Support for Speculative Update Propagation and Mobility in Deno

Ugur Cetintemel  
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 Franklin  
Computer Science Division, EECS  
University of California, Berkeley  
franklin@cs.berkeley.edu

Contact author:
Dr. Peter Keleher  
Computer Science Department  
A. V. Williams Bldg.  
University of Maryland  
College Park, MD 20742-3255  
301 405-0345  
Fax: 301 405-6707  
keleher@cs.umd.edu

Category: Research
This paper presents the transactional framework of Deno, an object replication system specifically designed for use in 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. These features are all crucial to providing effective support for mobile and weakly-connected platforms.

Deno has been implemented and runs on top of Linux and Windows NT/CE 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. The results show that our protocols either outperform or perform comparably to existing approaches, while achieving higher availability. Further, we show that the incremental cost of providing global serializability in this environment is low. Finally, we show that commit delays can be significantly decreased by allowing votes to be cast, and votes and updates to be disseminated, speculatively.

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 communicate 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 [25, 26]. Asynchronous solutions [5, 11, 12, 24, 27] for managing replicated data have a number of advantages over traditional synchronous replication protocols in large-scale, mobile, and weakly-connected environments. They can operate with less than full connectivity, easily adapt to frequent changes in group membership, and make few demands on the underlying network topology. However, this functionality comes at a price. Asynchronous solutions are generally either slow and require reconciliations, or have lower availability because they rely on primary-copy schemes [32].

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.
Our prior work only defined consistency for single replicated objects. This paper extends bounded voting to include multi-item transactions and serializability. The voting protocol ensures mutual exclusion among conflicting transactions, guaranteeing that no two concurrent conflicting transactions can both commit. However, all transactions execute locally and no local or global deadlocks are possible.

The protocol’s strengths result from a novel combination of weighted voting and epidemic information flow [15], a process where information flows pairwise through a 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 [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., [18, 23, 35]), votes are cast and disseminated among system servers asynchronously through pair-wise, epidemic-style propagation. Any server can either 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 [13]. 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 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 are usually 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/Win32/WinCE 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 inexpensive in this environment. One of our protocols implements a form of weak consistency [10, 17], 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, by providing a global commit order on all update transactions. Under both protocols, read-only transactions execute entirely at the local server, and do not require network communication.

Finally, we show that casting and disseminating votes speculatively can significantly improve the performance of protocols based on epidemic or similar communication mechanisms.

The rest of this paper is structured as follows. Section 2 describes the Deno architecture and prototype. Section 3 describes the base protocol, and Section 4 describes the extended version. Section 5 describes Deno’s support for mobility, and Section 6 presents the results of our performance study. Finally, Section 7 briefly describes related work, and Section 8 concludes.

2. Deno prototype

This section briefly describes the basic architecture of Deno object replication system. The overriding goal of the Deno project is to investigate replica consistency protocols for dis- and weakly-connected environments. We are therefore not motivated to build large and complicated interfaces to the object system. By the same token, we feel that lightweight interfaces are the appropriate choice for many applications, and that more complex services can be efficiently built on top of Deno services if needed. The basic Deno API supports operations for creating objects, creating and deleting object replicas, and performing reads and writes on the shared objects in a transactional framework.

2.1 Architecture

Figure 2 illustrates the basic Deno server architecture, consisting of the following components:
• The Server Manager is in charge of coordinating the activities of the various components. It handles client requests by implementing the basic Deno API.

• The Consistency Controller implements the decentralized voting protocols used by Deno. In particular, it maintains a vote pool that summarizes the votes known to the server.

• The Synch Controller is responsible for implementing efficient synchronization sessions with other Deno servers by maintaining version vectors that compactly summarize the events of interests from other servers. This component implements different synchronization policies that specify when and with whom to synchronize. In the current implementation, it implements a naïve policy that chooses synchronization partners randomly at regular intervals.

• The Trans Manager is mainly responsible for the local execution of transactions. It maintains a transaction pool that contains all active (i.e., non-obsolete) transactions known to the server.

• The Storage Manager provides access to the object store that stores the current committed versions of all replicated objects at the server. The object store is currently a simple in-memory database.

The prototype makes relatively few demands on the operating system and is therefore highly portable. The current prototype runs on top of Linux and WindowsNT/CE platforms. All communication is made on top of UDP/IP. Deno’s source consists of ~10,000 lines of multi-threaded C++ code.

