A Security Infrastructure for Mobile Transactional Systems

Peter J. Keleher, Bobby Bhattacharjee, Kuo-Tung Kuo, and Ugur Cetintemel

Dept. of Computer Science
University of Maryland
{keleher, bobby, ktg, ugur} @cs.umd.edu

In this paper, we present an infrastructure for providing secure transactional support for mobile databases. Our infrastructure protects against external threats — malicious actions by nodes not authorized to access the data. The major contribution of this paper, however, is to classify and present algorithms to protect against internal *security threats. Internal threats are malicious actions by authenticated nodes that misrepresent protocol specific information. We quantify the cost of our security mechanisms in context of Deno: a system that supports object replication in a transactional framework for mobile and weakly-connected environments.*

Our results show that protecting against internal threats comes at a cost, but the marginal cost for protecting against larger cliques of malicious insiders is low. However, even with all the security mechanisms in place, our system commits updates over 50% faster than systems that depend on the Read-once Write-all commit protocol. Lastly, we present results from a probabilistic version of our algorithm that has several orders of magnitude lower computation cost than the traditional public-key based schemes.

1. Introduction

We present an infrastructure for providing secure transactional support for mobile databases. In particular, we concentrate on providing solutions to *internal* security threats, and quantifying their costs.

This work is done in the context 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. The server labeled "(CA)" is a *certificate authority*, which will be explained in Section 3.

Deno's underlying protocols are based on an asynchronous protocol called bounded weighted voting [1, 2]. Asynchronous solutions [3-7] 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 [8].

Deno's protocol retains the advantages of current asynchronous protocols, but generally performs better, has fewer connectivity requirements, and provides 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 or intermittent connectivity.

Despite the good performance, however, no such system could be widely deployed in mobile environments without ensuring that the infrastructure is secure. We distinguish between internal and external security threats. The prime *external* threat is of an unauthenticated server attempting to read or modify data. We prevent this through public-key cryptography mechanisms. A request for data or

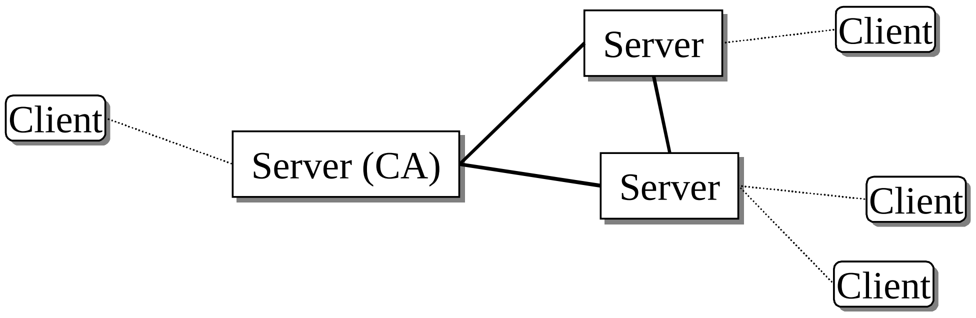

Figure 1: Basic Deno system model

protocol information must be accompanied by a signed hash of the request. The destination verifies the hash via the server's certificate, which is signed by a trusted Certificate Authority (CA) and is appended to the request. Data privacy can be provided by conventional symmetric encryption algorithms, such as Triple-DES or IDEA.

Dealing with *internal* threats to security is much more problematic. Internals threats arise from duly authenticated servers that attempt to cheat. As a trivial example, a user of a distributed meeting room scheduler might attempt to falsify votes of other servers in order to ensure that he or she gets a prime reservation. More serious scenarios could arise in collaborative intranet and Internet applications, such as scheduling and workflow applications. Finally, this work has obvious applications in military scenarios. Consider communication among tanks or mobile command posts. There is a clear need for secure, highly-available, replicated, consistent data, which is not easily met using traditional protocols.

The base, non-secure Deno system has been fully implemented. Deno's source consists of ~10,000 lines of multi-threaded C++ code. We have also fully implemented the *write-all* protocol discussed below, as well as the changes to the basic protocol needed to tolerate malicious servers. However, we are still building the public-key infrastructure that will be used to address external threats.

