# Support for Speculative Update Propagation and Mobility in Deno

Uğur Çetintemel
*Dept. of Computer Science*
*University of Maryland*
*ugur@cs.umd.edu*

Peter J. Keleher
*Dept. of Computer Science*
*University of Maryland*
*keleher@cs.umd.edu*

Michael J. Franklin
*Computer Science Divison, EECS*
*University of California, Berkeley*
*franklin@cs.berkeley.edu*

## Abstract

*This paper presents the replication framework of Deno, an object replication system specifically designed for mobile and 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. Deno has been implemented and runs on top of Linux and Win32 platforms. We use the Deno prototype to characterize the performance of two versions of Deno's protocol. The first version enables globally serializable execution of update transactions. The second supports a weaker consistency level that still guarantees transactionally-consistent access to replicated data. We demonstrate that the incremental cost of providing global serializability is low, and that* speculative *dissemination of updates can significantly improve commit performance.*

## 1 Introduction

This paper describes the design, implementation, and performance of Deno, a system that supports object replication in a transactional framework for mobile and weakly-connected environments. Deno's system model is illustrated in Figure 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 underlying protocols are based on an asynchronous protocol called *bounded weighted voting* [16]. Asynchronous solutions for managing replicated data [5, 12, 15, 17] 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 slow or require reconciliation, or have low availability because they rely on primary-copy schemes [20].

The focus of this paper is a new decentralized, asynchronous replica management protocol that addresses these concerns. The protocol retains the advantages of current asynchronous protocols, but generally performs better, has fewer connectivity requirements, and 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 [9], a process where information flows pair-wise through the system 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 [21]). 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., [11, 14, 22]), 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 currency distributions to site availabilities, update activity, or other relevant characteristics [6]. 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 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

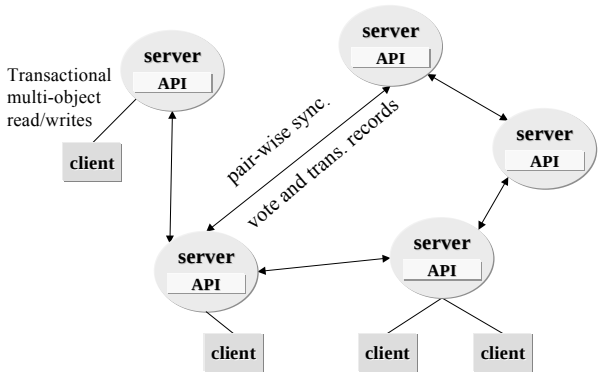


**Figure 1: Basic Deno system model**

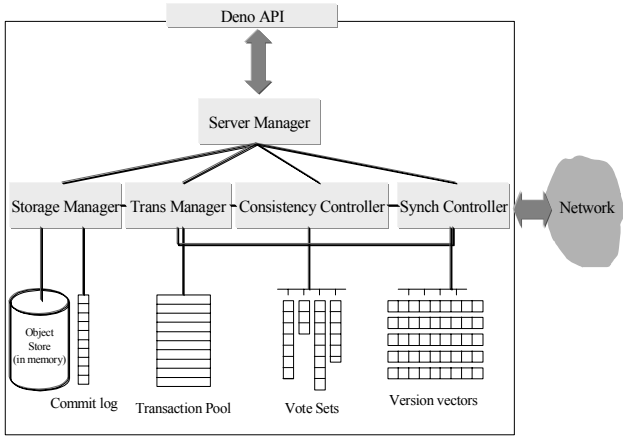


**Figure 2: Basic Deno architecture**

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 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.

Our performance study is based on the Deno prototype. The basic Deno architecture has been implemented and runs on top of Linux and Win32 platforms. 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 just as fast as a much less available/reliable primary-copy protocol.

Our second finding was that support for global serializability is relatively inexpensive in this environment. One of our protocols implements a form of weak consistency [4, 10], where update transactions are serializable and queries always access transactionally-consistent database state. While this is sufficient for many applications, we also have a second variant that supports globally serializable executions. Under both protocols, read-only transactions execute entirely at the local server, and do not require network communication.

