Deno: A Decentralized, Peer-to-Peer Object-Replication System for Weakly Connected Environments

Ugur Çetintemel, *Member*, *IEEE*, Peter J. Keleher, Bobby Bhattacharjee, and Michael J. Franklin

Abstract—This paper presents the design, implementation, and evaluation of the replication framework of Deno, a decentralized, peer-to-peer object-replication system targeted for weakly connected environments. Deno uses weighted voting for availability and pair-wise, epidemic information flow for flexibility. This combination allows the protocols to operate with less than full connectivity, to easily adapt to changes in group membership, and to make few assumptions about the underlying network topology. We present two versions of Deno's protocol that differ in the consistency levels they support. We also propose security extensions to handle a class of malicious actions that involve misrepresentation of protocol information. Deno has been implemented and runs on top of Linux and Win32 platforms. We use the Deno prototype to characterize the performance of the Deno protocols and extensions. Our study reveals several interesting results that provide fundamental insight into the benefits of decentralization and the mechanics of epidemic protocols.

Index Terms—Data replication, epidemic protocols, peer-to-peer systems, weak consistency, voting.

1 Introduction

This paper describes the design, implementation, and performance of Deno [10], [13], [14], [25], a system that support object replication in a transactional framework for weakly connected environments. Deno's system model is illustrated in Fig. 1. One or more clients connect to each *peer* server, which communicates through pair-wise information exchanges. The servers are not necessarily *ever* fully connected. Deno's application domain include asynchronous groupware applications (e.g., Lotus Notes [24]), traditional file and database systems, and distributed middleware systems that require asynchronous consensus support.

Deno's underlying protocols are based on an asynchronous protocol called *bounded weighted voting* [25]. Asynchronous solutions for managing replicated data [7], [21], [24], [26] have a number of advantages over traditional synchronous replication protocols in large-scale 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 slow or require reconciliation or have low availability because they rely on primary-copy schemes [33].

The protocols retain the advantages of current asynchronous protocols, but generally perform better, have fewer connectivity requirements, and achieve higher availability. 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 protocol's strengths result from a combination of weighted voting and epidemic information flow [17], a process where information flows pair-wise through the system (much 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 [34]). 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., [20], [23], [35]), votes are cast and disseminated among servers *asynchronously* through pair-wise propagation. Servers commit or abort transactions locally and all servers eventually reach the same decisions.

The use of *voting* allows the system to have higher availability than primary-copy protocols [1], [4], [20], [30], [35]. The use of *weighted* voting allows implementations to improve performance by adapting currency (a.k.a. weight) distributions to site availabilities, update activity, or other relevant characteristics [9]. Each server has a specific amount of currency and the total currency 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

- *U. Çetintemel is with the Department of Computer Science, 115 Waterman St., Brown University, Providence, RI 02912. E-mail: ugur@cs.brown.edu.*
- *P.J. Keleher and B. Bhattacharjee are with the Department of Computer Science, A.V. Williams Bldg., University of Maryland, College Park, MD 20742-3255. E-mail: {keleher, bobby}@cs.umd.edu.*
- *M.J. Franklin is with the Electrical Engineering and Computer Science Department, Computer Science Division, University of California, Berkeley, CA 94720-1776. E-mail: franklin@cs.berkeley.edu.*

Manuscript received 26 July 2001; revised 9 May 2002; accepted 22 May 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 116613.

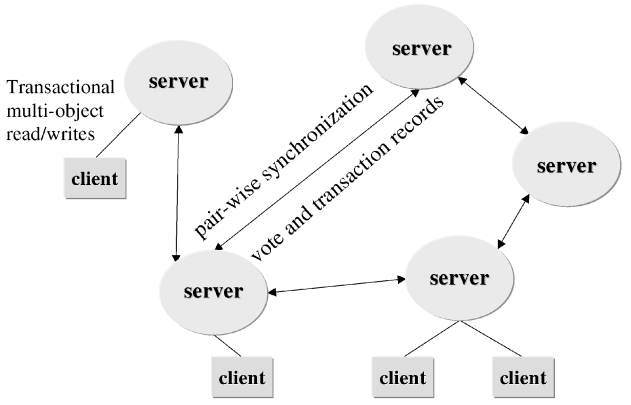

Fig. 1. Basic Deno system model.

the votes has 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 weakly connected environments, where individual nodes are routinely 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.

The key contributions of this paper can be summarized as follows: First, we present decentralized, peer-to-peer replication protocols that combine weighted voting and asynchronous, epidemic information flow. This combination enables our protocols to operate with less than full connectivity, easily adapt to changes in group membership, and make few assumptions about the underlying network topology. We describe in detail the consistency levels provided by the protocols and outline formal correctness proofs.

Second, we discuss an extension of our replication protocols to provide security against a specific class of internal security threats that involve misrepresentation of protocol-specific voting information. To handle such threats, we propose a vote validation technique that requires cryptographic primitives and modifications to the update commit criteria. A unique aspect of our solution is that it not only enables a trade off between performance and the degree of tolerance to malicious servers, but also allows for individual servers to support nonuniform degrees of tolerance without adversely affecting the performance of the rest of the system.

Third, we present the design of the Deno prototype system that implements the proposed protocols. The basic Deno architecture has been implemented and runs on top of Linux and Win32 platforms. We conduct a detailed performance study of our protocols and other comparable epidemic protocols that appeared in the literature using the Deno prototype. Our study provides fundamental insight into the benefits of decentralization and mechanics of epidemic protocols. One particularly interesting result of our evaluation is that the presumed performance advantage of the centralized approach over a decentralized voting approach is not significant with asynchronous, epidemic information flow.

The rest of this paper is structured as follows: Section 2 describes the basic Deno system model and the decentralized replication protocols employed by Deno in detail. Section 3 briefly addresses security issues and describes a security extension based on vote validation. Section 4 summarizes other Deno features that promote weakly-connected operation. Section 5 presents our prototype-based performance study that characterizes the performance of the replication protocols and their extensions as well as other competitive epidemic protocols. Section 6 briefly describes related work and Section 7 summarizes and concludes the paper. Finally, the Appendix provides correctness proofs for the proposed Deno protocols as well as pseudocode for basic Deno operations.

2 Decentralized Replication Protocols

Before delving into the fine detail, we give a quick overview of the *life* of a Deno transaction (Fig. 2 depicts the entire process in more detail). A transaction is submitted by a client to any server, which executes the transaction locally. Upon completion of the execution, the transaction either

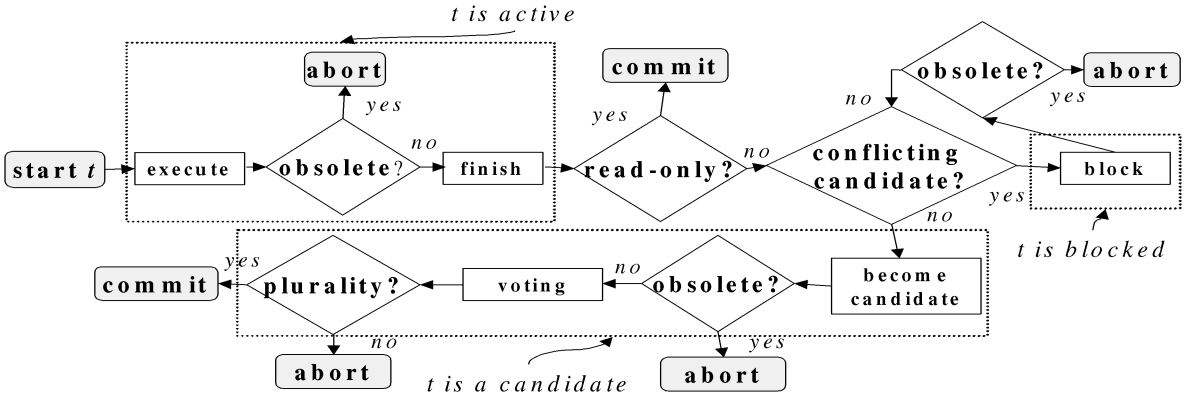

Fig. 2. A transaction's life in Deno.