The rest of the paper is structured as follows. Section 2 briefly describes the design and performance of Deno's asynchronous protocol. Section 3 describes a public-key based infrastructure that addresses external threats by providing secure authentication and encryption without compromising Deno's ability to make progress with low or non-existent connectivity. Section 4 describes our approach to handling internal threats, which is the main contribution of this paper, and Section 5 evaluates the effect of our security measures on commit performance. Finally, Section 6 describes related work and Section 7 concludes.

2. Background: Deno

Deno is a replicated-object system that relies on a decentralized, asynchronous replica management protocol to addresses concerns of performance and reliability. Under Deno, 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 novel combination of weighted voting and epidemic information flow, a process where information flows pairwise through a system like a disease passing from one host to the next [9]. The protocol is completely decentralized. There is no primary server that "owns" an item or serializes the updates to that item (as in Bayou [10]). 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., [11-13]), 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 [14]. 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 any potential single point of failure.

The protocol is defined for both single-object updates and serialized multi-item transactions [15]. 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.

2.1 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.

2.2 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.

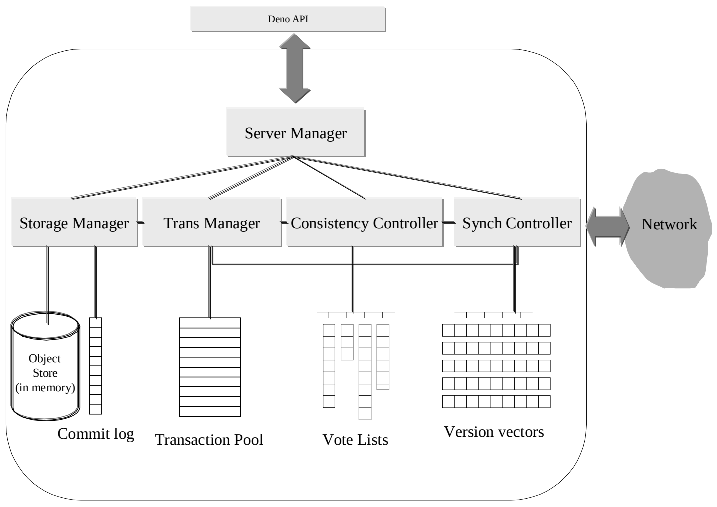

Figure 2: Basic Deno architecture

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 layered atop UDP/IP.

2.3 Protocol overview

At its simplest, Deno can be thought of as a set of servers that are cooperating in order to determine a sequential ordering of *committed* updates. Asynchronous voting is used to determine which updates actually commit. Asynchronous information pulls between randomly selected pairs of servers move

Definition 1: *Define* uncommitted(v_i) *as:* $\sum_{j=1}^{n} v_i.curr[j]$, *s.t. $v_i[j]$ is equal to* $\perp$.

Definition 2: *Define* votes(v_i, k) *as* $\sum_{j=1}^{n} v_i.curr[j]$, *s.t. $v_i[j]$ is equal to k.*

Definition 3: *A candidate c_j wins v_i's current election when:*

1.	*votes(v_i, j) > 0.5, or*	// c_j gathers *majority* of votes
2.	$\forall$ *k ≠ j, votes(v_i, k) + uncommitted(v_i) < votes(v_i, j) or*	// c_j gathers *plurality* of votes
	((votes(v_i, k) + uncommitted(v_i)) = votes(v_i, j) and (j < k))	// tie-break case

Figure 3: Definitions

newly created update records, together with votes for such, among the servers. There are a number of potential performance problems with this model. We will discuss these briefly at the end of the section.

We assume a model in which the shared state consists of a set of objects that are replicated across multiple servers. Objects do not need to be replicated at all servers, and servers may replicate multiple objects. For simplicity of presentation, however, we limit our discussion to single objects that are cached at all servers. Our discussion is easily extended to include the more general case.

Deno supports strict serializability between arbitrary multi-item transactions and queries. However, the single-object/transactional axis is orthogonal to the main thrust of this paper, so we restrict our discussion to single-object updates for pedagogical reasons.