Finally, we show that disseminating updates and protocol-specific information *speculatively* can significantly improve the performance of protocols based on epidemic or similar communication mechanisms.

This paper extends our prior work [6, 16], which defined consistency for only single replicated objects, with support for multi-item transactions, serializability, and speculative information propagation, and with experimental evaluation on a prototype system.

The rest of this paper is structured as follows. Section 2 describes the Deno architecture and Section 3 describes Deno's decentralized replication protocols. Section 4 describes Deno's support for mobility, and Section 5 presents the results of our performance study. Finally, Section 6 briefly describes related work, and Section 7 concludes.

## 2 Deno 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.

Figure 2 illustrates the basic 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. All communication is made on top of UDP/IP. Deno consists of ~15,000 lines of multi-threaded C++ code, and has a footprint of ~200KB.

## 3 Decentralized replication protocols

Before delving into the fine detail, we give a quick overview of the *life* of a Deno transaction (Figure 3). A

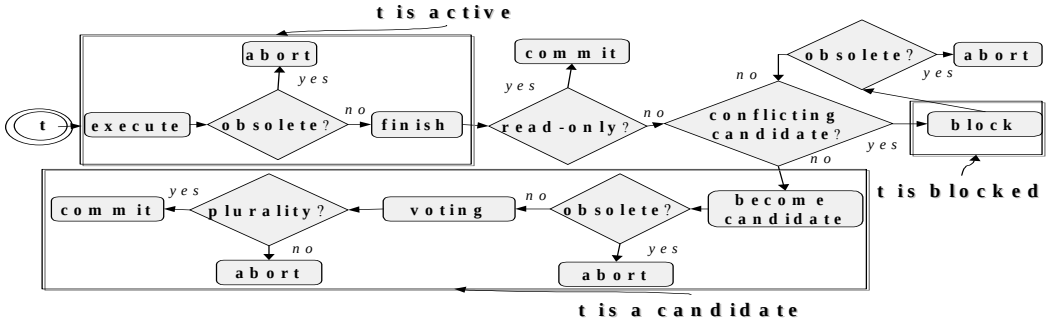


**Figure 3: A transaction's life**

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* — meaning that the update can become visible to other servers. Candidates are voted on, and are eventually either committed (if they corner a plurality of the total system currency), or aborted.

### 3.1 Providing weak consistency: base protocol

**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*. Current values 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.

**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*,
- *v.tstamp* is the value of *v.voter*'s local timestamp, which 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 with its full 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 0.0 currency, in which case the vote is called a *no vote*.

We now describe the voting process from the perspective of a single server. Each server $s_i$ maintains the following major data structures: (1) a set of votes, $V_i$; (2) 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 $s_i$; (3) a list of *blocked* transactions, $B_i$, consisting of locally completed transactions waiting to become candidates; and (4) 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* 1.0. 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*; $s_i$ 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 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 *conflicting* candidate transactions.

*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

$$votes(t) = \sum v.curr ,$$

where $v \in V_i$, and $v.trans=t$, and the unknown votes of a transaction *t* as

$$unknown(t) = 1.0 - \sum s.curr ,$$

where *s* is a server that already voted yes or no for *t*, and *s.curr* is the currency held by *s*.

In other words, *unknown*(*t*) is essentially 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:

$$votes(t) > votes(t') + unknown(t)$$

The commit rule states that candidate transaction *t* can commit if it gathers the *plurality* of votes. The rule enforces mutual exclusion by ensuring that no other conflicting transaction, which may or may not be known to server $s_i$, can gather more votes. Note that ties between transactions having the same amount of votes can be 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.

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

**Synchronization.** A pair-wise synchronization session essentially 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 [18]. 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. We denote the $j^{th}$ such event as $e_i^j$. 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 a unidirectional pull synchronization, although other modes are possible [9, 16]. 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 $l \le vv_j[k]$, for all $k = 1 \ldots n$.
3. Server $s_i$ incorporates the new events in the same order that 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 $vv_j$.