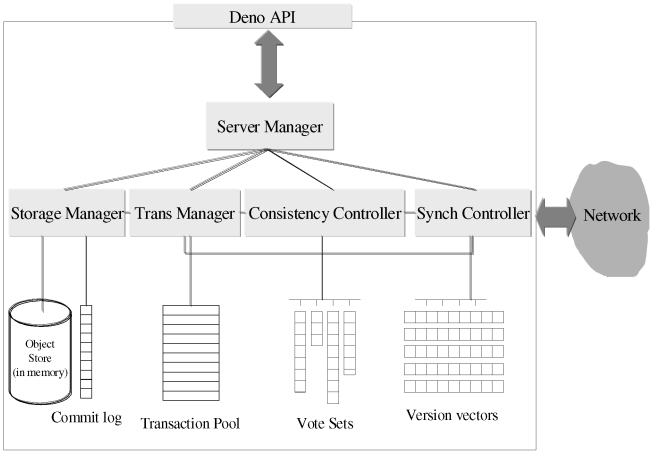

Fig. 3. Basic Deno architecture.

blocks (if the local server has seen a conflicting transaction) or becomes a *candidate* and participates in the election. Candidates are voted on and are eventually either committed (when/if they corner a plurality of the total system currency) or aborted.

2.1 Deno System Architecture

We now briefly describe the architecture of the Deno object replication system. The basic Deno API supports operations for creating objects, creating and deleting object replicas, and performing reads and writes on the shared objects in a transactional framework.

Fig. 3 illustrates the Deno server architecture. The *Server Manager* is in charge of coordinating the activities of the various components and handling client requests by implementing the Deno API. The *Consistency Controller* implements the decentralized voting protocols and maintains a *vote pool* that summarizes the votes known to the server. The *Synch Controller* implements efficient synchronization sessions with other Deno servers by maintaining *version vectors* that compactly summarize the events of interests. The *Trans Manager* handles the local execution of transactions. It maintains a transaction pool that contains all active transactions known to the server. The *Storage Manager* provides access to the *object store* that stores the current committed versions of all locally replicated objects. The object store is currently implemented as a simple in-memory database. The current prototype runs on top of Linux and Win32 platforms. Communication is made over IP using UDP or TCP.

2.2 Providing Weak Consistency: Base Protocol

We now describe the basic Deno protocol, describing in detail the primary data structures, the transaction and the synchronization models, and the processing rules. Appendix B provides a detailed pseudocode that describes the implementation of these models and rules in detail.

2.2.1 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*. Database states are tracked by associating a *version number* with each database item. The items in the local copy of the database are modified and their version numbers incremented only when update transactions commit.

We distinguish between *queries* (i.e., read-only transactions) and *update transactions*. Both types of transactions execute entirely locally. However, queries are light weight in that a query can commit immediately after it successfully finishes its execution. Update transactions, on the other hand, must participate in a distributed commitment process after finishing execution.

Each 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. When an active update transaction successfully completes its execution, it takes one of the following two paths: 1) The transaction can either become a candidate transaction at its local server and participate in a distributed voting process that determines whether it commits or aborts or 2) the transaction blocks and waits for the termination of other previous transactions before becoming a candidate. The blocked transactions are later reconsidered for becoming candidates.

2.2.2 Voting

We define V_i as the set of all votes seen 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.curr* denotes the amount of currency *v.voter* voted for *v.trans*, and *v.tstamp* is the value of *v.voter*'s local timestamp that is incremented each time the server casts a vote.

Two transactions are said to *conflict* if 1) their common read items have the same version numbers and 2) at least one of the transaction's read items overlaps with the other's update items.

A server, s_i, votes for a transaction by creating a vote, v, assigning a currency value to v, and inserting it into V_i. The currency value for a vote can be set in two distinct ways based on the state of the vote set. Server s_i votes all of its currency for transaction t_i if s_i 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 currency, in which case the vote is called a *no vote*. In the rest of the paper, when we speak about a vote without indicating its type, we imply a *yes* vote by default.

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

- a set of votes, V_i,
- a list of *candidate* transactions, C_i, consisting 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 list of *blocked* transactions, B_i, consisting of locally completed transactions waiting to become candidates,
- a commit log containing an ordered 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 the server 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 created, votes may not 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 currency in the system is *always* 1. 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*—s_i votes *yes* for t if s_i has not already voted for a conflicting transaction; otherwise, s_i votes *no*.
2. *When s_i commits or aborts a candidate transaction*—s_i considers all transactions t in the blocked list (i.e., all transactions waiting to become candidates) in insertion 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 C_i. Otherwise, s_i blocks t and inserts t into 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 *conflicting* candidate transactions.

2.2.3 Update Commitment

Given a server s_i and its vote set V_i, we compute the sum of votes cast for a transaction t as:

$$|votes(t)| = \sum_{v \in V_i \wedge v.trans=t} v.curr.$$

We then compute the unknown votes of t as:

$$unknown(t) = 1.0 - \sum_{v \in V_i \wedge v.trans=t \wedge v.voter=s} s.curr,$$

where $s.curr$ is the currency held by s. In other words, $unknown(t)$ is the sum of the currencies of those servers whose votes for transaction t are *not* yet available.

We now define the commit rule that s_i uses to decide which candidate transactions to terminate (i.e., commit or abort) on the basis of *local* information. The fundamental idea is to commit a transaction when it is guaranteed that no other conflicting transaction can gather more votes.

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
2. $|votes(t)| = (|votes(t')| + unknown(t))$ and $t.server < t'.server$,

where *t.server* is the identifier of the server that executed t.

The commit rule states that candidate transaction t can commit if it gathers the *plurality* of votes. The two conditions stated above enforce mutual exclusion by ensuring that no other conflicting transaction, which may or may not be known to s_i, can gather more votes than t. Note that the latter condition breaks the ties between transactions having the same amount of votes using a simple deterministic comparison between the indices of the servers that created the transactions.

When a candidate transaction t commits at 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.

Abort rule. All active and candidate transactions whose read items are modified are said to become *obsolete* and are aborted. Additionally, the commitment of a transaction causes all votes cast for an obsolete transaction to be discarded.

2.2.4 Synchronization

A pair-wise synchronization session involves the propagation of 1) committed updates, 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 [28]. Each server s_i maintains an n-element vector, vv_i, where n is the number of servers which 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. There are three types of events of interest: transaction commits, transaction promotions, and votes. A commit event is created whenever the local server commits a transaction. A promotion event is created whenever a transaction becomes a candidate on the server where it executed. 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 for all servers. We denote the pth such event (created by s_j) as e_j^p. As information about events is always propagated in local order, 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. We here assume unidirectional pull synchronization, although other modes are possible [17], [25]. When s_i pulls information from s_j, the following actions take place:

1. Server s_i sends vv_i to s_j.
2. Server s_j responds with all events e_k^l s.t. $l > vv_i[k]$ and $l \leq vv_j[k]$, for all $k = 1 \ldots n$.
3. Server s_i incorporates the new events in the same order in which they originally occurred by processing new commitments, candidates, and votes; applying the voting rule, the commit rule, and the abort rule for all relevant transactions; and updating vv_i to the pair-wise maximum of vv_i and vv_j.

