+]$, based on:

1. the global range,
2. Server s_i's local knowledge about the updates committed by the other servers in the system, and
3. Server s_i's bound *weights* that define s_i's portion in the global range.

Since s_i's local knowledge may not be up-to-date, s_i conservatively computes its own local bounds by *assuming an upper bound on the ranges of other servers*. These local bounds then indicate the sum of the weights of updates that s_i can commit entirely locally, i.e., without contacting any other server or client.

Each server s_i computes its local divergence range as follows:

$$[\delta_i^-, \delta_i^+] = [w_i^-(\delta^- - \sum_{j \neq i} w(\min(unknown_j^i))), w_i^+(\delta^+ - \sum_{j \neq i} w(\max(unknown_j^i)))]$$

where $(w_i^-, w_i^+), 0 \leq w_i^-, w_i^+ \leq 1.0$, are s_i's lower and upper bound weights. Without loss of generality consider the local upper bound as computed above. Intuitively, the value $\delta^+ - \sum_{j \neq i} w(\max(unknown_j^i))$ gives the total maximum weight of updates that can be committed without going over the global upper bound δ^+ (the sum of the weights of updates committed by s_i is factored in the later). Server s_i then computes its share of this value by using its upper bound weight, and thus computing the total maximum weight of updates that it can commit locally. Note that the sum of bound weights across all servers always sum up to 1.0; i.e., $0 \leq w_i^-, w_i^+ \leq 1.0$ and $\sum_i w_i^- = \sum_i w_i^+ = 1.0$, $\forall i = 1 \ldots n$. Fixing the sum of all bound weights in this manner, as we discuss in Section 6.4.2, enables light-weight, server-server bound redistribution while maintaining the correctness of the algorithm.

(*Local commit criteria*) Server s_i commits a new update u if:

1. $w(\min(unknown_i^i)) + w(u) \geq \delta_i^-$ *if* $w(u) < 0$, or
2. $w(\max(unknown_i^i)) + w(u) \leq \delta_i^+$ *if* $w(u) > 0$

In other words, s_i can commit u if the sum of the weights of the unknown updates committed by s_i (including u) does not exceed s_i's local bounds, and thus, does not invalidate the shared global bounds.

6.3.2 The Partition-Bound (PB) Algorithm

In this approach, the global divergence bounds are explicitly *partitioned* as local bounds across the servers using the local bound weights at each server. The local bounds at a server s_i are computed as:

$$[\delta_i^-, \delta_i^+] = [w_i^- \delta^-, w_i^+ \delta^+]$$

For all i=1…n, $0 \le w_i^-, w_i^+ \le 1.0$ and $\sum_i w_i^- = \sum_i w_i^+ = 1.0$, the sum of the local bounds always sum up to the global bounds: $\delta^- = \sum_i \delta_i^-$ and $\delta^+ = \sum_i \delta_i^+$. The local bounds at each server s_i bound the sum of the weights of the updates that s_i can commit without contacting other servers and refreshing client caches. Unlike Share-Bound, Partition-Bound does *not* require a server to take into account the updates committed by other servers when computing local bounds. This is due to the fact that global bounds are not shared in Partition-Bound. Since the global bounds are partitioned, it is sufficient for each server to limit only its own updates to ensure that the global bounds are maintained, regardless of the updates committed by others.

(*Local commit criteria*) Server s_i can commit a new update u if:

1. $w(\min(unknown_i^i)) + w(u) \ge \delta_i^- \quad if \ w(u) < 0$, and
2. $w(\max(unknown_i^i)) + w(u) \le \delta_i^+ \quad if \ w(u) > 0$

6.4 Server-Client Update Propagation and Server-Server Bound Redistribution

If a server cannot commit a new update locally based on the computed local bounds, it has two alternatives that can be used to increase the sum of the weights of updates that it can commit locally:

- Advance local view by pushing updates to a subset of clients
- Extend local divergence bounds by bound redistribution (efficiently accomplished by pulling bound weights from a subset of servers).

In this section, we discuss these two alternatives in detail.

6.4.1 Refreshing Client Caches with Unknown Updates

One way to accommodate a new update is to sufficiently advance the server's view of the clients regarding the updates unknown to those clients. This can be accomplished by refreshing the caches of a proper subset of clients by pushing a subset of the updates unknown to those clients. More specifically, server s_i chooses a subset of clients $C_i \subseteq C$ such that the propagation of unknown updates to each client in C_i will advance the view of s_i, and therefore potentially decrease $unknown_j^i$ for some $j = 1 \ldots n$.

Definition 15 (*Minimum quantitative view advance*) We define the minimum quantitative view advance server s_i requires as the value decrease in the sum of weights of the unknown updates required to commit a new update u.

Based on the local commit criteria presented in Section 6.3.1, we calculate the minimum required view advance for a server s_i using the Share-Bound algorithm as follows:

$$req\_view\_adv_i = \begin{cases} [w_i^-[\delta^- - \sum_{j \neq i} w(\min(unknown_j^i))] + w(\min(unknown_i^i)) + w(u)] - \delta_i^-, \\ \qquad\qquad if\ \ w(u) < 0 \\ [w_i^+[\delta^+ - \sum_{j \neq i} w(\max(unknown_j^i))] + w(\max(unknown_i^i)) + w(u)] - \delta_i^+, \\ \qquad\qquad if\ \ w(u) > 0 \end{cases}$$

For the Partition-Bound algorithm, the corresponding value can be calculated as:

$$req\_view\_adv_i = \begin{cases} [w_i^-(\min(unknown_i^i)) + w(u)] - \delta_i^-, & if\ w(u) < 0 \\ [w_i^+(\max(unknown_i^i)) + w(u)] - \delta_i^+, & if\ w(u) > 0 \end{cases}$$

6.4.2 Server-Server Bound Redistribution for Commit Criteria Relaxation

An alternative to pushing updates to refresh client caches is to *relax local bounds by tightening remote bounds*. This is efficiently and practically accomplished by a pair-wise weight redistribution mechanism: a server willing to increase its local weight, thereby relaxing its local bound, contacts other servers and requests some amount of *bound* weight. The contacted server computes the amount it can give away and responds with that amount. In effect, the weights of the contacted servers, and therefore their bounds, are redistributed between them. This operation is light-weight in that *only* two servers are involved, and since the total amount of weight in the system remains fixed, correctness of the protocol is not affected (provided that the responding server computes the response amount properly, which we discuss below).

Given an update u that cannot be committed locally at s_i, the minimum amount of extra bound weights for the lower and upper bounds, $required\_w_i^-$ and $required\_w_i^+$, s_i requires in order to commit u when using the Share-Bound algorithm is:

$$\begin{pmatrix} required\_w_i^- \\ required\_w_i^+ \end{pmatrix} = \begin{cases} \begin{pmatrix} \dfrac{w(\min(unknown_i^i)) + w(u)}{\delta^- - \sum_{j \neq i} w(\min(unknown_j^i))} - w_i^- \\ 0 \end{pmatrix}, & if\ w(u) < 0 \\ \begin{pmatrix} 0 \\ \dfrac{w(\max(unknown_i^i)) + w(u)}{\delta^+ - \sum_{j \neq i} w(\max(unknown_j^i))} - w_i^+ \end{pmatrix}, & if\ w(u) > 0 \end{cases}$$

The corresponding value for the Partition-Bound algorithm is:

$$\begin{pmatrix} required\_w_i^- \\ required\_w_i^+ \end{pmatrix} = \begin{cases} \begin{pmatrix} \dfrac{w(\min(unknown_i^i)) + w(u)}{\delta^-} - w_i^- \\ 0 \end{pmatrix}, & if\ w(u) < 0 \\ \begin{pmatrix} 0 \\ \dfrac{w(\max(unknown_i^i)) + w(u)}{\delta^+} - w_i^+ \end{pmatrix}, & if\ w(u) > 0 \end{cases}$$

Note that the required bound weight might be larger than 1.0 due to already committed updates. Since the total bound weights held by all servers are fixed at 1.0, it may not be possible to commit the new update solely by bound redistribution. In such a case, client caches need to be refreshed before committing new updates.