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 transaction is submitted by a client to any server, which executes it locally. Upon completion, the transaction either blocks (if the local server has seen a conflicting transaction) or becomes a candidate, which means that the update can become visible to other servers. Candidates are voted on, and eventually either commit (if they corner a plurality of the system currency), or are aborted. We now describe Deno’s replication framework employed in detail.

3.1 Providing weak consistency: Base protocol

3.1.1 Transaction model

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

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

The items in the local copy of the database are modified, and their version numbers incremented, only when update transactions commit. In other words, transactions only access committed values. Depending
on application semantics, however, this requirement can be relaxed, and transactions may be allowed to see new values written by uncommitted transactions. Such a model may especially be useful in facilitating disconnected operation, and has been investigated in the context of Bayou [34].

We distinguish between two types of transactions, queries (i.e., read-only transactions) and update transactions. Both types of transactions execute entirely locally. However, queries are more light-weight in that a query can commit without further processing immediately after it successfully finishes its execution. Update transactions, on the other hand, must participate in a distributed commitment process after finishing execution. When an active transaction successfully completes its execution, it takes one of the following two paths: (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 considered for becoming candidates. In Section 4, we eliminate this restriction mainly in order exploit commutativity.

3.1.2 Voting
Define $V_i$ as the set of all votes seen (the vote set) by Deno 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 the local $v.voter$’s local timestamp when $v.trans$ was created. A timestamp is a monotonically increasing local counter that is incremented each time the server casts a vote.

---

1 Servers can dynamically exchange currencies in order to improve availability and performance. This issue, however, is outside the scope of this paper.
Notice that the combination of “voter” and “trans” fields uniquely identify a vote. In the following, we omit the other fields of a vote when they are not relevant to the discussion.

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

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

1. \( t_i.vers(d) = t_j.vers(d), \forall d \in t_i.read \cap t_j.read \), and
2. \( (t_i.read \cap t_j.update) \cup (t_j.update \cap t_i.read) \neq \phi \).

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 it has not already voted for a conflicting candidate transaction. Such a vote is called a **yes** vote and is an indication of the support of the server for the corresponding transaction. Otherwise, \( s_i \) votes with 0.0 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 Deno server \( s_i \) maintains the following major data structures:

1. a set of votes, \( V_i \).
2. a list of **candidate** transactions, \( C_i \). \( C_i \) consists of those update transactions that are known to \( s_i \), have finished execution either locally or remotely, but have yet to be either committed or aborted at \( s_i \).
3. a list of **blocked** transactions, \( B_i \), consisting of locally completed transactions waiting to become candidates.
4. a commit log, which contains a list of committed transaction records.

A server may create a **vote** for a candidate or locally completed transaction that does not conflict with any other candidate transaction for which it has also voted. If the server votes for a blocked transaction, the transaction becomes a candidate transaction and is moved from the blocked list to the candidate list. Once made, 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:** it votes yes for \( t \) if it has not already voted for a conflicting transaction; Otherwise, \( s_i \) votes no.
2. **When \( s_i \) commits or aborts a candidate transaction:** it considers all transactions \( t \) in the blocked list (i.e., all transactions waiting to become candidates) in 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 existing conflicting candidate transactions.
3.2 Update commitment
Given a server $s_n$ and its vote set $V_n$, we make the following definitions:

**Definition 2** We define the sum of votes cast for a transaction $t$ as: $\text{votes}(t) = \sum v.curr$, where $v \in V_n$ and $v.trans = t$.

**Definition 3** We define the unknown votes of a transaction $t$ as: $\text{unknown}(t) = 1.0 - \sum s.curr$, where $s$ is a server who has already voted for $t$.

In other words, $\text{unknown}(t)$ is essentially the sum of currencies of those servers whose votes for transaction $t$ are not available. We now define the commitment rule that server $s_i$ uses to decide which candidate transactions to terminate (i.e., commit or abort) on the basis of local information (i.e., votes that are locally available). The fundamental idea is to commit a transaction when it is guaranteed that no other conflicting transaction can gather more votes. Note that $\text{server}(t)$ is defined as the server on which transaction $t$ was executed.

**Commit Rule:** A transaction $t \in C_i$ commits when:

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

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

When a candidate transaction $t$ commits at server $s_i$, server $s_i$ 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 those logs will grow indefinitely. The candidate transactions and votes, on the other hand, do not need to be garbage-collected because they are discarded as soon as they become obsolete.