Deno's protocol divorces correctness requirements from the communication requirements: Any server can be chosen as a synchronization partner without affecting the correctness of the protocol. It is, however, clear that this choice can have significant performance affects and it is worthwhile to perform more intelligent, *directed* synchronization at the expense of storing and propagating extra information about the currencies held by the servers [9]. The investigation of this issue is beyond the scope of this paper and we assume that synchronization partners are chosen randomly in the rest of the paper.

2.2.5 Consistency and Correctness Issues

We now discuss the consistency level provided by the base voting protocol. See the Appendix for the correctness proofs of the theorems (and relevant lemmas) presented in this section.

We first formally define the consistency level supported by the base Deno algorithm:

Definition 1 (Weak consistency). *A query sees* weak consistency *if it serializes with respect to all update transactions, but possibly not with other queries [5], [6], [19].*

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).

We can now state the theorem that defines the consistency level provided by our protocol:

Theorem 1 (Weak consistency). *The base Deno protocol provides weak consistency (see the Appendix for a proof sketch).*

2.3 Illustration

We illustrate the basic Deno protocol in Fig. 4 with an example scenario. The system has four servers, each holding a currency of 0.25. Server s_a creates a new update, t_1, votes (*yes*) 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 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). It then commits t_1 (since $|votes(t_1)| = 0.75$) and aborts t_4 (since t_4 has become obsolete by the commitment of t_1).

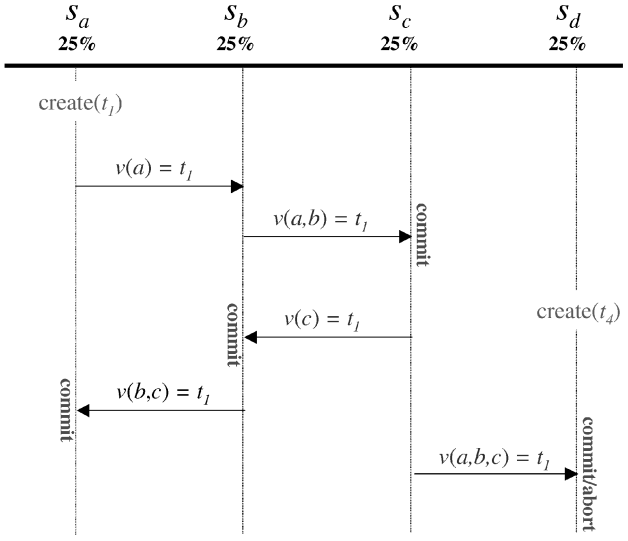

Fig. 4. Protocol illustration.

2.4 Providing Strong Consistency: Extended Protocol

2.4.1 Supporting Strong Consistency

The base protocol ensures that queries always access transactionally consistent data and that update transactions are globally serialized with respect to each other. However, the base protocol does not serialize update transactions with respect to all queries. We now describe an extension of the base protocol that provides strong consistency [5], [6], [19], which we define as follows:

Definition 2 (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 multiquery cycles (i.e., cycles involving multiple queries and one or more update transactions). This form of consistency guarantees globally-serializable executions [5], [6], [19].*

The base protocol fails to provide strong consistency because nonconflicting 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 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 its votes (using timestamps). 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.

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 et al.'s protocol [3]). This latter approach requires contacting all servers in the system, which may be a serious restriction during times of low availability.

2.4.2 Update Commitment

Formally, we refer to a vote $v \in V_i$ as a *top vote* at s_i if $v.tstamp < v'.tstamp,\ \forall v' \in V_i\ , v.voter = v'.voter$. We then refer to a candidate transaction $t \in C_i$ as a *top transaction* at s_i if $\exists v \in V_i,\ v.trans = t$ and v is a top vote at s_i. We then compute the votes of a top transaction t as:

$$|votes(t)| = \sum_{v \in V_i \wedge v.trans = t} v.curr.$$

In this case, there is a single unknown value that is the same for all transactions (as opposed to different individual unknown values as describes in Section 2.2.3):

$$unknown = 1.0 - \sum_{t \in C_i \wedge t \text{ is a top transaction}} votes(t).$$

The updated commit rule can then be stated as follows: A *top* transaction $t \in C_i$ commits when, $\forall t' \in C_i$ such that t' is also a top transaction:

1. $|votes(t)| > |votes(t')| + unknown(t)$ or
2. $|votes(t)| = (|votes(t')| + unknown(t))$ and $t.id < t'.id$,

where *t.id* is the identifier of the server that created *t*.

2.4.3 Correctness

The following lemma establishes the unique global commit order of all updates (refer to the Appendix for correctness proof sketches of the following theorem).

Theorem 2 (Strong consistency). *The extended Deno protocol provides strong consistency and serializability.*

3 Secure Update Commitment

In this section, we discuss how we deal with a specific class of internal threats that result from authenticated but malicious servers (see [11] for a description of how to provide security against external threats). Such malicious insiders misrepresent protocol-specific information and can cause potentially corrupt objects to propagate throughout the system. Under certain circumstances, even a single malicious insider with an arbitrarily small amount of currency can cause different transactions to be committed at different servers. We first discuss the set of malicious actions a server can undertake and then discuss our approach to handling them.

3.1 Malicious Actions

Before we classify the actions a malicious intruder can take, we note that malicious servers can always commit arbitrary transactions to their *local* databases. 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 currency. This attack is handled by the Deno's standard *currency revocation* mechanism [9] that is used to recover from benignly failed servers. This mechanism requires at least a majority of the currency to be held by nonmalicious nodes, thereby providing *liveness* assuming at most $\lfloor n/2 \rfloor - 1$ malicious nodes exist (under uniform currency distribution).

Malicious insiders, therefore, 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 replication framework, a malicious server can incorrectly report currency values or votes, which we discuss below.

Currency misrepresentation. The problem here is of a server misrepresenting the amount of currency it has. Due to the decentralized nature of the system and the fact that Deno allows *peer-to-peer currency exchanges* [9], the currency a server holds cannot be directly verified by other servers. We make this operation secure by requiring each currency exchange to be formalized as an update. A currency transfer from s_i to s_j is only considered complete when the corresponding *exchange update* is committed. Note that such exchange updates are commutative with respect to all other updates.

Vote misrepresentation. There are two types of vote misrepresentation:

- *Misrepresenting nonlocal 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 peer model does not impose any specific connectivity requirements, this behavior can only delay the commitment of transactions, but cannot affect correctness.
- *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 currency more than once for multiple transactions.Consider the example shown in Fig. 5. 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

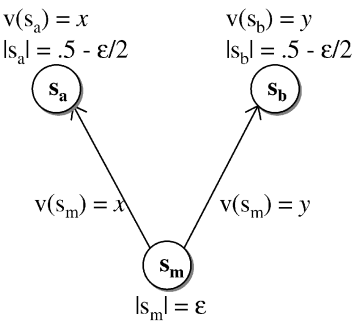

Fig. 5. Vote misrepresentation.

candidates have more than 50 percent 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.

3.2 Vote Validation

We now investigate approaches for detecting malicious servers and develop an algorithm that guarantees correctness at nonmalicious servers. We revise our decentralized update commitment algorithm such that 1) it guarantees correctness *even when there are* (*multiple*) *malicious servers*, 2) it allows progress *even when not all votes have been reported*, and 3) it provides safety at a node *if the node supports a sufficient degree of tolerance* (at the expense of potential performance degradation). The central idea is to make commit decisions based on votes that are guaranteed to belong to *non*malicious servers.