When a server s_i is contacted for bound redistribution, s_i computes the maximum lower and upper bound weights that it can give away, $extra\_w_i^-$ *and* $extra\_w_i^+$ as follows for the Share-Bound algorithm:

$$\begin{pmatrix} extra\_w_i^- \\ extra\_w_i^+ \end{pmatrix} = \begin{pmatrix} w_i^- - \dfrac{w(\min(unknown_i^i))}{\delta^- - \sum_{j \neq i} w(\min(unknown_j^i))} \\ w_i^+ - \dfrac{w(\max(unknown_i^i))}{\delta^+ - \sum_{j \neq i} w(\max(unknown_j^i))} \end{pmatrix}$$

For the Partition-Bound algorithm, the corresponding value is:

$$\begin{pmatrix} extra\_w_i^- \\ extra\_w_i^+ \end{pmatrix} = \begin{pmatrix} w_i^- - \dfrac{w(\min(unknown_i^i))}{\delta^-} \\ w_i^+ - \dfrac{w(\max(unknown_i^i))}{\delta^+} \end{pmatrix}$$

such that $0 \leq extra\_w_i^+ \leq w_i^-$ and $0 \leq extra\_w_i^+ \leq w_i^+$,

As we can see from the above equations, s_i may give away *only* a subset of its weight if some of the updates it already committed are unknown to the clients (i.e., $w(unknown_i^i) \neq 0$). Intuitively, the reason is that the sum of the weights of the un-

Server s_i:

1. Set $Q = \{s_i\}$
2. While (*local_read_quorum_criteria* (r, c, $[\delta_o^-, \delta_o^+]$, Q) not satisfied)
    i. Select a new quorum server $s_q \notin Q$
    / * based on some divergence control policy */
    ii. Pull from s_q those updates unknown to c
    iii. Set $Q = Q \cup \{s_q\}$
3. Push all updates unknown to c

Figure 32: Basic algorithm executed by server si to commit a one-time read requested by client c with divergence bounds $[\delta_o^-, \delta_o^+]$ (on a data item r)

known updates can be thought of as *consuming* some of the divergence range available to the server, making the corresponding amount of weight unavailable. If this weight were to be given away and then consequently used by another server, the total amount of weight in the system used at any one time might potentially exceed the fixed 1.0 value, thereby eliminating any global divergence bound guarantees.

6.5 Supporting One-Time Divergence Bounds

In this section, we describe how our algorithms support one-time divergence bounds specified by one-time read operations.

6.5.1 Basic Approach

In both algorithms, the server that received the read, say *s*, *pulls* unknown updates from a sufficiently large *quorum* of servers and *pushes* those updates to the client that issued the one-time read. More specifically, *s* needs to form a read quorum, $Q \subseteq S$, such that the remaining set of *non*-quorum servers, $NQ = S - Q$, can not commit up-

dates whose sum of weights will invalidate the limit set by the client. In such a case, *s* does not need to expand *Q* anymore by pulling unknown updates from non-quorum servers.

We depict the basic quorum formation algorithm executed by a server s_i to commit a one-time read submitted by client *c* in Figure 32. Note that if the specified divergence bounds are hard, then the quorum servers need to be locked until the a large quorum is formed and the read returns. Otherwise, i.e., if the bounds are soft, no locking needs to be performed as shown in Figure 32.

Assume that a client *c* wants to read an item *r* with a maximum divergence of $[\delta_o^-, \delta_o^+]$ (the subscript '*o*' indicates a one-time bound). Two scenarios are possible: either there are no registered divergence bounds on *r*; or there exist registered continuous divergence bounds, $[\delta^-, \delta^+]$, on *r*. In the former case, server *s* that received the read request has no option but to contact all servers (practically forming a read quorum $Q = S$), and pull all updates unknown to *c*. This is necessary because no continuous bounds are maintained in the system and, thus, there is virtually no limit on the sum of the weights of updates that can be committed by a even a single server.

In the latter case, *s* can exploit the fact that a continuous bound is being maintained in the system to more efficiently execute the read operation. Note that the case where the one-time bounds are more relaxed than the corresponding continuous bounds (i.e,, $[\delta, \delta] \subseteq [\delta_o^-, \delta_o^+]$) is trivial: client *c* can simply complete the read using the data on its cache because the data is already guaranteed to satisfy the divergence bounds.

6.5.2 Read Quorum Criteria for Share-Bound

Given already registered continuous bounds, $[\delta^-, \delta^+]$ and a one-time bound, $[\delta_o^-, \delta_o^+]$, server s_i using the Share-Bound algorithm decides that the read quorum, Q, is sufficiently large to guarantee the one-time bound if the following local conditions are satisfied:

1. $\sum_{k \notin Q} w_k^-[\delta^- - \sum_{q \in Q} w(\min(unknown_{k,q}^i))] - \sum_{k \notin Q} w(\min(unknown_{i,k}^i) \geq \delta_o^-$, and
2. $\sum_{k \notin Q} w_k^+[\delta^+ - \sum_{q \in Q} w(\max(unknown_{k,q}^i))] - \sum_{k \notin Q} w(\max(unknown_{i,k}^i) \leq \delta_o^+$

Intuitively, the left side of each condition computes, based on s_i's view, the sum of weights of the updates that can be committed by a *non*-quorum server s_k, summed over all non-quorum servers $s_k \notin Q$: For each such s_k, the formula computes the local bound of s_k (indicating the total weight of updates that can be committed by s_k) less the sum of weights of updates that s_k already committed. The result then gives, for each $s_k \notin Q$, the total weight of updates that can further be committed by s_k. Notice that the sum

$$\sum_{k \notin Q} w_k^+$$

in the formula (consider the upper bound case) cannot be directly computed by s_i, since, by definition s_k is not a quorum server and its bound weight may not be available to s_i. However, since the sum of bound weights across all servers is fixed to 1.0, s_i can compute the sum indirectly using the bound weights of quorum servers as:

$$\sum_{k \notin Q} w_k^+ = 1.0 - \sum_{q \in Q} w_q^+$$

6.5.3 Read Quorum Criteria for Partition-Bound

Given already registered continuous bounds, $[\delta^-, \delta^+]$ and a one-time bound, $[\delta_o^-, \delta_o^+]$, server s_i using the Partition-Bound algorithm decides that the read quorum, Q, is sufficiently large to guarantee the one-time bound if the following local conditions are satisfied:

1. $\sum_{k \notin Q} w_k^- \delta^- - \sum_{k \notin Q} w(\min(unknown_{i,k}^i)) \geq \delta_o^-$, and

2. $\sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} w(\max(unknown_{i,k}^i)) \leq \delta_o^+$

The left hand-side of each condition computes the sum of the weights of the updates that can be committed by the non-quorum servers by subtracting the weight of the unknown updates that are already committed by those servers from the total bounds available to the those servers. The sum

$$\sum_{k \notin Q} w_k^+$$

is calculated indirectly as in the case of the Share-Bound algorithm.

6.6 Correctness

In this section, we provide correctness proofs for the criteria that we used in our divergence control protocols.