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

**Abort Rule:** An active, blocked or candidate transaction, $t \in C_i$, is aborted at $s_i$ when there exists a data item $d$, s.t.: 1. $d \in t.read$, and 2. $s_i.vers(d) > t.vers(d)$.

where $vers(d)$ is the version of $d$ observed by a transaction or server. Additionally, committing a transaction causes all votes cast for an obsolete transaction to be discarded, i.e., a vote $v \in V_i$ is discarded if $v.trans = t$ and $t$ has become obsolete.

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

In Deno, synchronization is controlled via version vectors [28]. In our model, each server $s_i$ maintains an $n$-element vector, $vv_i$, where $n$ is the number of servers, that describes the number of events of each

---

2 This process is also called anti-entropy [15] in the terminology of epidemic algorithms.
other server seen by \( s_i \). Element \( vv_j[i] \) is a scalar count of the number of \( j \)'s events that have been seen at \( s_i \). For example, if \( vv_j = \{2, 3, 1\} \), then \( s_i \) has had two local events, has seen three events of \( s_j \), and has seen one event of \( s_i \)'s. There are three types of events: transaction commits, transaction promotions, and votes. A commit event is created whenever the local process commits a transaction, a promotion event is created whenever a transaction becomes a candidate on the server where it executed, and a vote event is created whenever a vote is cast.

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

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

Synchronization is then straightforward. For purposes of exposition, we assume a unidirectional pull synchronization, although other modes are possible [15, 26]. When \( s \) pulls information from \( s_j \), the following actions take place:

1. Server \( s_j \) sends \( vv_j \) to \( s \).
2. Server \( s_j \) responds with all events \( e_k^l \) s.t. \( l > vv_j[k] \) and \( l \leq vv_j[k] \), for all \( k = 1 \ldots n \).
3. Server \( s_j \) incorporates the new events in the same order that they originally occurred by:
   a. processing new commitments, candidates, and votes,
   b. applying the voting rule, the commit rule, and the abort rule for all relevant transactions,
   c. updating \( vv_j \) to the pairwise maximum of \( vv_j \) and \( vv_j \)

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

\[
vv_j = \{ (j, c_j), (k, c_k) \ldots \}
\]

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

### 3.4 Protocol illustration

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

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

\[
V_1=\{<s_1, t_1, 0.2>, <s_1, t_2, 0.0>, <s_1, t_3, 0.2>, <s_1, t_4, 0.0>, <s_2, t_2, 0.2>, <s_3, t_2, 0.25>, <s_3, t_3, 0.25>, <s_4, t_2, 0.0>, <s_4, t_4, 0.25>\},
\]

\[
V_2=\{<s_2, t_2, 0.2>, <s_4, t_2, 0.0>, <s_4, t_4, 0.25>\}.
\]

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

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

**Example 2:** Suppose that $s_4$ now pulls information from $s_1$. The events propagated from $s_1$ to $s_4$ consist of the commit of $t_2$, candidate $t_4$, and votes $<s_1, t_3, 0.2>$ and $<s_3, t_3, 0.25>$. The arrival of the commit decision causes $s_4$ to commit $t_2$, and to discard $t_4$. At the end of the synchronization, $V_2=\{<s_1, t_3, 0.2>, <s_3, t_3, 0.25>\}$. At this point, $s_4$ votes yes for $t_3$, and adds $<s_4, t_3, 0.25>$ into $V_4$. It then applies the commit rule, and commits $t_3$ since votes($t_3$)=0.70. Server $s_4$ then discards all relevant votes. Finally, $V_4=\{\}$.

### 3.5 Correctness and consistency issues
We now discuss the consistency level provided by the base voting protocol. First, we present the following lemma:

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

**Proof (Sketch):** Assume that transaction $t$, 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($t_i$), preventing $t_j$ from committing at $s_j$. Therefore, $s_j$ will eventually deduce the same outcome as $s_i$ and commit $t_i$ itself, or be told of the commitment of $t_i$ by another server.

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

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

1. **rw** ($t_2$ writes an item $d$ which is then read by $t_1$): Since $t_2$ updated $d$ when it committed at $s'$, the version number of $d$ recorded by $t_1$ will be strictly smaller than the version number of the copy of $d$ at the database of $s'$. This establishes $t_1$ as an obsolete transaction at $s'$ and leads to $t_1$ being aborted.
2. **wr** ($t_2$ reads an item written by $t_1$): This case is the 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. **ww** ($t_2$ writes an item written by $t_1$): This conflict type implies both rw and wr conflicts among $t_1$ and $t_2$. It is, therefore, subsumed by the previous two cases (since we do not allow blind-writes).

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

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

**Definition 4 (Weak Consistency):** A query sees weak consistency if it serializes with respect to all update transactions, but possibly not with other queries [9, 10, 17].