We first distinguish between *validated* and *unvalidated* votes: The former are *known* to be correct (i.e., nonmalicious) and the latter may or may not be correct. Our approach hinges on the observation that "*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 ($\delta = 0 \ldots n-1$).

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 because 1) validated votes cannot be duplicates by definition and 2) of all the unvalidated votes, at worst the largest unvalidated vote may be a duplicate.

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 (nonmalicious) server to be validated votes. We denote the currency of any vote v in $votes(t_i)$ by $|v|$. Similarly, we denote the total currency for a set V of votes by $|V|$. For example, $|votes(t_i)|$ denotes the sum of the currencies of all votes cast for $t_i \in C$, where C is the set of candidate transactions. Finally, let $unvalid(\delta, C)$ be the set of δ elements with the largest currency in $unvalid(C)$. 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 1, where *unknown* is defined as $1 - |votes(C)|$.

In order to provide resilience against malicious servers, the nonsecure commit criterion is modified as in the second row of Table 1. The lefthand side of the inequality provides a *lower bound* on the amount of currency that t_i is guaranteed to have by not using the δ largest unvalidated votes cast for t_i. The righthand side of the inequality provides an upperbound on the amount of currency t_j can possibly get. Thus, the amount of currency required to commit t_i must be larger than the total currency 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, nonsecure 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, 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 affect the liveness properties of the algorithm, but, once again, safety guarantees are intact.

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 currency revocation mechanism [9] to cancel the voting rights of the malicious server. If the server already

TABLE 1
Secure versus Nonsecure Commit Criteria

$\|votes(t_i)\| > \|votes(t_j)\| + unknown, \quad t_i, t_j \in C, i \quad j$	non-secure criterion
$\|votes(t_i)\| - \|unvalid(\delta, t_i)\| > \|votes(t_j)\| + unknown, \quad t_i, t_j \in C, i \quad j$	secure criterion

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. The Appendix contains a correctness proof sketch of the revised commit criterion.

3.3 Illustration

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., $\delta = 1$). 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. In order to keep the exposition clean, we omit the timestamp element when representing a vote (as the value of this element is irrelevant for the purposes of the following examples).

Example 1. Assume five servers, $s_1, s_2, \ldots, s_5$, in the system, each holding equal (i.e., 0.2) currency, and the following votes at s_1:

$$V_1 = \{(s_1, t_1, 0.2), (s_2, t_1, 0.2), (s_3, t_1, 0.2), (s_4, t_1, 0.2), (s_5, t_2, 0.2)\}.$$

In terms of the secure commit criterion: $|votes(t_1)| = 0.8$, $|unvalid(1, t_1)| = 0.2$, $|votes(t_2)| = 0.2$, and $unknown = 0$. In this case, s_1 can commit t_1 without validating a single vote!

The next 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 2. Assume servers $s_1, s_2, \ldots, s_4$ have currencies 0.2, 0.4, 0.2, and 0.5, respectively. Votes at s_1 are:

$$V_1 = \{(s_1, t_1, 0.2), (s_2, t_1, 0.4), (s_3, t_1, 0.2), (s_4, t_2, 0.5)\}.$$

Using the secure commit criterion: $|votes(t_1)| = 0.8$, $|unvalid(1, t_1)| = 0.4$, $|votes(t_2)| = 0.5$, and $unknown = 0$. Server s_1 cannot commit t_1 because

$$|votes(t_1)| - |unvalid(1, t_1) = 0.4,$$

whereas $|votes(t2_)| + unknown$ 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.

4 Other Deno Features

For completeness, we briefly describe other key features of Deno:

- **Exploiting application-specific commutativity information**. Applications running on top of weakly connected environments and systems need be designed to minimize conflicts among updates in order to avoid high abort rates [21]. One approach is to have applications export domain-specific semantic information that can be used to modify the application's consistency requirements [34]. Deno's extended protocol supports *commutativity procedures* to exploit application-specific commutativity information. A commutativity procedure is a simple query over the database specifying an acceptance criterion [21]. If the query is satisfied, the transaction is considered to be valid with respect to the current state of the database. Deno executes a transaction's commutativity procedure (if it exists) if and when the transaction becomes obsolete. If the acceptance criterion is satisfied, the transaction is not aborted. Note that the use of commutativity procedures does not affect the consistency guarantees.
- **Light-weight, dynamic currency management**. In general, the best (target) currency distribution depends on application semantics, expected availability of individual servers, and network topology. Initial currency allocation is nontrivial because no server may have accurate knowledge or estimate about the size of the anticipated set of servers. The system initially gives all currency to the server that created the objects and other servers obtain currency along with their initial copies of the data. Subsequent peer-to-peer currency exchanges allow the system to incrementally converge to any global target distribution [9], exponentially fast and using only local information (i.e., without global synchronization).
- **Currency proxies**. Deno uses a proxy mechanism to transparently handle planned disconnections of mobile servers [9], [25]. The key idea is to allow servers to specify proxies to represent them during planned disconnections (during an airplane trip, for example) by voting in their place. A proxy vote is then indistinguishable to the other servers from the situation where a server votes and then disconnects. The result is that there are no race conditions and the entire proxy engagement is transparent to the rest of the system. The use of proxies in this manner can prevent degradation in overall commit rate when devices have expected disconnections.

5 Performance Evaluation

This section describes the performance of the Deno prototype. Note that the primary advantage gained in combining voting with epidemic information flow is in increased availability. On the other hand, our evaluation focuses on the relative convergence speeds of different protocols as a function of several metrics, including update contention, commutativity ratio, and the degree of security provided by the system. Results relating to the effects of disconnections and intelligent synchronization partner selection algorithms can be found elsewhere.

5.1 Experimental Environment

We performed the experiments on a cluster of 15 Linux machines (each with two 400 MHz Pentium II's, and 256 MBytes of memory), each running a single copy of the Deno server. The machines were connected via a 100Mbps Ethernet network and the servers communicated using UDP/IP packets. We used a small database consisting of 100 data objects of size 20K each. Each server periodically initiated a synchronization session (with a given synchronization

TABLE 2
Primary Experimental Parameters and Settings

Parameter	*Description*	*Setting*
Synch Period (SP)	Mean synchronization period (uniform)	0 – 5 (secs)
Transaction Rate (TR)	Mean transaction generation rate (uniform)	0 – 25 (trans/SP)
Num Servers (N)	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 database state	0 – 1
Degree of tolerance (δ)	Number of malicious servers that can be tolerated	0 – N-1

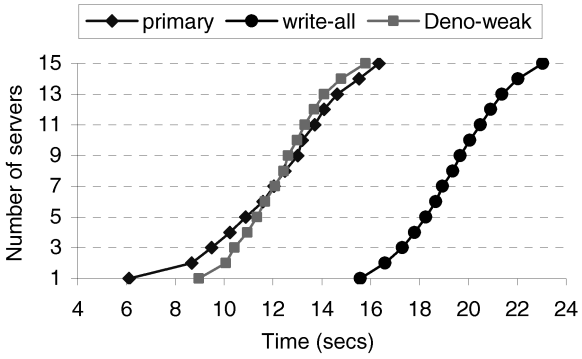

Fig. 6. Number of commitments versus time (N = 15, TR = 0.01, SP = 5).

period) by sending a *pull* request to another randomly selected server. We note that neither the bandwidth nor the CPU is saturated in any of the experiments.

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 update consistency protocols, we did not model any read-only transactions. All objects are replicated at all servers and currency is uniformly distributed across servers in all the experiments. The results presented in the following graphs are the average of five independent runs of executing 1,000 transactions in the system. The main parameters and settings used in the experiments are summarized in Table 2. Our performance evaluation concentrates on relative performance by comparing representative protocols.