**Consistency issues.** The base Deno protocol described above supports a form of *weak consistency* [3, 4, 10] where each query serializes with respect to all update transactions, but possibly not with other queries. More specifically, the protocol ensures globally serializable execution of update transactions alone, i.e., no *update transaction cycles* in the serialization graph. However, the protocol allows *multiple-query cycles*, i.e., cycles involving multiple queries and multiple update transactions. In other words, each query observes a serial order of update transactions, which is not necessarily the same order observed by other queries. This form of weak consistency does ensure that queries always observe transactionally-consistent database states. Furthermore, as proved in [7], no local or global deadlocks are

| 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 | Number of Deno servers | 3 – 15 |
| Trans Size | Number of items updated by a trans. (uniform) | 0 – 5 |
| Commutativity Ratio | The probability that a trans. is acceptable on a given db state | 0 – 1 |

**Table 1: Primary experimental parameters and settings**

possible. A more detailed discussion including correctness proofs, and illustrative examples can be found in [7].

### 3.2 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, 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 [3, 4, 10], where each query is serialized with respect to both other queries and update transactions, thereby guaranteeing globally-serializable executions. This form of consistency is characterized by an *acyclic* serialization graph [3], prohibiting both update transaction cycles and multi-query cycles.

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 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. The details of the strong-consistency protocol and the corresponding correctness proofs can be found in [7].

## 4 Support for mobility

For completeness, we briefly discuss some of Deno's mobility-related features:

**Proxies**. Deno allows servers to specify proxies to represent them during planned disconnections (during an airplane trip, for example) by voting in their place [6, 16].

**Application-specific commutativity information**. Applications running on top of dis- and weakly-connected environments and systems need be designed to minimize conflicts among updates in order to avoid high abort rates [12]. One approach is to have applications export domain-specific semantic information that can be used to modify the application's consistency requirements [21]. 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 [12]. 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**. The system initially gives all currency to the server that created the objects. Other servers obtain currency along with their initial copies of the data. Subsequent peer-to-peer currency exchanges allow the system to approach to any global target distribution *exponentially fast* [6].

## 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, which we do not discuss in this paper.

### 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 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 period) by sending a *pull* request to another randomly selected server.

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 1000 transactions in the system. The main parameters and settings used in the experiments are summarized in Table 1. Our performance evaluation concentrates on relative performance by comparing representative protocols.

We evaluate two versions of Deno's protocol, `Deno-weak` (Section 3.1), and `Deno-strong` (Section 3.2). Additionally, we investigate two representative epidemic replication schemes from the literature. The first scheme, `primary`, is an epidemic primary-copy scheme that uses a specialized primary server to serialize the updates,

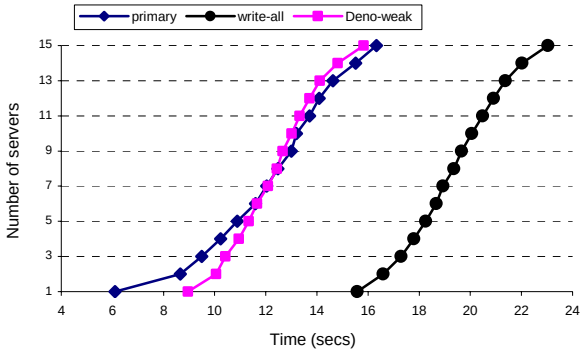


**Figure 4: Number of servers that committed the transaction as time progresses (15 servers, *TR*=0.01, *SP*=5.0)**

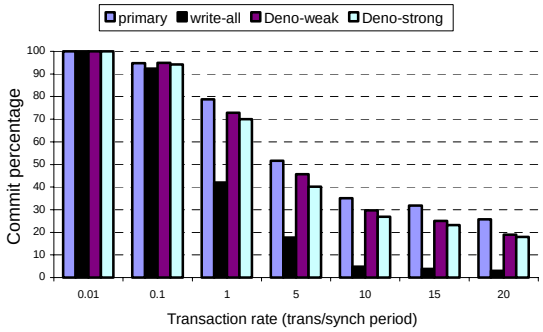


**Figure 5: Commit percentages (15 servers, *SP*=5.0)**

while propagating the updates using epidemic flow. This protocol is similar to that used in Bayou [21]. 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) [3] 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.* [2].