Individual objects are modified by *updates*, which are issued by servers. An update consists of either a code fragment or a run-length encoding of binary changes. Updates can be transmitted to other servers and are assumed to execute atomically at remote sites. Given a consistent initial state, application of the same updates in the same order on multiple replicas of the same object result in the same final object state.

Updates do not commit globally in one atomic phase because we assume an epidemic style of updates and poor connectivity. Instead, each server commits updates based on local information. However, we show below that any update that commits at any server eventually commits everywhere, and in the same order with respect to other committed updates.

2.4 Elections

A clean way of thinking about update commitment is as a series of elections. A server is analogous to a voter, creating an update is analogous to a voter deciding to run for office, and a committed update is analogous to a candidate winning the election. Voters (and hence candidates) have indexes *0* through *n-1*, where *n* is the total number of voters. We use v_i to refer to the voter with index *i*, and c_i to refer to the candidate with index *i*. Candidates win elections by cornering a plurality of the votes. Each election begins with an underlying agreement of the winners of all previous elections. Once an election is over, a new election commences. Any given election may have multiple candidates (logically concurrent tentative updates), and candidates from different elections might be alive in the system at the same time. In the latter case, however, uncommitted candidates for any but the most recent election have already lost, but this information has not yet made it to all voters.

Because of the style of information flow, there is no centralized vote-counting. Instead, each voter independently collects votes from other voters and deduces outcomes. This creates situations in which the "current" election of distinct servers is temporarily out of sync. Voter v_i's current election is the election for which v_i is collecting votes. In order to implement this protocol, each voter maintains three pieces of state:

1. *v_i.completed* – the number of elections completed locally, and
2. *v_i.[j]* – is either the index of the candidate voted for by v_j in v_i's current election, or $\perp$, which means that v_i has not yet seen a vote from v_j.

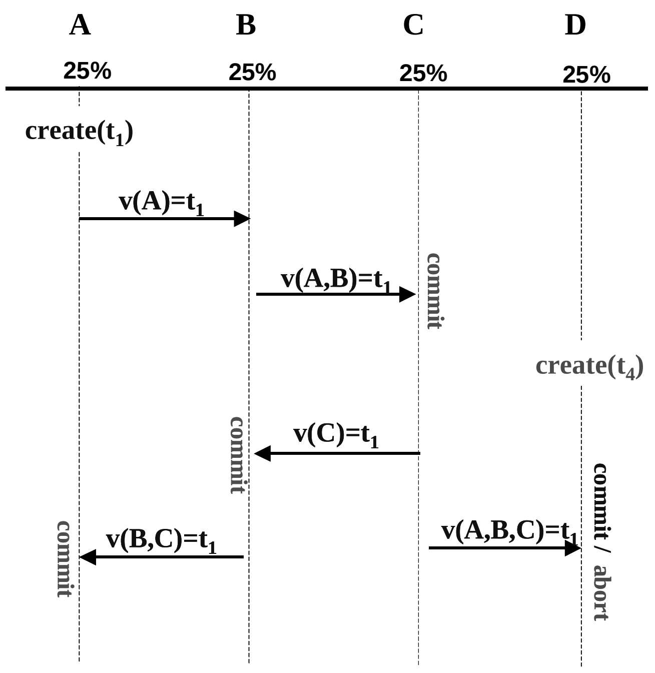

Figure 4: Update commitment

The size of the array is bounded by the total number of voters.

3. *v_i.curr [j]* – The amount of currency voted by v_j in v_i's current election or ⊥, which means that v_i has not yet seen a vote from v_j. Currency allocation may change with each election.

The total amount of currency in any election is 1.0.

Definitions 1-3 essentially say that a candidate is committed if it has a plurality of the vote at a given server. Ties are broken with a simple deterministic comparison between the indexes of the servers that created thee competing updates. The winner of the j^{th} vote at v_i is denoted *v_i.commit(j)*. When an election is won at v_i, all votes *$v_i[j]$* are reset to ⊥.

It follows naturally from the above definitions that candidates can win without all the votes being known. Similarly, updates can be committed by a server without complete knowledge of which servers have seen the update, or even complete knowledge of which servers cache the object.

More details are provided in [1] and [15].