We evaluate two versions of Deno's protocol, `Deno-weak` (Section 2.2) and `Deno-strong` (Section 2.4). Additionally, we investigate two representative epidemic replication schemes. The first scheme, `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 [34]. Note that primary-copy protocols trade availability for a presumed advantage in performance. The second scheme, `write-all`, is an epidemic "Read-One, Write-All" (ROWA) [5] 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 observes conflicting transactions, it has to abort all of those transactions to ensure global consistency. This protocol is similar to that proposed by Agrawal et al. [3]. Note that `primary` and `write-all` are the only other pessimistic replication protocols that appeared in the literature. We have also not included any optimistic protocols (see Section 6) in our evaluation as they do not provide formal correctness guarantees.

5.2 Commit Delays

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

Fig. 6 presents commit delays by plotting the number of servers that committed the transaction as time progresses for `primary`, `write-all`, and `Deno-weak`, when there is no update contention. 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 average) committed the transaction independently. The delay between the first and subsequent commits is thus quite small.

One important implication of this result is that the performance penalty of using voting rather than a primary-copy approach is not as large as commonly assumed. The results for `Deno-strong` (not shown) are virtually similar to those for `Deno-weak` because there is no contention and thus no conflicts.

5.3 Contention Effects

The previous subsection focused on the speed of transaction commits when there is no update contention. Fig. 7 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 (TR in [0, 1]), all protocols perform fairly well, achieving commit percentages of around 100 percent. With increasing transaction rates, however, commit percentages drop for all protocols significantly. 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

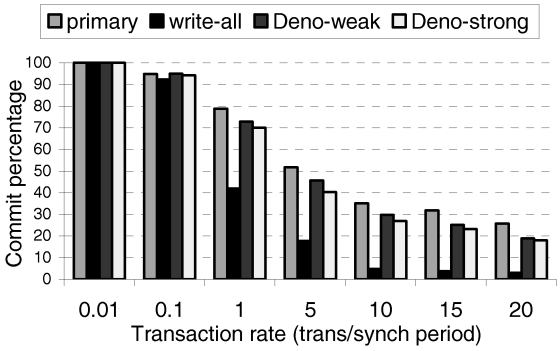

Fig. 7. Commit percentage versus update contention (N = 15, SP = 5).

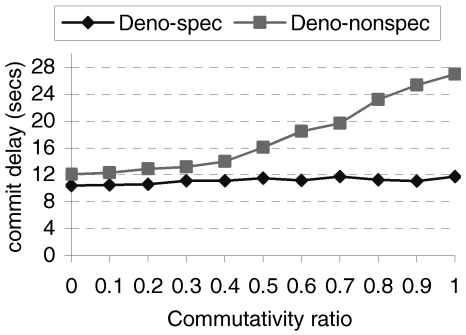

Fig. 8. Speculation effects on commit delay (N = 15, TR = 0.01, SP = 5).

whole range shown is small (within absolute 5 percent). The performance of `write-all` is significantly lower than the rest of the protocols. In fact, at (and beyond) a transaction rate of 25 (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 mechanism 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).

The most interesting result from this series of experiments is that the base Deno protocol did not appear to have any significant performance advantage of the extended version. The difference between the commit delays of the two with little contention appears is up to an average of 10 percent with reasonable 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 currency to be inspected before a winner of a given *election* can be determined. For example, we only need more than 50 percent of the currency in order to determine the winner of an election if there are no conflicting transactions, but we may need all of the currency in order to decide between two or more. However, the increase in required *currency* is offset by an increase in *concurrency*. Therefore, update contention does not necessarily increase commit delays.

5.4 Speculative Voting and Update Propagation

Recall from Section 2 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 proceed and participate in the global voting protocol only after the conflicting transactions are terminated.

We now propose an optimistic alternative that skips the blocking phase by having the servers immediately vote for all transactions as soon as they finish their local execution. These transactions immediately become candidates to be added to subsequent synchronization sessions. 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 most useful when previous conflicting transactions are aborted. As shown below, the advantage conferred by this technique is larger when there are commuting updates in the system. The cost of speculation is that some transactions that will eventually get aborted are propagated through the system unnecessarily, resulting in a waste of communication bandwidth.

Fig. 8 examines the benefits of speculative update propagation and voting for varying degrees of commutativity by showing the performance of speculative (`Deno-spec`) and nonspeculative (`Deno-nonspec`) versions of `Deno-strong` (a description of the modifications required to support speculation can be found in [12]). Somewhat nonintuitively, larger commutativity ratios result in larger commit delays for the nonspeculative Deno. The reason is that increasing commutativity results in fewer aborted transactions, which in turn increases contention for those transactions that are yet to be terminated. By contrast, `Deno-spec`'s commit delay is largely constant across all commutativity ratios. Speculative voting confers a performance advantage of about 15 percent even with a commutativity ratio of 0, the default case where no transactions commute. The gap increases with commutativity ratio until `Deno-nonspec`'s commit delay is more than twice `Deno-spec`'s at a ratio of 1.0.

The benefits of speculation come at the expense of propagating more transactions and votes. To this end, we investigate the relative bandwidth utilizations of the protocols in Fig. 9, which shows the amount of information sent across all servers (in KBytes) per committed transaction for `Deno-spec` and `Deno-nonspec`. For low commutativity ratios (i.e., up to 0.1), `Deno-spec` propagates about 4-6 percent more information per committed transaction. Beyond a commutativity ratio of 0.2, however, the speculative

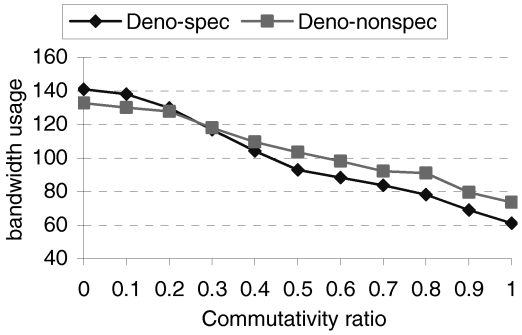

Fig. 9. Speculation effects on bandwidth (N = 15, TR = 0.01, SP = 5).

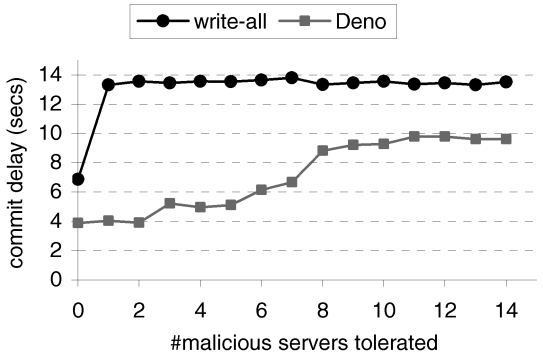

Fig. 10. Commit delay versus degree of tolerance (N = 15, TR = 0.01, SP = 2).

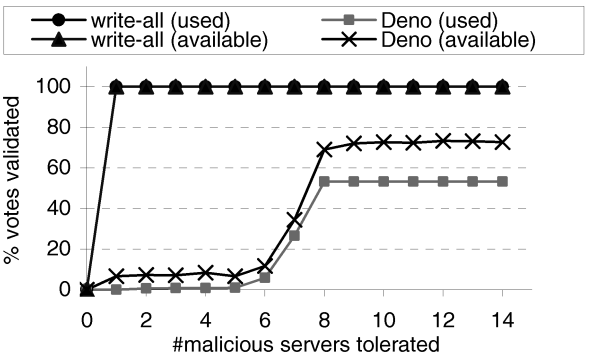

Fig. 11. Percentage of validation versus degree of tolerance (N = 15, TR = 0.01, SP = 2).

protocol sends less information than the nonspeculative version, with the difference increasing as the commutativity increases. At a commutativity ratio of 1, `Deno-spec` propagates about 16 percent less information per committed transaction. To summarize, the speculative version not only decreases average commit delays, but it also decreases bandwidth requirements per committed transaction.