---

3 A serialization graph [9] consists of nodes that represent transactions and directed edges that represent conflicting operations. A path from $t_i$ to $t_j$ indicates that $t_i$ has to precede $t_j$ in any equivalent serial ordering.
In weak consistency, each query observes a serial order of update transactions, which is not necessarily the same order observed by other queries. However, weak consistency does ensure that queries always observe transactionally-consistent database states. In other words, a query does not see partial effects of any update transaction. Weak consistency prohibits both update transaction cycles (i.e., cycles involving only update transactions), and single-query cycles (i.e., cycles involving a single query and one or more update transactions).

**Lemma 3 (Query-Transaction ordering)** Let \( q \) be a query and \( t \) be an update transaction that respectively reads and updates item \( d \). The dependency \( q \rightarrow i \) 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 \( t \rightarrow q \); \( q \) reads \( d \) after \( t \) updates \( d \). In this case, \( q \) must have read \( d \) and committed after \( t \) since any update transaction (including \( t \)) installs its updates and commits atomically.

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

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

**4. Providing serializability: Extended protocol**

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

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

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

**Definition 5 (Strong Consistency)** A query sees strong consistency if it is serialized with respect to both queries and update transactions. Strong consistency is characterized by an acyclic serialization graph, prohibiting both update transaction cycles and multi-query cycles (i.e., cycles involving multiple queries and one or more update transactions). This form of consistency guarantees globally-serializable executions [9, 10, 17].

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\(^4\) by providing mutual exclusion among all transac-

\(^4\) In fact, there are at least two other approaches to provide strong consistency. One approach is to include queries in the global voting process, which is clearly not desirable in our target environments. A second approach is to commit an update transaction...
tions, rather than just among conflicting transactions as the base protocol does. We accomplish this by modifying the voting process such that each server votes yes for all candidate transactions (whether or not they conflict), but specifies a total order on all of it’s votes. The commit process is then restricted so that only the top transactions, which are the candidate transactions that come first in any server’s ordering, are considered for commitment.

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

**Definition 6** A vote $v \in V_i$ is said to be a top vote at server $s_i$ if:

$$v.vstamp < v'.vstamp, \forall v' \in V_i \text{ s.t. } v.voter = v'.voter.$$  

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

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

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

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

**Definition 8** We define the unknown votes as:

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

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

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

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

Aborts are handled as in Section 3.2. The following example illustrates how the extended protocol works:

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

$$D(t_1)=\{d_1, d_2^s\}, D(t_2)=\{d_3, d_5\}, D(t_3)=\{d_3, d_4^s\}, D(t_4)=\{d_1, d_3^s\}, \text{ and}$$

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

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

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

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

after it is certified by all servers (similar to Agrawal’s protocol [3]). This latter approach always requires contacting all servers in the system, which may be a serious restriction during times of low availability.
5. Support for mobility

This paper focuses on the performance of Deno’s commit protocols. For completeness, however, we briefly summarize some of Deno’s mobility-specific 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. The use of proxies can prevent degradation in the overall commit rate when devices have expected, planned-for disconnections. In fact, proxies can even improve commit latency because currency is concentrated in fewer servers, and fewer rounds of communication are required to establish a quorum. Proxies are implemented by having the proxy tell other servers that the primary’s vote is the same as its own while the proxy is engaged. A proxy vote is then indistinguishable to other servers from the situation where a server votes and then disconnects. When a primary reconnects, it updates its own information to match that of the proxy, including votes on prior and current tentative updates. The primary treats any votes cast in its behalf as if they had been cast directly. In particular, any votes cast for tentative updates remain cast. The result is that there are no race conditions, and the entire proxy engagement is transparent to the rest of the system.

Currency management

This paper assumes a uniform distribution of currency across all servers. In fact, 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. Since the total amount of currency in the system always remains constant, a server that provides currency to another must lose currency of the same amount. In general, servers do not know the number of servers in the system, and so cannot create an initial uniform distribution. However, the use of subsequent peer-to-peer currency exchanges allows the system to quickly approach any target distribution [13].

A second major question is the issue of how to deal with failures. Deno detects failures and network partitions through standard timeout mechanisms. Machines are declared “dead” by committing a retirement transaction that revokes the dead machine’s currency and redistributes it to other servers. This is effectively implemented as an involuntary proxy assignment.

Access control

The design of Deno’s security architecture includes authenticating servers using public-key cryptography when they first request copies of the database. All subsequent server-to-server communication is encrypted with private-key cryptography using a per-session key, where a session is defined to be the life of a particular database. However, this architecture is not fully implemented, and is not reflected in the performance data below.