6.6.1 Share-Bound: Continuous Bounds

At all times, the algorithm must preserve the global divergence bounds in the system. Formally, we show that the algorithm preserves the following inequalities at all times, for all $i = 1 \ldots n$:

1. $\sum_i (\delta_i^- - w(\min(unknown_i^i)) + \sum_i w(unknown_i^i) \geq \delta^-$

2. $\sum_i (\delta_i^+ - w(\max(unknown_i^i)) + \sum_i w(unknown_i^i) \leq \delta^+$

Without loss of generality, consider the left side of upper global bound constraint:

$$\sum_i (\delta_i^+ - w(\max(unknown_i^i)) + \sum_i w(unknown_i^i) \qquad (6.1)$$

The first summation gives the sum of the weights of the updates that can be committed locally, without remote communication (as computed by the local commit criteria), and the second summation gives the total weight of updates that are already committed across all servers. When we expand δ_i^+, Formula 6.1 equals:

$$\sum_i w_i^+ \delta^+ - \sum_i w_i^+ \sum_{j \neq i} w(\max(unknown_j^i)) - \sum_i w(\max(unknown_i^i)) + \sum_i w(unknown_i^i)$$

The following are true for all *i,j*=1…*n*:

1. $\sum_i w_i^+ \delta^+ = \delta^+$

   // because $\sum_i w_i^+ = 1.0$

2. $\sum_i w(\max(unknown_i^i)) \geq \sum_i w(unknown_i^i)$

   // because $w(\max(unknown_i^i)) \geq w(unknown_i^i)$ by definition

3. $\sum_i w_i^+ \sum_{j \neq i} w(\max(unknown_j^i)) \geq 0$

   // because $\max(unknown_j^i) \geq 0$ and $w_i^+ \geq 0$ by definition

Using the three equations above because to expand formula 6.1, we observe that formula 6.1 is always less than or equal to δ^+, preserving the upper global bound, and thus concluding the proof (note that the proof for the global lower bound is independent and analogous to that for the upper bound).

6.6.2 Share-Bound: One-Time Bounds

Without loss of generality, consider the upper bound condition presented in Section 6.5.2:

$$\sum_{k \notin Q} w_k^+ [\delta^+ - \sum_{q \in Q} w(\max(unknown_{k,q}^i))] - \sum_{k \notin Q} w(\max(unknown_{i,k}^i) \leq \delta_o^+ \qquad (6.2)$$

We show that the left side of formula 6.2 provides an upper bound on the sum of weights that can be committed by the non-quorum servers, given by the formula:

$$\sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} \sum_{j \neq k} w(\max(unknown_{k,j}^k)) - \sum_{k \notin Q} w(\max(unknown_{k,k}^k) \qquad (6.3)$$

When we expand the left side of condition 6.2, we get

$$\sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} \sum_{j \neq k} w(\max(unknown_{k,j}^k)) - \sum_{k \notin Q} w(\max(unknown_{k,k}^k) \qquad (6.4)$$

The following holds for all $i, k, q = 1 \ldots n$:

1. $w(\max(unknown_{k,k}^k) \geq w(\max(unknown_{i,k}^i)$, and

   // by view properties

2. $\sum_{k \notin Q} \sum_{j \neq k} w(\max(unknown_{k,j}^k)) \geq \sum_{k \notin Q} \sum_{q \in Q} w(\max(unknown_{k,q}^i))$

   // by view properties

Formula 6.3 therefore provides a lower bound on Formula 6.4 and, by transitivity, is smaller than or equal to δ_o^+, concluding the proof (note that the proof for the lower bound is analogous).

6.6.3 Partition-Bound: Continuos Bounds

As we discussed in Section 6.3.2, the following global divergence bounds need to be enforced at all times:

1. $\sum_i (\delta_i^- - w(\min(unknown_i^i)) + \sum_i w(unknown_i^i) \geq \delta^-$

// lower bound limit

2. $\sum_i (\delta_i^+ - w(\max(unknown_i^i)) + \sum_i w(unknown_i^i) \leq \delta^+$

// upper bound limit

Without loss of generality, consider the upper bound constraint stated above. When expanded, the left side equals:

$$\sum_i w_i^+ \delta^+ - \sum_i w(\max(unknown_i^i)) + \sum_i w(unknown_i^i) \qquad (6.5)$$

The following are true for all i=1…n:

1. $\sum_i w_i^+ \delta_i^+ = \delta^+$, and

// since $\sum_i w_i^+ = 1.0$

2. $\sum_i w(\max(unknown_i^i)) \geq \sum_i w(unknown_i^i)$

// since $w(\max(unknown_i^i)) \geq w(unknown_i^i)$ by definition

Using the above two formulas to expand Formula 6.5, we see that the value in Formula in Formula 6.5 is always smaller than or equal to δ^+, concluding the proof (note that the proof for the lower bound constraint is independent and analogous).

6.6.4 Partition-Bound: One-Time Bounds

Consider, without loss of generality, the upper bound criterion discussed in Section 6.5.3:

$$\sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} w(\max(unknown_{i,k}^i)) \leq \delta_o^+ \qquad (6.6)$$

As in the case of Share-Bound, we show that the left side of Formula 6.1 provides a conservative estimate of the actual sum of the weight of the updates that the set of *non*-quorum servers can commit, which can be computed as:

$$\sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} w(\max(unknown_{k,k}^k)) \quad (6.7)$$

The inequality

$$\sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} w(\max(unknown_{k,k}^k)) \leq \sum_{k \notin Q} w_k^+ \delta^+ - \sum_{k \notin Q} w(\max(unknown_{i,k}^i))$$

always holds because $\forall i$, $k = 1 \ldots n$,

$$w(\max(unknown_{k,k}^k) \geq w(\max(unknown_{i,k}^i)$$

By transitivity, therefore, the value of Formula 6.7 is always smaller than or equal to δ_o^+, which concludes the proof (note that the proof for the lower bound is analogous).

6.7 ReBound Architecture

This section briefly describes the architecture of a ReBound server. Figure 33 illustrates the basic server components:

- The *Server Manager* is in charge of coordinating the activities of the various components and implementing the basic server API that accepts updates and continuous/one-time reads to data items it maintains.
- The *Divergence Controller* implements the divergence bounding algorithms used by ReBound. In particular, it maintains a *bound list* that contains the nu-

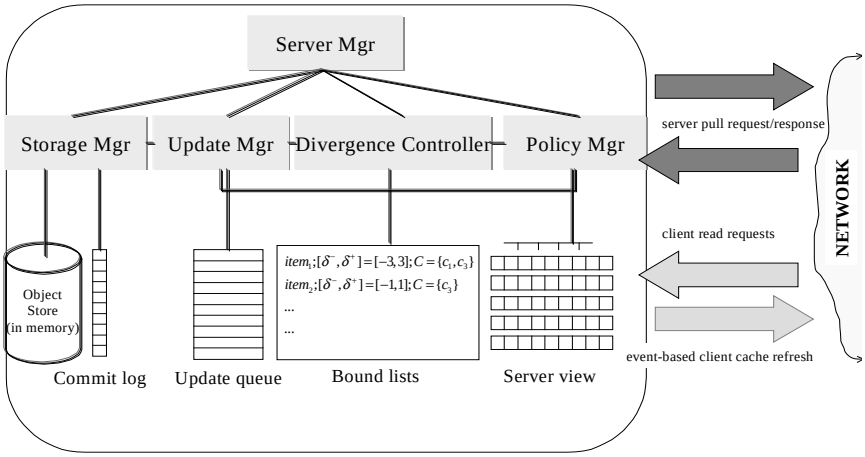

Figure 33: Basic ReBound server architecture

merical divergence bounds registered at the server, and a server *view* that compactly summarizes the committed updates propagated in the system.

- The *Policy Manager* is responsible for implementing efficient divergence control policies. This component implements different synchronization policies that specify when, with whom, and what data to pull or push.
- The *Update Manager* handles the local execution of updates. It maintains an update queue that contains all active (initiated but not-yet-committed) updates.
- The *Storage Manager* provides access to the *object store* that stores the committed versions of all replicated items. The object store is currently implemented as an in-memory database.

The current ReBound prototype runs on top of Linux and Windows32 platforms. All communication is made on top of UDP/IP.

6.8 Performance Evaluation

In this section, we describe the results of our experiments characterizing the performance of the algorithms using our *ReBound* prototype.