5.5 Commit Delays versus Degree of Tolerance

Fig. 10 shows the average commit delays for small transaction generation rates (i.e., no update contention), for Deno and `write-all`, for varying degrees of tolerance to malicious servers. On the x-axis, we vary δ from 0 (nonsecure system) to $n-1$ (max-security system). The Deno curve follows an *S*-shape: It initially increases gradually with increasing δ, makes a significant jump in the vicinity of $\delta = n/2$, and then essentially stays flat afterward. 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. However, when δ is more than half the number of 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 which 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.

The figure also depicts commit delays for nonsecure `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. For all δ, Deno commits updates significantly faster than `write-all`, reducing the commit delays of `write-all` by 40 percent and 30 percent for nonsecure ($\delta = 0$) and maximum security cases ($\delta = n-1$), respectively. The most dramatic improvement, 60 percent, 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.

5.6 Supporting Nonuniform 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 will be *independent* of the degrees of tolerance supported by others since each Deno server makes 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, $\delta(rest)$.

Fig. 12 presents commit delay results for s and the rest of servers (averaged) for the cases where $\delta(s) = 0$ and $\delta(s) = N-1$, as we vary $\delta(rest)$—note that d refers to δ in the figures. When we consider the commit delay for s when $\delta(s) = 0$, we observe that it remains essentially flat regardless of $\delta(rest)$. The same observation holds for the case where $\delta(s) = N-1$. It is clear 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 $\delta(s) = N-1$ illustrate the complementary case. We observe that these two curves are essentially identical, revealing that system performance as a whole is not affected by $\delta(s)$. It is therefore clear that the degree of tolerance adopted by a server does not adversely affect the performance of other servers and vice versa.

Fig. 11 provides further insight by plotting the percentage of validated votes that are used and available at

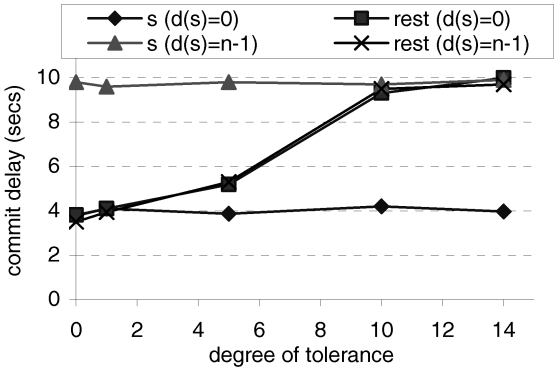

Fig. 12. Commit delays versus degree of tolerance (N = 15, TR = 0.01, SP = 2).

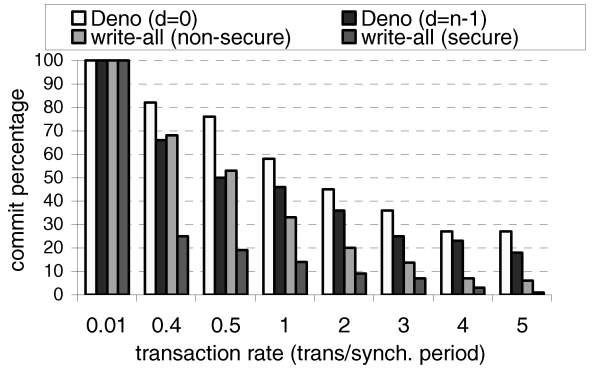

Fig. 13. Commit percentage versus contention (N = 15, SP = 2.0).

commit time at each server, averaged over all commits across all servers. No validated votes are used at commit by Deno when $\delta < N/2$. Validated votes available at commit time at each server are nonzero because each server considers its own vote as validated by default. Notice that Deno requires at most 50 percent of the votes to be validated for supporting any degree of tolerance. On the other hand, `write-all` requires 100 percent of the votes to be validated to tolerate any number of malicious servers, thereby incurring relatively large commit delays.

5.7 Update Contention Effects and Scalability

Fig. 13 plots the commit percentage results for varying transaction generation rates for Deno and `write-all`. The figure shows that the 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. Under very small transaction rates (*TR* in [0.0, 0.01]), all protocols perform fairly well, committing all updates. With increasing transaction rates, however, commit percentages begin to drop 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 percent, whereas the commit percentages of the other protocols are all above 65 percent. Notice that, beyond a transaction rate of 0.5, *max-security* Deno has a higher commit percentage than even the *nonsecure* `write-all`. The `write-all` protocol lacks any mechanism that can pick one out of multiple conflicting updates and thus has to abort all conflicting updates. Due to this behavior (beyond six and 10 transactions/synch period, respectively), the `write-all` approaches, both secure and nonsecure, cannot commit any updates. On the other hand, the Deno approaches continue to make progress and commit updates regardless of the update generation rate (not shown). Scalability results (see [11]) show that the performance difference between Deno and `write-all` increases with increasing system size.

6 Related Work

The problem of consistent access to replicated data has long been studied in many contexts and a wide variety of solutions have been proposed (e.g., [2], [5], [16], [19], [33], [35]). Because of the intrinsic shortcomings of traditional synchronous replication solutions [20], [33], [35] in mobile and weakly connected environments [21], asynchronous replication protocols have recently gained a lot of attention (e.g., [7], [24], [26]). Asynchronous approaches commonly allow servers to execute updates locally without any synchronization with other servers and propagating updates afterward as separate activities. Due to space limitations, we restrict our attention to asynchronous update-anywhere approaches that utilize the epidemic model [3], [17], [24], [29], [34].

Many epidemic systems take an optimistic approach and use reconciliation-based protocols (e.g., Ficus [29], Lotus Notes [24]) that are only viable in nontransactional single-item domains such as file systems. These approaches only ensure that all copies of a single item eventually converge to the same value and therefore are not safe for environments requiring transactional semantics.

Bayou [34] takes a more pessimistic approach and ensures that all committed updates are serialized in the same order at all servers using a *primary-copy* scheme. More recently, Agrawal et al. [3] described a pessimistic ROWA [5] approach that ensures strong consistency and serializability. Our protocols differ from these protocols primarily in using a novel combination of weighted-voting and epidemic information flow to improve availability and performance.

Holliday et al. [22] also proposed an epidemic quorum-based approach that provides serializability as our extended protocol. Holliday's work assumes a more traditional replicated database environment and static currencies, whereas our emphasis is on making progress under incomplete system information in dynamic environments.

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. Security research in synchronous centralized systems mostly targeted group communication protocols. Ensemble [32] addresses only external security threats providing secure authentication, integrity, and privacy, whereas Rampant [31] is designed to handle Byzantine attacks. These systems 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., [15], [18]) provide voter privacy and rely on a small number of central facilities for counting votes, but are impractical under weak-connectivity as restrictions in connectivity and disconnections make reliance on central authorities untenable.

Castro and Liskov [8] described a practical replication algorithm for tolerating Byzantine attacks in asynchronous environments where update commitment is centralized and coordinated via a primary-copy server. To the best of our knowledge, the only proposal that addresses system security using asynchronous epidemic information flow is that of Malkhi et al. [27]. Malkhi et al. provided an analytical treatment of epidemic-style update diffusion algorithms that are tolerant of Byzantine faults. This work makes strong assumptions about the number of malicious servers and where and how updates are created and initially received. No previous work, including Malkhi et al.'s [27], has addressed update commitment in decentralized, epidemic systems and databases.