Application-specific commutativity information

Disconnected and weakly-connected environments can be characterized by disconnections, high communication latencies, and incomplete connectivity. Therefore, applications running on top of these environments and systems need be designed so as to minimize conflicts among updates in order to avoid high abort rates [19]. One approach to accomplish this is to have the applications export domain-specific semantic information that can be used by the underlying system to modify application’s consistency requirements [34]. To this end, Deno’s extended protocol supports commutativity procedures to exploit application-specific commutativity information. A commutativity procedure is a simple query over the data-
base specifying an acceptance criterion [19]. If the query is satisfied, the transaction is considered as valid with respect to the current state of the database (event though it may already become obsolete in the traditional sense).

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

6. Performance evaluation
This section describes the performance of the Deno prototype. Note that the primary advantage gained in combining weighted voting with epidemic information flow is in increased availability. Rather than belabor the obvious, we instead investigate the performance impact of such a combination. Our earlier work [25, 26] presented simulation results showing the potential for good performance on single-object updates. This work presents results for multi-object transactional updates on a real system. Additionally, this work introduces and characterizes the performance impact of speculative voting and information dissemination.

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

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

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

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

Note that the bandwidth requirements for transactional and consistency data were negligible compared to that required for propagating updated values, so we do not consider this question further.
We now investigate the performance characteristics of our protocols using our prototype. We look at two versions of Deno’s protocol, Deno-weak (Section 3), and Deno-strong (Section 4). Additionally, we investigate two representative epidemic replication schemes from the literature. The first, primary, is an epidemic primary-copy scheme that uses a specialized primary server to serialize the updates, while propagating the updates using epidemic flow. This protocol is similar to that used in Bayou [34], but does not include advanced Bayou features such as dependency checks and merge procedures. In our implementation, we modeled this scheme simply by allocating all the currency at a single server. Note that primary-copy protocols trade availability for a presumed advantage in performance. One of our findings is that this performance advantage is not significant in protocols that use epidemic-style communication.

The second scheme, write-all, is a “Read-One, Write-All” (ROWA) [9] protocol, where servers can only commit transactions after ensuring that all other servers are ready to commit. Therefore, a transaction has to be propagated to all the servers before it can be committed. Furthermore, when a server receives conflicting transactions, it has to abort all of those transactions in order to ensure global consistency. A similar epidemic ROWA protocol was proposed by Agrawal et al. [3].

### 6.3 Commit delays

We first investigate the update propagation characteristics of different epidemic protocols. We begin by examining the first commit delay, which is the traditional commit delay denoting the time between the initiation of a transaction and the time it is first committed at a server. Figure 4(a) shows the first commit delays for the primary, write-all, and Deno. In primary, a transaction commits when the primary server receives the transaction (unless the primary decides to abort the transaction). Therefore, such a scheme clearly (first) commits a transaction much faster than a uniform voting approach, which requires a transaction to visit at least a majority of servers (assuming a uniform currency distribution). On the other hand write-all requires all servers to receive and certify the transaction, and performs poorly compared to either Deno protocol or primary.

Unlike traditional synchronous environments where transactions are committed synchronously at every server, commit times typically exhibit wide variability in asynchronous environments. The time at which the first server commits a transaction is, thus, not necessarily the quantity that best predicts application performance with epidemic information propagation. Since typically all servers have an equal chance of being accessed, an equally useful metric would be the average commit time, the time to commit a transaction averaged over all servers. Figure 4(b) presents the corresponding average commit delays for the primary, write-all, and Deno protocols. We see that write-all still performs significantly worse than the other protocols, and we also observe that the gap between Deno and primary does not exist anymore.

We further explore the reason behind the good performance of the Deno protocol by plotting the number of servers that committed the transaction as time progresses (for 15 servers) in Figure 5. 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 5. In an environment where updates are propagated asynchronously, average commit delay is as important as the first commit delay because committed transactions are only useful at a server when reflected in the local replicas. One important implication is that the performance penalty of using voting rather than a primary-copy approach is not as large as commonly assumed in the kinds of environments we address.

However, the most interesting results from this series of experiments is that the basic Deno protocol did not appear to have any advantage of the extended version. We had expected the stronger semantics of the extended protocol to exact a performance cost. Instead, the difference between the commit delays of the two with little contention appears to be negligible (as shown in Figure 4), and is only an average of 10% with significant contention. The case with contention was where we expected the most degradation in performance, as the requirement of a global ordering effectively increases the number of conflicts. This increase in conflicts, in turn, forces more 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.