2.5 Example

Consider Figure 4. The system has four servers, all with currency of .25. Server s_A creates a new update, t_1, implicitly votes for it, and sends a message describing t_1 and its vote to s_B via an anti-entropy session. s_B votes for t_1, and then later transfers notice of t_1 and both votes to s_c. After adding its own vote, s_c can commit t_1 because it has gathered a plurality. Later anti-entropy sessions move the votes back to s_B and s_A, which also reach the same commit decision.

Meanwhile, s_D has created a conflicting transaction t_4. Eventually, s_D learns of t_1 and aborts t_4. It is irrelevant that t_4 is actually created after t_1 has been committed elsewhere in the system.

Note that this example differs slightly from the real system in that anti-entropy targets are actually chosen randomly, and that a tie-breaking procedure would allow s_B to commit before talking to s_C.

3. External threats

We define an *external* security threat as one that is posed by a principal that has not been authenticated into the system. We first discuss authentication, and then integrity and privacy. A principal is authenticated into the system by identifying itself to a certificate authority (CA), which responds with an *access certificate* that specifies the principal's rights in the system. Certificates may provide either *read* or *read/write* permission for a given database, and may contain a timestamp that delimits the certificate's lifetime. Since a certificate is signed by the CA, any server with the CA's public key can verify that the certificate is valid, and certificates can not be forged. Note that we assume a priori that all servers trust the CA, and know the CA's public key.

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

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

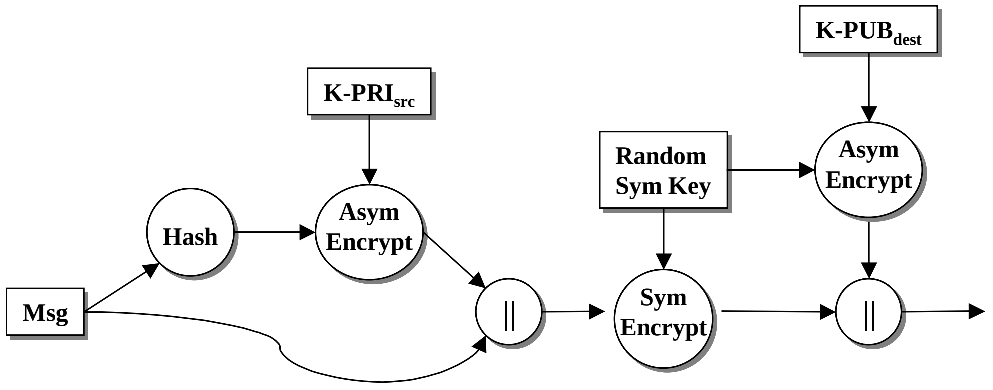

Figure 5: Integrity with privacy: We use symmetric encryption (i.e. Triple-DES) to encrypt the message, and asymmetric encryption (i.e. RSA) to encrypt the symmetric key and sign the message. K-PRI$_{src}$ and K-PUB$_{dest}$ are private and public keys of the source and destination of the message, respectively. Double bars indicate concatenation.

others cannot make copies unless they already have certificates, or are currently connected to the CA.

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

We allow certificates to be revoked via the issue of a *certificate revocation list* (CRL) from the primary CA. This presents problems because Deno servers have no notion of simultaneity, unlike secure multicast trees and other analogous systems. In other words, given that a CRL has been issued, when are revoked certificates guaranteed to be denied? We solve this problem by casting the issue of a CRL as just another update transaction. The CRL update competes with other transactions to win an election. Once the CRL update has been committed, we can guarantee that no subsequent update will be committed with the aid of a vote authenticated by a revoked certificate. A secondary advantage of casting the CRL issue as an update is that it guarantees quick dissemination. Otherwise, knowledge of the CRL might disseminate quite slowly because the CA is not consulted during the normal course of events.

3.1 Integrity and privacy

Figure 5 shows Deno's approach to providing both integrity and privacy guarantees for communicated data. Note that this method is very similar to the method used in PGP. Integrity is provided by appending a message authentication code (MAC) to each message, in this case the MD5 hash of the message signed by encrypting with the source's private key. Privacy is provided by encrypting the message and the MAC with a randomly generated, one-time *session key*. The session key is then encrypted with the destination's public key, and the concatenation of the encrypted session key, MAC, and message are sent to the destination.