6.8.1 Experimental Environment and Methodology

Even though our protocols are designed for wide-area environments, we conducted our experiments on a local area network primarily for repeatability. Specifically, we performed our experiments on a cluster of 10 Linux machines, each having two 400 MHz Pentium II's, and 256 MBytes of memory. The machines are connected via a 100Mbps Ethernet network and communication is performed on top of UDP/IP. We, however, artificially injected a 100 milliseconds one-way latency to each outgoing message in order to emulate typical communication latencies over wide areas.

We present performance results for our approaches, *Share-Bound* (*SB*) and *Partition-Bound* (*PB*). In this initial performance study, we are mainly interested in the relative convergence characteristics of the different approaches, which are largely determined by the conservativeness of the commit criteria used by each algorithm. We study the general multiple server/multiple client configuration (with five servers and five clients), and characterize the overhead of maintaining both continuous and one-time divergence bounds. Although our results are based on hard divergence bounds, the results are also representative of the cost of enforcing soft bounds as well.

Notation	Description	Setting
UR	Mean global update generation rate	0.1,0.2 updates/s (uniform)
RR	Mean global read generation rate	0.1 reads/s (uniform)
$\|S\|$	Number of servers	5
$\|C\|$	Number of clients	5
$w(u)$	Weight of an update u	1
msg_latency	One-way message latency	100ms
bound_size	Continuous bound size: δ^- or δ^+ (both set equal)	[0,50]
bound_ratio	Ratio of one-time bound size to continuous bound size: δ_0^- / δ^- or δ_0^+ / δ^+ (both set equal)	[0, 1]

Table 4 : Primary experimental parameters and settings

In all the experiments we assume that the database consists of a single data item, and that the clients registered a single continuous divergence bound to the servers. We assume that the lower and upper bounds are set equal (i.e., $\delta^- = \delta^+$). Unless otherwise stated, each server independently initiates updates based on a uniform *update rate*. We assume that an update u has unit weight, $w(u) = 1$. This setting understates the performance of our system because it eliminates the possibility of multiple updates offsetting each other's effects. Each client independently initiates one-time reads based on a uniform *read rate*, and submits the read to a uniformly randomly selected server. The one-time bounds are also assumed to be equal (i.e., $\delta_o^- = \delta_o^+$), and we use the variable *bound ratio* to define the ratio of the one-time bounds to the continuous bound (i.e., δ_o^+ / δ^+). The main experimental parameters and settings are shown in Table 4.

The primary performance metrics we use are:

- *Commit latency*, which indicates the time between the initiation of an update or one-time read operation and the time the operation commits;
- *Local commit ratio*, which indicates the ratio of all committed updates that are committed at their initiating server without the need for any remote communication; and
- *Read quorum size*, which indicates the number of servers contacted to satisfy the bounds specified by a one-time read (including the server that initially received the read request).

In the experiments, we employed a simple cost-based policy that uses only compulsory push and pull, which a client request cannot be committed locally. Our policy does not make pro-active push and/or pull decisions such as background update propagation and view advance, which would significantly improve overall system performance.

When an update cannot be committed locally, there are essentially two alternative mechanisms our policy can employ: server-client push or server-server pull (as described in Sections 6.4.1 and 6.4.2, respectively). Unless otherwise stated, our policy uses a simple cost function that assumes that a server-server pull operation is much more expensive than a server-client push, thereby forcing the system to always favor server-client pushes. The reason why this cost function makes sense in our setting is that all the messaging latencies are the same, all servers are available, and the push operation is guaranteed to advance the server's view, whereas the server-server pull is not. The performance utility of server-server pull will be investigated in future work (see Section 7.2.2).

The policy used to select new servers to add to the read quorum when handling one-time reads is also straightforward. In this case, all servers are assumed to have the same cost of being accessed, so the server randomly chooses a new server to add to the read quorum when it decides that the one-time bound cannot be satisfied with the existing quorum (as described in Section 6.5). The performance of our algorithms can significantly be improved by designing and using more intelligent policies, which is outside the scope of this thesis.

In the rest of the section, we first investigate the overhead of maintaining continuous bounds in terms of the latency of committing an update at servers. We then present results describing the cost of reads that specify one-time bounds. We finally characterize the potential performance benefits that adaptive weight redistribution provides. The result presented in the following graphs are the averaged results of ten independent runs of executing 500 updates/reads in the system. The contributions of the first 50 updates/reads are excluded to eliminate system *warm-up* effects.

6.8.2 Enforcing Continuous Divergence Bounds

Figure 34 shows the commit latency results of our algorithms and a decentralized write-all type strict consistency protocol, labeled SC, as a function of bound size. SC is a conventional write-all [12] type of protocol that pushes an update to all clients prior to committing the update.

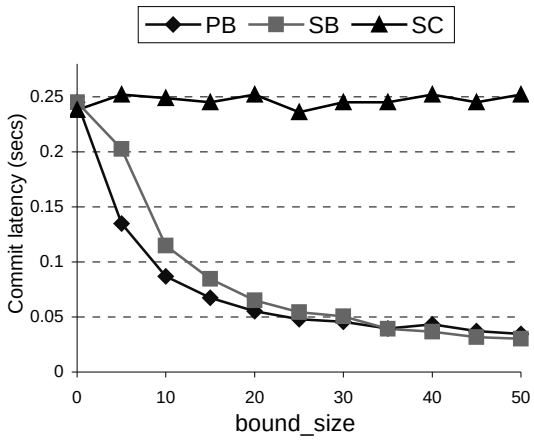

Figure 34: Commit latency vs. continuous bound size (UR=0.2, RR=0.0)

We begin by observing that our algorithms demonstrate the same write-all behavior when the local bounds at each server are sufficiently small. That is the reason why the three algorithms, SB, PB, and SC, converge at a bound size of 5, below which none of the servers can locally commit any updates. We can see that our divergence control algorithms have lower latency than the strict consistency protocol, and the gap increases significantly with increasing bound size. Clearly, the improvement comes at the expense of data accuracy. The figure also reveals that our bound partitioning algorithm achieves lower latency values than the bound sharing algorithm, especially for, relatively small-moderate bound sizes. The difference between the PB and SB curves quantifies the conservativeness of SB's local commit criteria relative to that of PB.

Figure 35 provides further insight by plotting the corresponding local commit ratio curves. Local commit ratio, as described earlier, is the ratio of updates that are committed without resorting to any remote communication (i.e., based solely on information locally available at the initiating server). It is evident that SC cannot commit any updates locally as it requires pushing updates to all clients first. Our bounding algo-

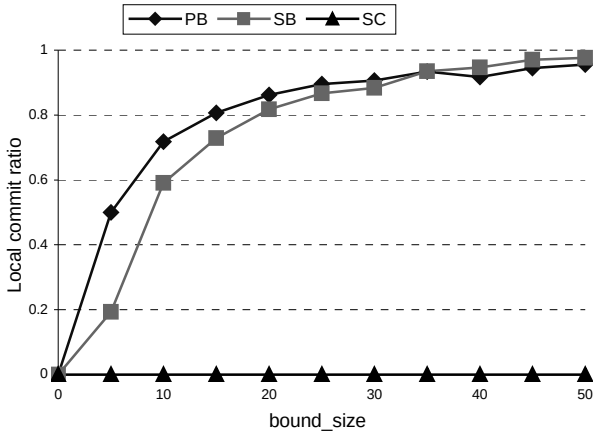

Figure 35: Local commit ratio vs. continuous bound size (UR=0.2, RR=0.0)

rithms, on the other hand, can commit updates locally most of the time, and local commit ratio increases with increasing bound size. PB commits more updates locally than SB does due to its less conservative local commit criteria, which essentially provides more server autonomy.