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

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

Figure 6 presents the performance results of the protocols under update contention. More specifically, the figure shows the commit percentage (i.e., the percentage of initiated transactions that are committed)
results for different levels of transaction generation rate (for 15 servers) for all protocols. The figure shows that all approaches suffer from the increased transaction rate due to the global update consistency requirement that only one out of a set of conflicting transactions can commit. Under very small transaction rates ([0.0-1.0] trans/synch period), all protocols perform fairly well, achieving commit percentages of around 100%. With increasing transaction rates, however, commit percentages drop for all protocols significantly. We observe the most dramatic fall for **write-all**: at around a transaction rate of 1.0, the commit percentage of **write-all** is less than 50%, whereas the commit percentages of the other protocols are above 70%. Overall, **primary** achieves the best commit percentage, followed closely by the weak and strong versions of Deno. The difference between the two versions of Deno as well as the difference between Deno protocols and **primary** over the whole range shown is small (within absolute 5%).

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

### 6.5 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 only proceed and participate in the global voting protocol after the conflicting transactions are terminated. Such a conservative blocking approach is used by many pessimistic distributed protocols [9, 20].

An optimistic alternative is for the blocking phase to be skipped by having servers immediately vote for all transactions as soon as they finish their local execution. These transactions are immediately candidates to be added to subsequent anti-entropy 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 useful when previous conflicting transactions are aborted. As shown below, the advantage conferred by this technique is larger when there are commuting transactions in the system.

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

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

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

Interestingly, we do not need to make any modifications to the commit rules we described earlier. The two commit rule versions, both the weak- and the strong-consistency versions, transparently decide on which votes should be considered in the current commitment decision, thereby ignoring the speculated votes for the current commit decision. The strong-consistency rule utilizes timestamp information (already available at the votes) to take into account the only the top votes as current votes. The weak-consistency rule implicitly utilizes \textit{yes} and \textit{no} votes. For each transaction $t$, only the votes from a given server that have lower timestamps than the timestamp of that server’s vote for $t$ are used in the current commit decision.

Figure 7 examines the benefits of speculative update propagation and voting for varying degrees of commutativity by showing the performance of speculative (Deno-spec) and non-speculative (Deno-nonspec) versions of Deno-strong. A commutativity ratio of .25, for example, implies that 25% of transactions made obsolete (in the sense discussed in Section 3) run commutativity procedures whose acceptance criteria are satisfied. Somewhat non-intuitively, larger commutativity ratios result in larger commit delays for the non-speculative version of the protocol. The reason is that increasing commutativity results in fewer aborted transactions, which in turn increases contention for those transactions that have not been aborted.

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

As discussed above, the benefits of speculation come at the expense of propagating more transactions and votes. To this end, we also investigated the relative bandwidth requirements of both approaches. We
found out that Deno-spec indeed sends about 6% larger messages on the average (which turns out to be about 200 bytes/message more than the non-speculative case on average in our setting). Although the speculative approach uses more bandwidth per message, as demonstrated in the previous experiment, the commit rates for different approaches are quite different and need to be taken into account when comparing the relative bandwidth utilizations (e.g., how much useful work is accomplished per byte transmitted.) Figure 8 shows the amount of information sent across all servers (in KBytes) per committed transaction for Deno-spec and Deno-nonspec. For low commutativity ratios (up to .10), Deno-spec propagates about 4-6% more information per transaction commit. At a commutativity ratio of .20, both approaches propagate almost the same amount of information. Beyond this ratio, the speculative protocol sends less information than the non-speculative version, with the difference increasing as the commutativity increases. At a commutativity ratio of 1.0, Deno-spec propagates about 16% less information per committed transaction.

To summarize, the speculative version not only decreases average commit delays, but it also decreases bandwidth requirements per committed transaction.

7. 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, 2, 9, 14, 17, 32, 35] (see [20] for a compact survey of replication techniques in distributed environments). Due to the intrinsic shortcomings of traditional synchronous replication solutions [18, 32, 35] in large-scale, mobile, and weakly-connected environments [19], asynchronous replication protocols have recently gained a lot of attention (e.g., [5, 11, 12, 24, 27]). Asynchronous approaches commonly allow servers to execute updates locally without any synchronization with other servers, and propagating updates afterwards as separate activities.