## 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.

Figure 4 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 (for 15 servers). 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 subsequent commits is thus quite small, as revealed by the high slope of the `Deno-weak` curve in Figure 4.

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 in the kinds of environments we address. 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. Figure 5 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 [0.0-1.0]), all protocols perform fairly well, achieving commit percentages of around 100%. 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 whole range shown is small (within absolute 5%). 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

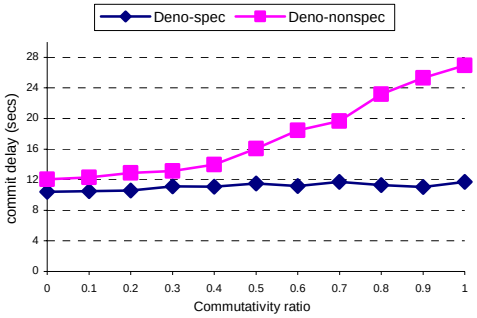


**Figure 6: Speculation effects on commit delay (15 servers, *SP*=5.0 secs)**

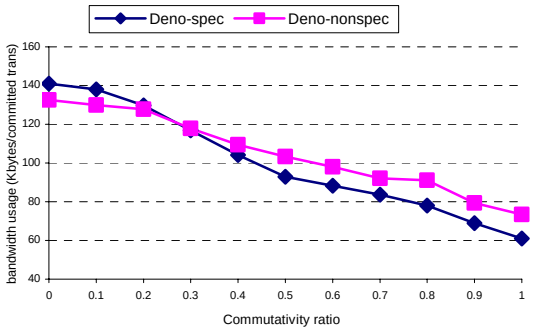


**Figure 7: Speculation effects on bandwidth usage (15 servers, *SP*=5.0 secs)**

average of 10% 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 >50% 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 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 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.

Figure 6 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 description of the modifications required to support speculation can be found in [7]). Somewhat non-intuitively, larger commutativity ratios result in larger commit delays for the non-speculative 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% even with a commutativity ratio of 0.0 –– 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 Figure 7, 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 .1), `Deno-spec` propagates about 4-6% more information per committed transaction. Beyond a commutativity ratio of .2, however, 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. To summarize, the speculative version not only decreases average commit delays, but it also decreases bandwidth requirements per committed transaction.

## 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., [1, 3, 8, 10, 20, 22]. Due to space limitations, we restrict our attention to asynchronous update-anywhere approaches that utilize the epidemic model [2, 9, 15, 19, 21]. Many epidemic systems take an optimistic approach and use reconciliation-based protocols that are only viable in non-transactional 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 [21] 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*. [2] described a pessimistic ROWA [3] 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.

Independent of our research, Holliday *et al.*[13] 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. In addition, we also describe a weak-consistency version of the protocol, and discuss how to propagate updates speculatively.

# 7 Conclusions

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 [21], and higher availability and performance than ROWA approaches [2].

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.

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.

# References


[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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, 1999.

[8] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a Partitioned Network: A Survey. *ACM Computing Surveys*, 17(3):341-370, 1985.

[9] 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.

[10] 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.

[11] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. 7th ACM Symp. on Operating Systems Principles (SOSP)*, Pacific Grove, 1979.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. ACM Intl. Conf. on Management of Data (SIGMOD)*, Montreal, June 1996.

[13] 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.

[14] 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.

[15] 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.

[16] P. J. Keleher. Decentralized Replicated-Object Protocols. In *Proc. 18th ACM Symp. on Principles of Distributed Computing (PODC)*, Atlanta, May 1999.

[17] 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.

[18] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, Amsterdam, 1989.

[19] 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.

[20] 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.

[21] 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.

[22] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180-209, 1979.