6.8.3 Enforcing One-Time Divergence Bounds

We now turn our attention to reads that specify one-time divergence bounds. In this case, the commit latency refers to the time between the submission of a read to a server and the time the server gathers accurate enough information to satisfy the divergence bounds, $[\delta_o^-, \delta_o^+]$, specified by the read. For purposes of this experiment, we define *bound ratio* to be the ratio of the one-time bound to the continuous bound as $\delta_o^- / \delta^- = \delta_o^+ / \delta^+$, where $[\delta^-, \delta^+]$ is the continuous bound already registered to the system. As discussed earlier, if no continuous bound are registered or if the one-time bounds do not indicate a sub-range of the continuous bounds, the server that accepted

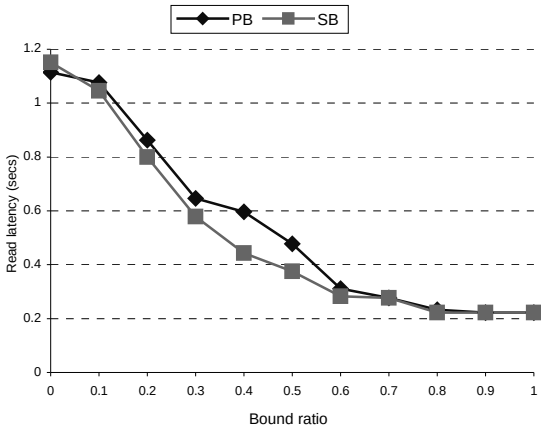

Figure 36: Commit latency vs. one-time bound ratio (UR=0.1, RR=0.1, bound_size=10)

the read request needs to pull the unknown updates from all other servers (i.e. form a read quorum containing all servers).

Figure 36 presents the commit latency results for increasing one-time bound ratio for PB and SB. As expected, the latency increases for both algorithms as the bound ratio decreases (i.e., as the read requires more accuracy). Unlike the results presented in Section 6.8.2, the bound sharing algorithm consistently achieves lower latency than the bound partitioning algorithm. In fact, the conservativeness of SB's local commit criteria, which has negative affect on performance when supporting continuous bounds, helps SB in this case as the non-quorum servers also use the same conservative criteria to compute their local ranges. Since PB's criteria is less conservative, the sum of the weights of the unknown updates that can be committed by the non-quorum servers can be more than that in SB, requiring more servers to be contacted and included in the read quorum for the divergence bound to be ensured.

Figure 37 shows the local commit ratio results for PB, SB, and a hypothetical variant of our bound partitioning algorithm, *static-quorum*. Static-quorum is similar to PB

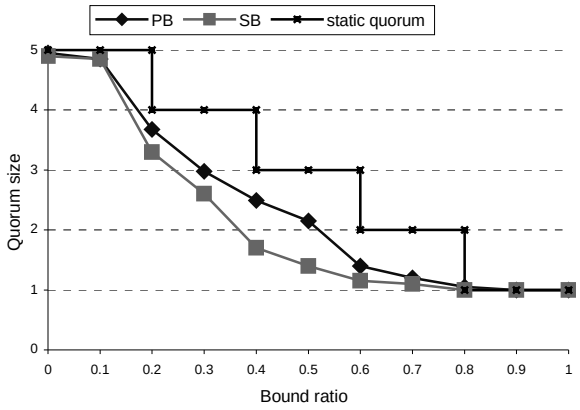

Figure 37: Read quorum size vs. one time bound ratio (UR=0.1, RR=0.1, bound_size=10)

in that the global registered bounds are partitioned across all servers, but differ in that servers do not utilize information regarding the updates seen from other servers. As a result, the curve for static-quorum follows a staircase-like shape. We observe that SB commits more updates locally than PB, thanks to its more conservative commit criteria. The difference between the curves for PB and variant static-quorum quantifies the benefits of exploiting views in this case. Note that, due to lack of space, we do not consider space and bandwidth consumption in this study and that the better performance comes at the cost of increased overhead in space and bandwidth while, respectively, maintaining and propagating view information.

6.8.4 Adaptation through Server-Server Bound Redistribution

Having discussed the base performance characteristics of our bounding algorithms, we now investigate the potential performance improvements attainable through adaptation, which is efficiently enabled through our server-server weight redistribution mechanism.

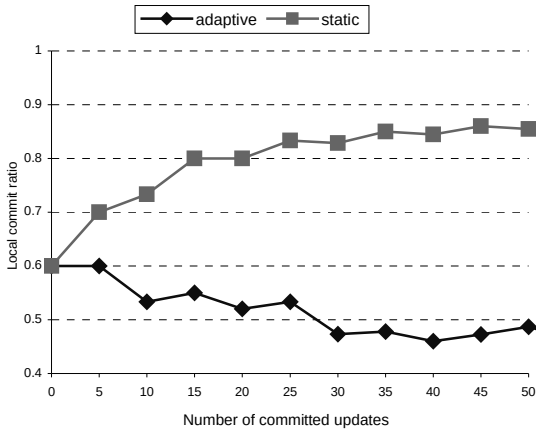

Figure 38: Update-rate based adaptation
(UR=0.2, RR=0.0, bound_size=10)

In this experiment, we consider a scenario of update-rate based adaptation during which the system adapts to dynamically changing update rates by redistributing server weights accordingly. We study an extreme-case scenario where there is an abrupt shift in the update rate distribution, which is initially uniform across all servers. After the shift, the update rate distribution is strictly biased towards a single *hot* server, which starts to initiate all the new updates. We then investigate how adaptive and non-adaptive versions of our algorithms are affected by such a non-trivial update rate shift. Note that, for purposes of this experiment, we use a cost function that always favors server-server weight redistribution over server-client update push.

Figure 38 shows how performance changes immediately after an extreme shift for the adaptive and *non*-adaptive versions of PB (the results for SB are similar) by plotting the local commit ratio as new updates are committed in the system. Weight redistribution is enabled in the adaptive version and disabled in the non-adaptive version of PB. We observe that the adaptive algorithm quickly reacts to the update shift and adapts to the new non-uniform update rate distribution. Unlike the adaptive algorithm,

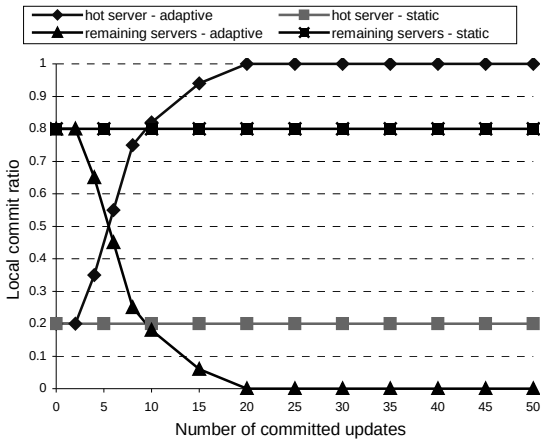

Figure 39: Redistribution of weights during adaptation (UR=0.2, RR=0.0, weight_skew=1.0, bound_size=10)

the non-adaptive algorithm cannot adapt and since its weight distribution does not anymore match the update rate distribution after the shift, its performance degrades quickly.

We explore the mechanics of weight redistribution in detail in Figure 39, which plots the weight of the *hot* server, and the sum of the bound weights of the remaining servers for both adaptive and non-adaptive algorithms. Initially, weight is uniformly distributed across all servers. After the shift, the hot server quickly increases its weight by automatically pulling weights from other servers in order to increase the size of its local divergence range as explained in Section 6.4.2. System-wide weight distribution then quickly converges to the new update rate distribution in the adaptive case. Even though the weights are redistributed in a pair-wise fashion, it has been shown that such a pair-wise weight redistribution can be used to incrementally converge to any desired target weight distribution exponentially fast [17].