7 Conclusions

We presented the design, implementation, and evaluation of Deno, a highly available object-replication system designed for 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 [34], and higher availability and performance than ROWA approaches [3].

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 classified and addressed a specific class of internal security attacks that involve vote misrepresentation. We proposed a secure version of our protocols that handles such malicious actions using a combination of cryptographic primitives, modifications to the update commit criteria, and explicit validation of votes. The proposed protocol is parameterized and can be tuned to trade off degrees of tolerance to malicious insiders and commit performance, thereby allowing individual servers to set arbitrary degrees of tolerance based on their individual requirements and resources.

Our performance study based on the Deno prototype 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. Third, protecting against internal threats comes at a cost, but the marginal cost for protecting against larger cliques of malicious insiders is generally low. 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.

Appendix A

Correctness of the Base Deno Protocol

We now provide proof sketches for the theorems presented in the paper. We first restate the notion of a serialization graph [5]. A serialization graph consists of vertices and edges, where vertices represent transactions and edges represent constraints on equivalent serial orderings. An edge $t_i \rightarrow t_j$ exists in the graph if t_j reads or writes a data item written by t_i or writes a data item read by t_i.

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

Proof (sketch). 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 serializability of updates.*

Proof (sketch). Assume that the protocol generates a nonserializable global schedule involving update transactions. Then, by Lemma 1, there exists a cycle in the global serialization graph [5] 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. *Read-Write* (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. *Write-Read* (t_2 *reads an item written by* t_1): This case is the opposite 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. *Write-Write* (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. □

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 (sketch). 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

```
Record Item  =
  id;       // item id
  version;  // version number
  data;     // data image

Record Transaction  =
  server;   // id of the server that created the transaction
  readSet;  // set of read data items
  writeSet; // set of write data items
  status;   // can be {"active", "pre_committed", "blocked", "candidate",
            //         "committed", "aborted"}

Record Vote  =
  voter;    // server that created the vote
  trans;    // transaction for which the vote is cast
  curr;     // currency the voter voted for
  tstamp;   // voter's local timestamp

Record Event  =
  type;     // can be {"commit", "promotion", "vote"}
  event;    // can be a transaction, vote, or candidate creation event

Database       : database of data items
EC_i[1..n]     : event count vector
VV_i[1..n, 1..m]: event vector
V_i            : vote set
C_i            : candidate transaction set
B_i            : blocked transaction set
CL_i           : commit log
Curr_i         : currency held by the server
```

Fig. 14. Basic record types and persistent data structures used by server i.

$t \rightarrow q$; 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 Deno protocol described in Section 2.1 provides weak consistency.*

Proof (sketch). 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 cannot be any update transaction cycles. Therefore, we conclude that the protocol provides weak consistency. □

Lemma 4 (Eventual termination). *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

```
MakeCandidate(Transaction t)
  if (t.server == i)
    B_i = B_i - {t};
    t.status = "candidate";
    C_i = C_i ∪ {t};
    e = ("promotion", t);
    VV_i[EC_i[i,i]++] = e;
    v = (i, t, Curr_i, VV_i[EC_i[i,i]]);
  else
    if ({∃t' ∈ C_i, ∃v ∈ V_i | Conflicts(t',t) ∧ v.trans == t ∧ v.voter == i})
      v = (i, t, 0, VV_i[EC_i[i,i]]);
    else
      v = (i, t, Curr_i, VV_i[EC_i[i,i]]);
  V_i = V_i ∪ {v};
  e = ("vote", v);
  VV_i[EC_i[i,i]++] = e;
```

Fig. 15. Procedure for making transaction t a candidate transaction at server i.

```
Commit(Transaction t)
  foreach {d ∈ t.writeSet}
      Database.(d.id) = d;
      Database.(d.id).version++;
  Append(CL_i, t);
  e = ("commit", t);
  VV_i[EC_i[i,i]++] = e;
  C_i = C_i - {t};
  V_i = V_i - {v ∈ V_i | v.trans == t}
  t.status = "committed";
```

Fig. 16. Procedure for committing transaction t at server i.

```
Abort(Transaction t)
  C_i = C_i ∪ {t};
  V_i = V_i - {v ∈ V_i | v.trans == t};
  t.status = "aborted";
```

Fig. 17. Procedure for aborting transaction t at server i.

t. When $unknown\_servers(t)$ becomes empty, $CS(t)$ cannot grow further due to the voting rule (see Section 2.2.2) since all servers voted for either t or another transaction that conflicts with t. Now, consider the case where all candidate transactions $t' \in CS(t)$ 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 zero. 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 cannot exist. Such a deadlock has to involve a cycle of the form $votes(t_1) > votes(t_2) > \ldots votes(t_n) > votes(t_1)$, where $t_1, t_2, \ldots, t_n$ are candidate transactions. Since both $votes(t_1) < votes(t_n)$ and $votes(t_n) < votes(t_1)$ cannot 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. □

```
ExecuteLocalTransaction(Transaction t)
  t.status = "active";
  read(t.readSet);
  write(t.writeSet);
  foreach {d ∈ t.readSet}
    if (d.version < Database(d.id).version)
       t.status = "aborted";
  if (t.status == "active")
    if (t.writeSet == ∅)
      t.status = "committed";
    else
      if ({∃t' ∈ C_i, ∃v ∈ V_i | Conflicts(t, t') ∧ v.trans == t' ∧ v.server == i})
        t.status = "blocked";
        B_i = B_i ∪ {t};
      else
        MakeCandidate(t);
    ApplyCommitRule();
```

Fig. 18. Procedure for executing a transaction t at server i.

```
ApplyCommitRule()
  foreach {t ∈ C_i}
    cancommit = true;
    foreach {t' ∈ C_i | Conflicts(t', t)}
      if ¬((votes(t) > votes(t') + unknown(t)) ∨
           ((votes(t) == votes(t') + unknown(t)) ∧ t.server < t'.server))
        cancommit = false;
    if (cancommit)
      Commit(t);
      foreach {t' ∈ C_i | Conflicts(t', t)}
        Abort(t);
      foreach {bt ∈ B_i}
        if ({t'' ∈ C_i, v ∈ V_i | Conflicts(t'', bt) ∧ v.trans == t'' ∧ v.voter == i} == ∅)
          MakeCandidate(t);
     ApplyCommitRule();
```

Fig. 19. Procedure for terminating a transaction t at server i.

```
HandlePullResponse(j, EC_j, VV_j)
  foreach {s ∈ [1..n]}
    foreach {e ∈ I | EC_i[s] < e ≤ EC_j[s]}
      VV_i[s,e] = VV_j[s,e];
      switch (VV_j[s,e].type)
        case "commit":
          Commit(VV_j[s,e].event.trans);
        case "promotion";
          MakeCandidate(VV_j[s,e].event.trans);
        case "vote":
          V_i = V_i ∪ {VV_j[s,e].event.vote};
    EC_i[s] = max(EC_i[s],EC_j[s]);
  ApplyCommitRule();
```

Fig. 20. Procedure for synchronizing with server j at server i.

APPENDIX B

CORRECTNESS OF THE EXTENDED DENO PROTOCOL

Lemma 5 (Global update consistency). *The extended Deno protocol ensures a unique global commit order on the set of update transactions.*

Proof (sketch). 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 extended Deno protocol provides strong consistency (as defined in Definition 2) and serializability.*

Proof (sketch). Lemma 5 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 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. □

APPENDIX C

CORRECTNESS OF THE EXTENDED SECURE COMMIT CRITERION

Theorem 3 (Correctness of the secure commit criterion). *The secure commit criterion presented in Section 3.2 provides safety at a node if the node supports a sufficient degree of tolerance.*

Proof (sketch). Consider n servers $s_1, s_2, \ldots, s_n$, with currencies $c_1, c_2, \ldots, c_n$. Consider a single nonmalicious 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 1. There are two cases: 1) s_i does not use c_m toward the votes cast for t_i and 2) s_i uses c_m toward the votes cast for t_i.