The use of peer-to-peer one-time session keys[1] allows us to avoid the key changing problem incurred by secure multicast trees. Secure multicast trees generally use a single session key for the entire group. Any change in group membership requires the session key to be changed. The key must be changed when s_1 is added to the group because we do not want s_1 to be able to read messages that were sent prior to its joining (we assume that s_1 might have recorded prior encrypted messages even though it could not read them). Similarly, the key should be changed when s_1 leaves the group because we do not want s_1 to be able to read messages that are sent after it leaves the group. A similar need could exist in a Deno replication group, but is avoided by the use of peer-to-peer session keys.

4. Internal threats

In this section, we consider internal threats — threats that result from authenticated but malicious servers. Such malicious insiders misrepresent protocol-specific information, and can cause potentially corrupt objects to propagate throughout the network.

[1] Note that these peer-to-peer keys need not be one time; instead they may be cached and re-used later.

Under certain circumstances, even a single malicious insider with arbitrarily small amount of currency can cause different transactions to be committed at different nodes. We begin with a discussion of the set of malicious actions a node can undertake.

4.1 Malicious actions

Before we classify the actions a malicious intruder can take, we should note that malicious nodes can always commit arbitrary transactions to their *local* databases without even advertising the transaction to other nodes. Malicious servers can also remain within the protocol framework and issue updates that obscure or undo the effects of other updates. This type of behavior must be handled in an application-specific manner and is beyond the scope of this work. The goal of this section is to classify and mitigate the damage malicious nodes can inflict on other nodes. Malicious servers can only corrupt the view of other nodes by incorrectly reporting votes from other servers, or incorrectly reporting its own votes.

Under certain circumstances, a denial-of-service attack can be accomplished by even one malicious server refusing to vote its currency. This is handled by the normal *currency revocation* mechanism that is used to recover from failed servers [14].

4.1.1 Currency misrepresentation

The problem here is of a server misrepresenting the amount of currency it has available to vote in an election. This is possible because the system uses *peer-to-peer currency exchanges* to migrate currency allocations towards a target distribution[2]. A peer-to-peer exchange is used by two servers to reallocate their currency between them. This is a local operation, and cannot be directly verified by other servers.

We make this operation secure by requiring each currency exchange to be formalized as an update. A “gift” from s_i to s_j is only considered complete when an “exchange update” has been committed. Note that such updates are commutative with respect to all other updates and are therefore committed more quickly than ordinary updates.

[2] It is not generally possible for the initial allocation to result in a uniform distribution unless complete information about the set of servers is known a priori [14]. Even if this were true, it is often desirable to change the distribution to respond to dynamically changing conditions.

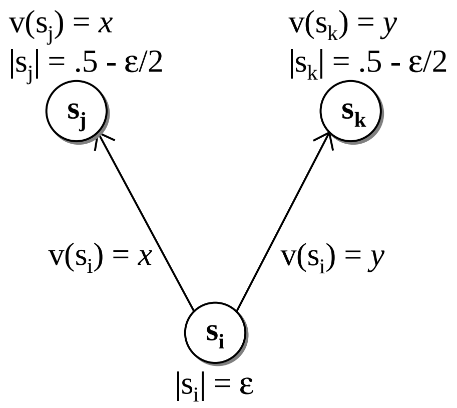

Figure 6: By telling s_j and s_k different votes, s_i can cause them to commit conflicting updates. $|s|$ is the currency held by s.

4.1.2 Vote misrepresentation

There are two types of vote misrepresentation: In the first case, a malicious server s_m misrepresents or forges some other server s_i’s vote to a third server s_j. This can happen when s_i and s_j are connected through s_m, s_i reports its vote to s_m and s_m forges this vote and reports a different vote for s_i to s_j. This type of malicious behavior can easily be prevented by requiring each server to sign its votes using a suitable digital signature technique. The worst a malicious server can do then is to never report s_i’s vote to s_j. Since Deno does not impose any specific connectivity requirements, this behavior will only delay committing of transactions and will not affect correctness.