6.9 Summary

We presented ReBound, a middleware layer that provides a general, flexible framework for efficiently supporting mobile and wide-area applications that do not require strict consistency of shared numerical data. ReBound generalizes previous work on numerical divergence bounding by enabling a set of clients to specify continuous/one-time, soft/hard divergence bounds on a set of servers. We argue the generality of our model by discussing how several representative mobile and wide-are applications can express their semantic numerical divergence requirements using our model.

We proposed two decentralized server-side algorithms, Share-Bound and Partition-Bound, for efficiently controlling numerical divergence of replicated data. Both algorithms maintain continuous divergence bounds, which are registered to servers, by limiting the sum of the weights of the committed updates that are unknown to clients. In Share-Bound, the registered bounds are *shared* among servers, whereas in Partition-Bound, the registered bounds are *partitioned* across servers. For both algorithms, we presented local update commit criteria that, if satisfied, ensure that the registered bounds will still be met after the commitment of an update. If a server's local update commit criteria are not satisfied, then the server effectively has to relax its criteria either by pushing updates to clients, or by tightening remote criteria. The algorithms handle one-time bounds by exploiting the already registered continuous divergence bounds, if available, to efficiently select a proper subset of servers whose unknown updates need to be pushed to the clients.

Using our prototype system, we performed a preliminary evaluation of the ReBound algorithms under various workloads and scenarios. The results revealed that the Parti-

tion-Bound algorithm, due to its less conservative local commit criteria, performs superior to the Share-Bound algorithm in terms of latency and local commit ratio when maintaining continuous bounds. Due to the same reason, interestingly, Partition-Bound performs *worse* than Share-Bound when handling one-time bounds. We also investigated the adaptive capabilities of our algorithms, and demonstrated that our server-server bound redistribution mechanism efficiently facilitates adaptation to changing factors such as update-rate distribution, and that adaptation has the potential to yield significant performance gains in real applications.

Chapter 7 Conclusions

In this thesis, we investigated the issue of efficient, consistent access to replicated data for distributed applications that run on top of mobile and wide-area environments. Based on the observation that existing replication protocols are typically ill suited for such environments because of implicit assumptions of high availability, strong connectivity, and static environmental and application-specific characteristics, we proposed and evaluated protocols that eliminate these restrictions by combining decentralized commitment, peer-to-peer information propagation, and light-weight adaptation mechanisms.

In particular, our work addressed data replication and consistency for the following applications and environments:

- In order to effectively support distributed applications that require access to shared replicated data in mobile and weakly-connected environments, we built Deno, a new decentralized object-replication system that provides formal strong consistency guarantees without sacrificing availability. Deno servers communicate using pair-wise, epidemic information flow, thereby making minimal demand from the underlying communications infrastructure. Deno also incorporates light-weight mechanisms that enable efficient adaptation to changing environmental and application-specific factors.
- In order to provide effective support for a class of inventory-based commodity sales and distribution applications in wide-area environments, we studied new token redistribution heuristics, and characterized the performance of data partitioning and replication-based approaches that address these applications.

- In order to provide effective support for a class of distributed wide-area applications that can tolerate and benefit from a bounded inconsistency in the data they observe, we proposed ReBound, an adaptable middleware system for efficient divergence control of numerical replicated data that can be updated at multiple wide-area locations. We proposed and evaluated new decentralized protocols that efficiently bound continuos and one-time numerical divergence constraints specified by clients.

The remainder of this chapter summarizes the main results and outlines future research directions for each study.

7.1 Decentralized Replication in Mobile and Weakly-Connected Environments

7.1.1 Summary of Results

We described the Deno decentralized object-replication system, focusing on its replication framework and security extensions. Deno's decentralized replication protocols are based on the combination of a new asynchronous protocol called *bounded weighted voting* and pair-wise epidemic information flow. This combination retains the advantages of current asynchronous protocols, but generally performs better, has higher availability, and provides stronger consistency guarantees.

The unique feature of *bounded* voting is that the total weight in the system is fixed at a system-wide, globally-known value. The advantage of a static total is that servers can determine when a plurality or majority of the votes have been accumulated *without complete knowledge of group membership*. This last attribute is key in dynamic,

distributed environments because it allows the protocol to operate in a completely decentralized fashion, eliminating performance bottlenecks and single points of failure.

Deno's replication framework supports two levels of consistency: a weak consistency level where update transactions are serializable and queries always access transactionally-consistent database states, and a strong consistency level that ensures globally serializable executions by providing a unique global commit order on all update transactions. We prove the correctness of both protocols and that neither suffers from local or global deadlocks.

We built a Deno prototype system that implements the described protocol. The current prototype runs on top of Linux and Win32 platforms. Using the prototype system (and a detailed simulation model), we characterized the performance of our protocols. The performance data yielded three main findings. The overriding motivation for Deno's protocols was to be able to make progress in weakly-connected environments. Protocols designed for such environments must make a number of tradeoffs that achieve availability at the possible expense of performance. Our first finding was that this performance impact was less than expected. On average, Deno servers learn of transaction commits almost as fast as a less available primary-copy protocol.

Second, our stronger consistency protocol performs nearly as good as the base, weak consistency protocol, while providing significantly stronger semantics. The lack of a performance advantage for a protocol with weaker semantics and fewer restrictions on update commits is perhaps one of our most surprising findings. The result is increased functionality at essentially little cost in performance.

Finally, speculative update propagation and voting provides a considerable performance advantage for protocols that use pair-wise, epidemic communication, and this advantage is magnified when application-specific commutativity information is utilized.

We also addressed the security issues that arise when deploying peer-to-peer decentralized databases and described the first treatment and evaluation of secure update commitment protocols for such systems in the context of Deno. We focused our work on a specific class of internal attacks and proposed a flexible, parameterized protocol that allows servers to set arbitrary degrees of tolerance to malicious insiders. A unique feature of our approach is that it allows servers to trade off performance and the degree of tolerance to malicious insiders. Our protocols provide this flexibility without adversely impacting the performance of other servers, allowing each server to set arbitrary degrees of tolerance based on its individual requirements and resources. We also described the conditions under which our protocols provide liveness and safety properties.

We evaluated the cost of our security extensions using the Deno prototype. Our main results revealed that protecting against internal threats comes at a cost, but the marginal cost for protecting against larger cliques of malicious insiders is generally low; and that our decentralized protocol performs, scales, and handles update contention significantly better than a ROWA-based decentralized protocol.

7.1.2 Future Research Directions

There are many challenging avenues for future research in replicated data management in mobile and weakly-connected environments. We now briefly summarize some of the more promising directions in the context of Deno.

Synchronization policies

One significant advantage of Deno's decentralized replication protocol is that the correctness of the protocol does not depend on what servers synchronize and when they synchronize. The particular choice of a synchronization policy, therefore, is a performance/cost issue, rather than a correctness issue. A synchronization policy needs to consider a variety of environmental factors such as the available bandwidth, cost of communication, server availability, and weight information as well as application-dependent factors such as update generation rate and commutativity ratio between updates. Furthermore, since many of the mentioned factors typically demonstrate dynamic behavior, adaptive policies are required. For instance, a mobile server that initiates synchronization sessions infrequently to save power or reduce its communication costs when running on batteries or using a wireless medium may significantly increase its synchronization rate when well connected. An interesting research direction would be to explore different server synchronization policies and characterize their effects on overall system performance and cost.

Dynamic weight redistribution

Previous work in voting defined and used weight mainly for system availability. Weight redistribution is not only useful in improving availability, but it can also be

used to increase performance. Consider a large-scale system in which most of the updates are performed by only a small subset of the servers. Allocating the weight at those servers will potentially increase throughput and decrease response times since the number of servers that need to be contacted for committing an update is effectively reduced. In the extreme case where all (or at least half) the weight is allocated at a single server, the system will resemble a dynamic primary-copy scheme.