In the former case, t_i gathered the plurality of votes by using only the nonmalicious 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 currencies that are cast by nonmalicious servers toward committing t_i. A straightforward induction on the number of malicious servers concludes the proof. □

APPENDIX D

PSEUDOCODE FOR THE BASE DENO PROTOCOL

Pseudocode for the base Deno protocol is shown in Figs. 14, 15, 16, 17, 18, 19, and 20.

REFERENCES

[1] A.E. Abbadi and S. Toueg, "Availability in Partitioned Replicated Databases," *Proc. Fifth ACM SIGACT-SIGMOD Symp. Principles of Database Systems,* 1986.

[2] D. Agrawal and A.E. Abbadi, "An Efficient and Fault-Tolerant Solution for Distributed Mutual Exclusion," *ACM Trans. Computing Systems,* vol. 9, no. 1, pp. 1-20, 1991.

[3] D. Agrawal, A.E. Abbadi, and R. Steinke, "Epidemic Algorithms in Replicated Databases," *Proc. 16th ACM Symp. Principles of Database Systems,* May 1997.

[4] Y. Amir and A. Wool, "Optimal Availability Quorum Systems: Theory and Practice," *Information Processing Letters,* vol. 65, pp. 223-228, Apr. 1998.

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

[6] P. Bober and M. Carey, "Multiversion Query Locking," *Proc. 18th Conference on Very Large Databases,* 1992.

[7] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silbershatz, "Update Propagation Protocols for Replicated Databases," *Proc. ACM Int'l Conf. Management of Data,* 1999.

[8] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Third Symp. Operating Systems Design and Implementation,* 1999.

[9] U. Cetintemel and P.J. Keleher, "Light-Weight Currency Management Mechanisms in Deno," *Proc. 10th IEEE Workshop Research Issues in Data Eng.,* Feb. 2000.

[10] U. Cetintemel and P.J. Keleher, "Performance of Mobile, Single-Object Replication Protocols," *Proc. 19th IEEE Symp. Reliable Distributed Systems,* 2000.

[11] U. Cetintemel, P.J. Keleher, and B. Bhattacharjee, "A Security Infrastructure for Mobile Transactional Systems," Univ. of Maryland, UMIACS-TR-2000-59, 2000.

[12] U. Cetintemel, P.J. Keleher, and M.J. Franklin, "Support for Speculative Update Propagation and Mobility in Deno," Univ. of Maryland, UMIACS-TR-99-70, Oct. 1999.

[13] U. Cetintemel, P.J. Keleher, and M.J. Franklin, "Support for Speculative Update Propagation and Mobility in Deno," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* 2001.

[14] U. Cetintemel, B. Ozden, M.J. Franklin, and A. Silberschatz, "Design and Evaluation of Token Redistribution Strategies for Wide-Area Commodity Distribution," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* 2001.
[15] L. Cranor and R. Cryton, "Sensus: A Security-Conscious Electronic Polling Scheme for the Internet," *Proc. Hawai Int'l Conf. System Sciences,* 1997.
[16] S. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a Partitioned Network: A Survey," *ACM Computing Surveys,* vol. 17, no. 3, pp. 341-370, 1985.
[17] 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," *Proc. Sixth ACM Symp. Principles of Distributed Computing,* 1987.
[18] A. Fujioka, T. Okamoto, and K. Ohta, "A Practical Secret Voting Scheme for Large-Scale Elections," *Advances in Cryptology,* 1992.
[19] H. Garcia-Molina and G. Wiederhold, "Read-Only Transactions in a Distributed Database System," *ACM Trans. Database Systems,* vol. 7, no. 2, pp. 209-234, June 1982.
[20] D.K. Gifford, "Weighted Voting for Replicated Data," *Proc. Seventh ACM Symp. Operating Systems Principles,* 1979.
[21] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The Dangers of Replication and a Solution," *Proc. 1996 ACM Int'l Conf. Management of Data,* June 1996.
[22] J. Holliday, R. Steinke, D. Agrawal, and A.E. Abbadi, "Epidemic Quorums for Managing Replicated Data," *Proc. 19th IEEE Int'l Performance, Computing, and Comm. Conf.,* 2000.
[23] S. Jajodia and D. Mutchler, "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database," *ACM Trans. Database Systems,* vol. 15, no. 2, pp. 230-280, 1990.
[24] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and L. Greif, "Replicated Document Management in a Group Communication System," *Proc. Conf. Computer Supported Cooperative Work,* 1988.
[25] P.J. Keleher, "Decentralized Replicated-Object Protocols," *Proc. 18th ACM Symp. Principles of Distributed Computing,* May 1999.
[26] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *ACM Trans. Computing Systems,* vol. 10, no. 4, pp. 360-391, Nov. 1992.
[27] D. Malkhi, Y. Mansour, and M. Reiter, "On Diffusing Updates in a Byzantine Environment," *Proc. 18th IEEE Symp. Reliable Distributed Systems,* 1999.
[28] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms,* 1989.
[29] 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,* vol. 28, no. 2, pp. 155-180, Feb. 1998.
[30] D. Peleg and A. Wool, "The Availability of Quorum Systems," *Information and Computation,* vol. 123, no. 2, pp. 210-223, 1995.
[31] M.K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," *Proc. Second ACM Conf. Computer and Comm. Security,* 1994.
[32] O. Rodeh, K.P. Berman, M. Hayden, Z. Xiao, and D. Dolev, "Ensemble Security," Cornell Univ. TR-98-1703, 1998.
[33] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. Software Eng.,* vol. 5, no. 3, pp. 188-194, May 1979.
[34] 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," *Proc. ACM Symp. Operating Systems Principles,* 1995.
[35] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. Database Systems,* vol. 4, no. 2, pp. 180-209, 1979.

Ugur Çetintemel received the PhD degree in computer science from the University of Maryland, College Park, in 2001. He is currently an assistant professor in the Department of Computer Science at Brown University. His research focuses on resource and data management issues in networked information systems. He is a member of the IEEE.

Peter J. Keleher received the PhD degree in computer science from Rice University in 1995. He is currently an assistant professor in the Computer Science Department at the University of Maryland, College Park. His primary interests are in the design and analysis of distributed computing infrastructure, including global resource management, distributed shared memory, and communication performance. He is currently leading the design of Active Harmony, a new framework for global resource management in dynamic, heterogeneous environments. He received the US National Science Foundation CAREER award in 1996.

Bobby Bhattacharjee is an assistant professor in the Computer Science Department at the University of Maryland, College Park. His research interests are in the design and implementation of scalable systems, protocol security, and peer-to-peer systems. He is a member of the ACM.

Michael J. Franklin received the PhD degree from the University of Wisconsin, Madison in 1993. He is an associate professor of computer science at the University of California, Berkeley. His research focuses on the architecture and performance of distributed databases and information systems. Previously, he was on the faculty at the University of Maryland, College Park, where he helped develop the DIMSUM flexible query processing architecture and the Broadcast Disks data dissemination paradigm. He was program chair for the 2002 ACM SIGMOD Conference, and currently serves as an editor of the *ACM Transactions on Database Systems*, vice chair of the ACM SIGMOD Advisory Board, a member of the Board of Trustees of the VLDB Endowment, and on several corporate technical advisory boards.