The second vote misrepresentation is more difficult to guard against and can quite easily be used to violate all correctness guarantees. In this case, a server (possibly signs) and illegally votes its own currency more than once for multiple transactions. Consider the example shown in Figure 6. Assume that server s_i is malicious. If s_i tells s_j that it votes for x, and s_k that it votes for y, then both destinations reach the conclusion that their candidates have more than 50% of the vote and can be committed. Further, secure signed votes do not help in this case since s_i can properly sign its own vote for any transaction. In the rest of this section, we investigate approaches to detecting such malicious nodes, and develop an algorithm that guarantees correctness at all non-malicious nodes.

$\lvert\text{votes}(t_i)\rvert > \lvert\text{votes}(t_j)\rvert + \textit{unknown},\ \forall j \neq i$	*insecure system*
$\lvert\text{valid}(t_i)\rvert+\lvert\text{unvalid}(t_i)\rvert - \lvert\text{top}(t_i)\rvert > \lvert\text{votes}(t_j)\rvert + \lvert\text{top}(t_i)\rvert + \textit{unknown},\ \forall j \neq i$	*secure system*

Table 1: Commit criteria

4.1.3 Approaches for handling internal threats

We present a new algorithm that (a) guarantees correctness and (b) allows progress under contention *even when not all votes have been reported.* We first develop a version that is parametric in the number of malicious servers in the system. In Section 5.2, we describe an extremely efficient probabilistic counterpart that also accepts the probability of a vote being forged as a parameter.

4.1.4 Algorithm description

Our approach hinges on the following key observation:

> *Up to* k *malicious servers can be kept from corrupting an election if the* k *largest unvalidated votes are not used in any commit decision.*

Consider the following example: if there is a single malicious server, then any single vote may be a duplicate. For a given transaction, a server accumulates a set of validated votes and a set of unvalidated votes. The server can commit the transaction if the transaction can obtain plurality *without* counting the largest *unvalidated* vote for that transaction. More precisely, the transaction must have a plurality even if its largest unvalidated vote is cast for any *other* transaction. This follows since (i) validated votes cannot be duplicates and (ii) of the unvalidated votes, at worst the largest unvalidated vote may be a duplicate. Therefore, this worst case duplicate vote cannot be counted towards the commit decision at this node but instead must be thought of as a duplicate and given to the best known competing transaction.

Formally, let **votes(t_i)** denote the set of votes for transaction *i*. In general, **votes(t_i)** consists of validated votes and unvalidated votes. We denote the set of validated votes for t_i by **valid(t_i)** and the set of unvalidated votes by **unvalid(t_i),** respectively. Note that we consider votes by the local server to be validated votes. We denote the currency of any vote *v* in **votes(t_i)** by $|v|$. Similarly, we denote the total currency for a set *S* of votes by $|S|$, e.g., **|votes(t_i)|** denotes the total currency of all votes for transaction t_i.. Let **top(t_i)** be the element with the largest currency in **unvalid(t_i)**.

If we consider all votes in the base Deno system to be validated, then the base commit criteria for transaction t_i can be stated as in the top row of Table 1, where *unknown* is defined as $1 - \sum_{\forall k \neq i,j}$ **|votes(t_k)|.**

In order to provide resilience against a single malicious server, this base commit criteria is generalized as in the second row of Table 2. Thus, the number of votes required to commit a transaction t_i must be larger than votes for any other transaction t_j even if the largest (unvalidated) vote for t_i is voted for any other transaction t_j. This technique generalizes to an arbitrary number of malicious servers. If the server knows of no other transactions t_j, but it has not yet seen votes from all other nodes, then it simply assumes all unknown votes are cast for some other transaction (analogous to the quantity *unknown* in the base commit criteria). For *k* malicious servers, **top(t_i)** simply needs to be re-defined as the set of votes with *k* largest currency shares in **unvalid(t_i)**. Note that this criteria is equivalent to the base commit criteria if we set *k* equal to zero, which case **top(t_i)** is the null set.