Unlike previous voting protocols that require the servers to have complete knowledge of system membership and the amount of weight at servers, Deno can work with incomplete information because of the bounded weight in the system. This feature enables Deno servers to perform light-weight, dynamic weight exchanges, that require the participation and the knowledge of only two servers. An interesting research direction would be to investigate the utility of dynamic weight reallocation as the workload in the system changes.

Hierarchical organizations

In this thesis, we assumed a flat organization of Deno servers. As with any distributed system, flat organizations, especially peer-to-peer models, suffer from low scalability. Significant performance improvements can be attained when a hierarchical synchronization topology is used to organize servers.

7.2 Decentralized, Semantics-based Data Consistency Protocols in Wide-Area Environments

7.2.1 Summary of Results

We first addressed a class of newly emerging Internet-based e-commerce applications that involve globally distributed commodity sales. These applications can be effectively supported by representing commodities using *tokens* and by redistributing tokens in the system based on demand. We experimentally evaluated and compared two fundamentally different approaches to token distribution —partitioning and replication— using a detailed simulation model and real Internet message traces. We also proposed several decentralized pair-wise token redistribution strategies for the partitioning-based approach and evaluated them under various workloads.

Our experimental evaluation revealed a number of significant results for token-based commodity sales and distribution. First, replication-based approaches are neither necessary nor desirable for the kinds of applications and environment we addressed: partitioning-based approaches performed and scaled better primarily due to their ability to provide higher server autonomy. Second, the use of information about the system state turns out to be crucial for making token redistribution decisions.

We then extended our discussion and addressed a more general class of distributed wide-area applications by introducing ReBound, a system that provides a general, flexible numerical divergence framework for efficiently supporting distributed wide-area applications that can tolerate and benefit from bounded divergence in the data they access.

ReBound generalizes previous work on numerical divergence bounding by enabling a set of clients to specify continuous/one-time, soft/hard divergence bounds on a set of servers. We proposed two decentralized server-side algorithms, Share-Bound and Partition-Bound, for efficiently controlling numerical divergence of replicated data. Both algorithms maintain continuous divergence bounds, which are registered to servers, by limiting the sum of the weights of the committed updates that are unknown to clients and by refreshing client caches as necessary. In Share-Bound, the registered bounds are *replicated* among servers, whereas in Partition-Bound, the registered bounds are *partitioned* across servers (in fact, these protocols can be regarded as generalizations of the protocols we studied for token partitioning and replication).

Using the ReBound prototype, we performed a preliminary evaluation of our algorithms under various workloads and scenarios. Initial results verified the performance benefits that can be attained by exploiting the data performance vs. precision tradeoff enabled by ReBound. The results revealed that the Partition-Bound algorithm, due to its less conservative local commit criteria, performs superior to the Share-Bound algorithm in terms of latency and local commit ratio when maintaining continuous bounds. Due to the same reason, interestingly, Partition-Bound performs worse than Share-Bound when supporting one-time bounds. We also investigated the adaptive capabilities of our algorithms and demonstrated that our server-server bound redistribution mechanism efficiently facilitates adaptation to changing factors such as update-rate distribution and that adaptation has the potential to yield significant performance gains in real applications.

7.2.2 Future Research Directions

We now outline several promising future research avenues that extend and further our research in semantics-based distributed data consistency protocols for wide-area applications and environments.

Hierarchical server organizations for token-based commodity distribution

In this thesis, we only investigated purely peer server organizations where there is no hierarchy among the servers in the system. Such a flat organization is attractive for several reasons including its simplicity and the flexibility it enables when choosing synchronization partners. Unfortunately, as our results demonstrated, this organization is not scalable especially when there are only a few tokens are left in the system. Hierarchical server organizations might be especially beneficial under such conditions and would significantly increase the scalability of the system.

Divergence control policies

In this thesis, we described efficient divergence control *mechanisms* in the context of ReBound, but did not address any *policy* issues. A divergence control policy needs to provide answers to the following questions:

1. When to contact other nodes?
2. Which nodes to contact? and
3. What updates to push/pull?

The mechanisms we described are designed so that the particular policy choice does not affect protocol correctness; only performance is affected. This flexibility allows numerous opportunities for developing highly efficient policies that take into account

a variety of environmental factors such as the available bandwidth, cost of communication, server and client availability, and bound information, as well as application-dependent factors such as update and read generation rates. An challenging research problem involves devising and evaluating *cost-based* divergence control policies where synchronization overheads are expressed in terms of low-overhead, *parameterized* cost functions.

Supporting divergence guarantees for aggregation queries over replicated data

In the ReBound work that we described in this thesis, we addressed the problem of providing precision guarantees to simple read operations involving single data items. A natural extension of this work involves providing guarantees for more complicated data access operations that involve aggregation queries such as sums, averages, etc. Even though this problem is addressed in centralized databases [54], no solution exists for symmetric replicated databases we address.

Proxy-based, hierarchical organizations for scalability

Scalability might become a crucial issue in a scenario where a large number of clients register a large number of divergence constraints. A proxy-based hierarchical organization might go a long way in effectively scaling up to large number of clients and divergence constraints. The basic idea is to have a proxy node sit between the individual clients and the servers.

In such a proxy-based scenario, the proxy serves as an intermediary between the clients and the servers: clients communicate only with the proxy and the proxy communicates with the servers and the clients. The proxy needs to represent the overall client

population by registering appropriate aggregate divergence bounds. One challenging question here is to decide, given a set of clients and their individual data requirements, how to efficiently set the divergence constraints at the proxy so as to minimize the interaction of the proxy with the servers.

BIBLIOGRAPHY

[1] CSIM 18 Simulation Engine Manual (C++ version). Mesquite Software, Inc.

[2] D. Agrawal and A. E. Abbadi. An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion. *ACM Transactions on Computing Systems*, 9(1):1-20, 1991.

[3] D. Agrawal and A. E. Abbadi. Integrating Security with Fault-Tolerant Distributed Databases. *The Computer Journal*, 33(1):71-78, 1990.

[4] D. Agrawal, A. E. Abbadi, and R. Steinke. Epidemic Algorithms in Replicated Databases. In *Proc. 16th ACM Symp. on Principles of Database Systems (PODS)*, Tucson, May 1997.

[5] M. Ahamad and M. H. Ammar. Multidimensional Voting. *ACM Transactions on Computing Systems*, 9(4):399-431, 1991.

[6] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems (TODS)*, 15(3):359-384, 1989.

[7] R. Alonso and H. F. Korth. Database System Issues in Nomadic Computing. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Washington, DC, May 1993.

[8] Y. Amir and A. Wool. Optimal Availability Quorum Systems: Theory and Practice. *Information Processing Letters*, 65:223-228, April 1998.

[9] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Linear Arithmetic Constraints in Distributed Database Systems. In *Proc. of the Intl. Conf. on EDBT*, 1992.

[10] D. Barbara and H. Garcia-Molina. Optimizing the Reliability Provided by Voting Mechanisms. In *Proc. of the International Conf. on Distributed Computing Systems*, San Francisco, October 1984.

[11] D. Barbara, H. Garcia-Molina, and A. Spauster. Increasing Availability Under Mutual Exclusion Constraints with Dynamic Voting Assignment. *ACM Transactions on Computing Systems*, 7(4):394-426, 1989.

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

[13] P. Bober and M. Carey. Multiversion Query Locking. In *Proc. 18th Conf. on Very Large Databases (VLDB)*, Vancouver, 1992.

[14] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silbershatz. Update Propagation Protocols for Replicated Databases. In *Proc. ACM Int. Conf. on Management of Data (SIGMOD)*, Philadelphia, 1999.

[15] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast Security: A Taxonomy and Efficient Constructions. In *IEEE Conf. on Computer Communications (INFOCOM)*, 1999.

[16] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. Third Symp. on Operating Systems Design and Implementation (OSDI)*, New Orleans, 1999.