According to Gray et al. [19], existing asynchronous solutions fall into two broad categories, master and group, with respect to how they regulate update creation and data item ownership. Master approaches require a data item update to be first performed at the primary copy of the item. The updates are then propagated to secondary copies. For instance, Breitbart et al. [11] recently proposed a master asynchronous approach that ensure serializable executions, and demonstrated significant performance improvements over master synchronous protocols in many cases. Group approaches, also called “update-anywhere”, allow any copy to be updated, thereby yielding maximum flexibility with respect to where and when to create updates. This flexibility is highly desirable in many collaborative groupware applications (e.g., CAD, CASE, Lotus Notes [24], etc.), especially when operating under mobile and weakly-
connected environments [19]. The cost of this flexibility is, however, increased update conflicts, and the complexity and overhead of maintaining consistency.

Many existing asynchronous “update-anywhere” protocols typically utilize the epidemic model [3, 15, 24, 29, 31, 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 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. For example, Rabinovich et al. [31] discussed an epidemic approach that enables eventual consistency of the replicas in a non-transactional framework. The main focus of the work was, however, on minimizing the overhead of the periodic synchronization sessions.

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] proposed a pessimistic “Read-One, Write-All” (ROWA) [9] approach that was the first epidemic protocol to ensure strong consistency and serializability. This protocol allows a transaction to commit only after all servers “certify” the transaction. Our protocols differ from these protocols primarily in using a novel combination of weighted-voting and epidemic information flow [15] to improve availability and performance. In Section 3.2, we also compare the performance of our protocol with variants of both primary-copy and ROWA type epidemic approaches.

The impact of mobility on data management and replication has been discussed in [4, 6, 7, 16, 22]. Pitoura and Bhargava proposed a two-level, cluster-based consistency model for intermittently-connected environments [30]. Barbara and Imielinski investigated different cache invalidation strategies for mobile wireless environments [8]. Huang et al. analyzed different data replication methods with the objective of minimizing the communication cost between the mobile and stationary hosts in a mobile environment [21].

Finally, we note that recent work [36] investigated why voting systems (i.e., quorums) have yet to become widespread in real-world applications. One of the conclusions is that voting does not enhance availability because either failures are positively correlated (when servers are on a single LAN) or network partitions occur (when servers are distributed across multiple LANs). In the latter case, a quorum constructed on a single LAN has higher availability than quorums constructed across multiple LANs. However, the weakly-connected environments addressed in this work fit neither category. Most failures (e.g., disconnections) are likely to be independent, and network partitions, while possible, are not the dominant cause of unavailability.

8. Conclusions

We have presented the design, implementation, and performance of Deno, a highly-available object-replication system that provides 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 has significant advantages over previous schemes in that it achieves higher availability than primary-copy approaches [34], and higher availability and performance than ROWA approaches [3].

Our base protocol ensures weakly-consistent executions [9, 10, 17], where update transactions are serializable and queries always access transactionally-consistent database states. Our extended protocol provides strong consistency [9, 10, 17] and globally serializable executions by providing a unique global commit order on all update transactions. Both protocols allow queries to be executed and committed en-
tirely \textit{locally}, and \textit{without blocking}. Furthermore, neither protocol suffers from local or global deadlocks (see the Appendix for details).

To a large degree, our focus on mobility and weakly-connected environments dictated both the policies that we enforce and the mechanisms that we use to pursue policy goals. To summarize, our focus on mobility led directly to the following decisions:

i) availability \textit{paramount} $\rightarrow$ weighted voting

ii) dynamic network topology and connectivity $\rightarrow$ epidemic information flow

iii) new applications + weak connectivity $\rightarrow$ performance/semantics tradeoff, commut. procs.

We performed a detailed study of the Deno prototype’s performance, particularly in regards to commit performance versus other protocols designed for similar arenas. One of the focuses of this study is the performance impact of the above tradeoffs. Comparing uniform voting and primary-copy variants allows us to characterize the cost of the higher availability provided by voting. Comparing the performance of our base and extended protocols allows us to assess the cost of the stronger transactional semantics.

Our main results are the following. First, the presumed performance advantage of the primary-copy approach over a uniform voting approach is nearly non-existent with asynchronous epidemic protocols. Average commit delays and commit percentages are nearly identical for the primary-copy protocol and even the basic, non-speculative version of the Deno protocol. 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 lack of a performance advantage for a protocol with weaker semantics and fewer restrictions on update commits is perhaps our most surprising finding. The result is increased functionality at 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. This latter point is especially important as the database community is coming to the realization that non-transparent replication is crucial to good performance in dynamic and mobile environments [33].