In order to validate a vote for transaction t_x from a server s_j, a server s_i must ensure that all other servers in the system have seen the same vote. Thus, for each election, server s_i must collect *receipts* of the vote cast by s_j to all other servers. A *receipt* of server s_j's vote for election *n* from server s_k is a statement of the form "Server s_j votes for transaction t_x in election *n*", securely signed by server s_k using an appropriate digital signature. Server s_i considers a particular vote valid if and only if it has received receipts for that vote from all other servers in the system or if the vote is cast by server s_i itself. In order to validate a vote, a server s_i does not need to establish a peer-to-peer connection with all other servers in the system — instead, receipts for votes from any server can be forwarded by any other server in the system. Since the receipts are protected by strong cryptographic primitives, even malicious servers will not be able to alter the contents of the receipt. Malicious servers may corrupt or discard receipts: corrupt receipts will be detected by the server validating the receipt, while discarded receipts will be treated as any lost message. In the worst case, malicious servers may able to affect the liveness properties of the algorithm, but once again, we have been able to restore the safety guarantees[3].

4.1.5 Examples and discussion

In this section, we illustrate, via a set of examples, some of the more subtle properties of the secure plurality algorithm. We begin with a simple example of applying the secure protocol to the three server case shown in Figure 6. We had shown earlier that if server s_i is malicious, in the base protocol, under appropriate circumstances, it could cause the committed views of servers s_x and s_y to diverge arbitrarily far *even if it held arbitrarily small amount of currency in the system.* Now we show that even if server s_i is malicious and holds arbitrarily *large* amounts of currency in the system, it cannot cause a *single* incorrect commit at either servers s_x or s_y, as long as servers s_x and s_y operate under the assumption that there are malicious servers in the system. Assume s_i holds an arbitrary amount (say $a < 1$) of currency. Once again, assume the rest of the currency is distributed equally between servers s_x and s_y (the analysis for the other cases are analogous and is omitted for brevity).

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

In the following examples, assume the secure commit criteria is used with the assumption that there is at most one malicious server in the system. The first example shows that even under contention (i.e. when there are more than one transaction vying for the same election), the commit criteria does not necessarily require any votes to be validated before a transaction is committed.

Example 1: Assume five servers (s_1, s_2,…, s_5) in the system, each holding equal (0.2) currency, and the following votes:

V_1={<s_1, t_1>, <s_2, t_1>, <s_3, t_1>, <s_4, t_1>,<s_5,$\mathbf{t_2}$}

In terms of the new commit criteria:

|votes(t_1)| = 0.8,
|valid(t_1)| = 0.2 (local vote),
|unvalid(t_1)| = 0.6, and
|top(t_1)| = 0.2.

In this case, s_1 can commit transaction t_1 without validating a single vote!

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

Example 2: Assume servers s_1-s_4 have currency 0.2, **0.4**, 0.2, and 0.2, respectively. Votes at s_1 are:

V_1={<s_1, t_1>, <s_2, t_1>, <s_3, t_1>, <s_4, t_2>}

In terms of the new commit criteria:

|votes(t_1)| = 0.8,
|valid(t_1)| = 0.2 (local vote),
|unvalid(t_1)|= 0.6, and
|top(t_1)| = 0.4.

s_1 can not commit t_1 because:

|valid|+|unvalid|-|top| = 0.4, whereas
|votes(t_2)|+|top(t_1)| = 0.6.

Validating s_3's vote would have no immediate utility. However, if s_2's vote were validated instead, the commit could take place.

As can be seen from the secure commit criteria in Table 1, validating a vote can only have an immediate effect on a commit decision if it affects |top(t_1)|. Validating s_2's vote had such an effect; validating s_3's did not.

5. Performance evaluation

This section describes the performance of the Deno prototype. We performed the experiments on a 16 node Linux cluster with each node running a copy of the Deno server. Each node is contains two 400 MHz Pentium II processors and 256 MBytes of RAM. However, none of the results presented below consume all of a machine's resources. We intentionally set our communication rates low in order to reflect the constraints of our expected environment. Instead, our performance evaluation concentrates on relative performance by comparing the convergence rates of representative protocols.