[17] U. Cetintemel and P. J. Keleher. Light-Weight Currency Management Mechanisms in Deno. In *Proc. 10th IEEE Workshop on Research Issues in Data Engineering (RIDE)*, San Diego, February 2000.

[18] U. Cetintemel, P. J. Keleher, and M. J. Franklin. Support for Speculative Update Propagation and Mobility in Deno. University of Maryland, UMIACS-TR-99-70, Oct. 29, 1999.

[19] U. Cetintemel, B. Ozden, M. J. Franklin, and A. Silberschatz. Partitioning vs. replication for token-based commodity distribution. CS-TR-4180, UMIACS-TR-2000-62, University of Maryland, 2000.

[20] S. Y. Cheung, M. H. Ammar, and A. Ahamad. The Grid protocol: A High Performance Scheme for Maintaining Replicated Data. In *IEEE Int. Conf. on Data Engineering*, Los Angeles, California, May 1990.

[21] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):585-600, 1998.

[22] L. Cranor and R. Cryton. Sensus: A security-conscious electronic polling scheme for the Internet. In *Hawii International Conference on System Sciences*, 1997.

[23] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th ACM Symp. on Principles of Distributed Computing (PODC)*, Vancouver, 1987.

[24] L. W. Dowdy and D. V. Foster. Comparative Models of the File Assignment Problem. *ACM Computing Surveys*, 14(2):287-313, 1982.

[25] A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large-scale elections. In *Advances in Cryptology --- AUSCRYPT'92, Lecture Notes in Computer Science*, 1992.

[26] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database System. *ACM Transactions on Database Systems*, 7(2):209-234, June 1982.

[27] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. 7th ACM Symp. on Operating Systems Principles (SOSP)*, Pacific Grove, 1979.

[28] L. Golubchik and A. Thomasian. Token Allocation in Distributed Systems. In *Proc. of the Int. Conf. on Distributed Computing Systems*, Yokohama, 1992.

[29] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc.1996 ACM Intl. Conf. on Management of Data (SIGMOD)*, Montreal, June 1996.

[30] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*: Morgan Kaufmann, 1992.

[31] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, 1995.

[32] A. Gupta and J. Widom. Local Verification of Global Integrity Constraints. In *Proc. of the SIGMOD Conf.*, 1993.

[33] T. Haerder. Handling Hot Spot Data in DB-sharing Systems. *Information Systems*, 13(2):155-166, 1988.

[34] A. A. Helal, A. A. Heddaya, and B. B. Bhargava, *Replication Techniques in Distributed Systems*: Kluwer Academic Publishers, 1996.

[35] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi. Epidemic Quorums for Managing Replicated Data. In *Proc. 19th IEEE Intl. Performance, Computing, and Communications Conf. (IPCCC)*, Phoenix, 2000.

[36] W. Hsueush, G. E. Kaiser, C. Pu, K.-L. Wu, and P. S. Yu. Divergence Control for Distributed Database Systems. *Distributed and Parallel Databases*, 3(1):85-109, 1995.

[37] Y. Huang, R. H. Sloan, and O. Wolfson. Divergence Caching in Client Server Architectures. In *Proc. Int. Conf. Parallel and Distributed Information Systems (PDIS)*, 1994.

[38] N. Huyn. Speeding up View Maintenance Using Cheap Filters at the Warehouse. In *Proc. of Int. Conference on Data Engineering (ICDE)*, 2000.

[39] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Transactions on Database Systems*, 15(2):230-280, 1990.

[40] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and L. Greif. Replicated Document Management in a Group Communication System. In *Proc. Conf. on Computer Supported Cooperative Work*, 1988.

[41] P. Keleher and U. Cetintemel. Consistency Management in Deno. *The Journal on Special Topics in Mobile Networking and Applications (MONET). To Appear.*

[42] P. J. Keleher. Decentralized Replicated-Object Protocols. In *Proc. 18th ACM Symp. on Principles of Distributed Computing (PODC),* Atlanta, May 1999.

[43] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc.13th ACM Symp. on Operating Systems Principles (SOSP),* October 1991.

[44] N. Krishnakumar and A. Bernstein. Bounded Ignorance in Replicated Systems. In *Proc. 10th ACM Symp. on Principles of Database Systems (PODS),* 1991.

[45] N. Krishnakumar and A. J. Bernstein. High Throughput Escrow Algorithms for Replicated Databases. In *Proc. of the Int. Conf. on Very Large Databases,* Vancouver, Canada, 1992.

[46] A. Kumar. An Analysis of Borrowing Policies for Escrow Transactions in a Replicated Data Environment. In *Proc. of the IEEE Int. Conf. on Data Engineering,* Los Angeles, 1990.

[47] A. Kumar and A. Segev. Cost and Availability Tradeoffs in Replicated Data Concurrency Control. *ACM Transactions on Database Systems,* 18(1):102-131, March 1993.

[48] A. Kumar and M. Stonebraker. Semantics Based Transaction Management Techniques for Replicated Data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Chicago, USA, June 1988.

[49] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computing Systems*, 10(4):360-391, November 1992.

[50] P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases,* 8(1):7-40, 2000.

[51] D. Malkhi, Y. Mansour, and M. Reiter. On Diffusing Updates in a Byzantine Environment. In *Proc. 18th IEEE Symp. on Reliable Distributed Systems (SRDS),* Lausanne, 1999.

[52] D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *Proc. 29th ACM Symp. on Theory of Computing*, 1997.

[53] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms,* Amsterdam, 1989.

[54] C. Olston and J. Widom. Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data. In *Proc. of the 26th VLDB Conf.*, Cairo, Egypt, 2000.

[55] P. E. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4):405-430, 1986.

[56] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on Optimistically Replicated Peer-to-Peer Filing. *Software--Practice and Experience*, 28(2):155-180, February 1998.

[57] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM Symposium on Operating System Principles*, Saint-Milo France, October 1997.

[58] B. Pittel. On spreading a rumor. *SIAM Journal of Applied Math*, 47(1):213-223, 1987.

[59] C. Pu, W. Hseush, G. E. Kaiser, K.-L. Wu, and P. S. Yu. Distributed Divergence Control for Epsilon Serializability. In *Proc. of the Int. Conf. on Distributed Computing Systems*, Pittsburg, PA, USA, 1993.

[60] I. Ray, E. Bertino, S. Jajodia, and L. Mancini. An Advanced Commit Protocol for MLS Distributed Database Systems. In *Proc.3rd ACM Conf. on Computer and Communications Security*, New Delhi, 1996.

[61] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *2nd ACM Conference on Computer and Communications Security*, 1994.

[62] O. Rodeh, K. P. Berman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble Security. Cornell Univerisity TR-98-1703, 1998.

[63] N. Soparkar and A. Silberschatz. Data-value Partitioning and Virtual Messages. In *Proc. of the Symposium on Principles of Database Systems*, Nashville, 1990.

[64] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. of the Winter 1988 USENIX Conf.*, 1988.

[65] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5(3):188-194, May 1979.

[66] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in a Weakly Connected Replicated Storage System. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

[67] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180-209, 1979.

[68] A. Thomasian. Fractional Data Allocation Method for Distributed Databases. In *Proc. of the Int. Conf. on Parallel and Distributed Information Systems*, Austin, 1994.

[69] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on*

Operating Systems Design and Implementation, Monterey, CA, November 1994.

[70] A. Wool. Quorum Systems in Replicated Databases: Science or Fiction? *Bulletin of the Technical Committee on Data Engineering*, 21(4):3-11, 1998.

[71] K.-L. Wu, P. S. Yu, and C. Pu. Divergence Control for Epsilon-Serializability. In *Proc. of Int. Conf. on Data Engineering*, Tempe, USA, 1992.

[72] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, 2000.

[73] H. Yu and A. Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. In *Proc. of the 26th VLDB Conf.*, Cairo, Egypt, 2000.

[74] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 6(1):7-40, 1998.