This paper has assumed a flat organization of Deno servers. As with any distributed system, flat organizations, especially peer-to-peer models, suffer from low scalability. Significant performance improvements can be attained when a hierarchical synchronization topology is used to organize servers. Support for hierarchical organizations will form the basis of our future work.
References


Appendix

Termination Issues

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

Proof (Sketch): Suppose there exists a candidate transaction \( t \) that never terminates. We can partition the set of servers into three disjoint subsets as (1) the servers that voted \( \text{yes} \) for \( t \), \( \text{yes}(t) \), (2) servers that voted \( \text{no} \) for \( t \), \( \text{no}(t) \), and (3) servers that have not yet observed \( t \), denoted \( \text{unknown}_\text{servers}(t) \). Assuming that information eventually propagates to all servers, \( \text{unknown}_\text{servers}(t) \) will eventually become empty. Let the conflict set of \( t \), \( \text{CS}(t) \), denote the set of candidate transactions that conflict with \( t \). When \( \text{unknown}_\text{servers}(t) \) becomes empty, \( \text{CS}(t) \) can not grow further due to the voting rule (see Section 3.1.2), since all servers voted for either \( t \) or another transaction that conflicts with \( t \). Now consider the case where all candidate transactions \( t' \in \text{CS}(t) \) are observed at all servers. At this point, \( \text{votes}(t) \) and \( \text{votes}(t') \) for all \( t' \in \text{CS}(t) \) are determined. As a result, \( \text{unknown}(t) \) and \( \text{unknown}(t') \) for all \( t' \in \text{CS}(t) \) are all 0.0. Therefore, the commit rule will commit the transaction with the most votes (or in the case of a tie the one executed at the server with the smallest id) and the rest will be aborted, thereby contradicting our initial claim. Moreover, a deadlock situation due to vote dependencies can not exist. Such a deadlock has to involve a cycle of the form \( \text{votes}(t_1) > \text{votes}(t_2) > \ldots > \text{votes}(t_n) > \text{votes}(t_1) \) where \( t_1, t_2, \ldots, t_n \) are candidate transactions. Since both \( \text{votes}(t_1) < \text{votes}(t_n) \) and \( \text{votes}(t_n) < \text{votes}(t_1) \) can not be true at the same time, we conclude that such a deadlock cannot exist.

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

Lemma 5  (Eventual Termination – Extended Protocol) A candidate transaction eventually terminates (i.e., commits or aborts).

Proof (Sketch): We consider two cases. First consider a top transaction, \( t \). Assume eventually that the top votes cast by all servers are known (i.e., \( \text{unknown}=0.0 \)). Let \( U \) denote the non-empty set of top transactions that obtained more votes than the remaining top transactions. If \( U \) contains a single transaction, then that transaction commits. Otherwise, \( U \) contains a set of transactions and the top transaction whose execution server has the smallest id commits. If \( t \) is the transaction that commits, then we are done. Otherwise; if \( t \) becomes obsolete, then it gets aborted, or else \( t \) remains a top transaction. The process repeats and \( t \) either becomes obsolete and is aborted, or eventually gets enough top votes and commits. Now consider a non-top transaction, \( t \). Transaction \( t \) will either become obsolete and get aborted by the commit of a top transaction, or eventually become a top transaction itself and terminate (by the discussion in the first case). Therefore, we conclude that a candidate transaction always terminates.

Correctness of the Extended Protocol (Section 4)

Lemma 6 (Global Update Consistency) The protocol presented above 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 protocol presented above provides strong consistency and serializability.

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

\[
q_1 \rightarrow t_1 \rightarrow q_2 \rightarrow t_2 \rightarrow \ldots \rightarrow q_n \rightarrow t_n \rightarrow q_1.
\]

Consider \( q_1 \rightarrow t_1 \), which implies that there is an item \( d \) read by \( q_1 \) and then updated by \( t_1 \). By Lemma 3, \( q_1 \) commits before \( t_1 \) at the execution site of \( q_1 \), say \( s_1 \). Now consider \( t_1 \rightarrow q_2 \) and \( q_2 \rightarrow t_2 \), which together imply that \( t_1 \) commits before \( q_2 \) and, therefore, before \( t_2 \) at the execution site of \( q_2 \), say \( s_2 \). Therefore, by Lemma 1, \( t_1 \) commits before \( t_2 \) at all sites. Using a straightforward induction, we can say that \( t_1 \) commits before \( t_n \) at all sites. However, \( t_n \rightarrow q_1 \) implies that \( t_n \) commits before \( q_1 \) at \( s_1 \), creating the contradiction that concludes the proof.