[3] Note that we stated that we wanted to provide absolute, non-probabilistic, guarantees. Our scheme relies on the integrity of the digital signature: any digital signature scheme has a probability of failure at least inversely proportional the size of the key-space from which the encrypting and decrypting keys are chosen (exploited via a brute-force attack). Thus, more precisely, our scheme is provides guarantees only as strong as underlying digital signature scheme.

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

Table 2: Primary experimental parameters and settings

The machines were connected via a dedicated 100Mbps Ethernet network and the Deno servers communicated using UDP. 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.

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. Currency is uniformly distributed across servers except as noted. All objects are replicated at all servers. The main parameters and settings used in the experiments are summarized in Table 2.

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

For context, we also show the performance of a second scheme, `write-all`, which is a “Read-One, Write-All” (ROWA) [16] protocol modeling the best other transactional epidemic protocol in the literature. This protocol 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.* [17].

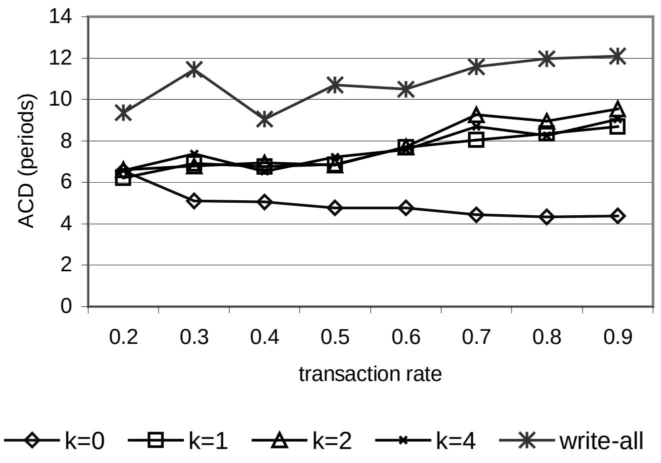

Figure 7: ACD vs. transaction rate

5.1.1 Performance Metrics

The primary performance metric we consider is *average commit delay*, which denotes the time between the initiation of a transaction and average the of the times at which it is committed by individual servers in the system. As a measure of scalability, we report the change in commit delay as the number of nodes in the system change. As a measure of robustness, we measure commit delay as the currency distribution in the system becomes non-uniform. In each case, we consider the efficacy of our algorithm by varying the number of malicious servers and where applicable, compare our results to the Write-all scheme.

5.1.2 Commit delays vs .malicious nodes

Figure 7 shows the average commit delays for the Deno protocol versus the number of tolerated malicious servers. The *x*-axis is the average number of transactions generated per server, per second, for 16-node runs. The *y*-axis shows the commit delay, normalized to the anti-entropy period. For comparison, we also include the average commit delay for the write-all scheme [16]. Note that commit delay actually goes down for the default case (no malicious servers). This is because additional transactions in the system, in essence, gather votes for a given election in parallel.

The result shows the following key points: the extra checks added to verify against duplicate votes increase the average commit time by about 50% for

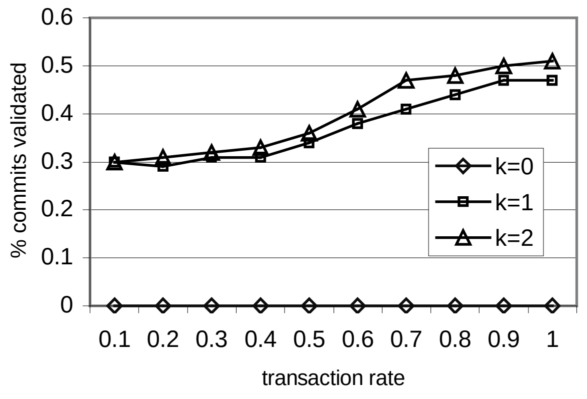

Figure 8: Upper bound on percent of commits that need at least one validated vote.

low transaction rates, and by a factor of 2 for high transaction rates.

However, even the secure Deno protocols perform significantly better than the Write-All scheme. Further, the number of rounds required to validate votes is relatively independent of the number of malicious servers and the transaction generation rate. A detailed look at our raw data revealed that the vast majority of transactions that require at least one validated vote ended up validating all of the votes due to the exponential increase in the amount of information passed in each round of the epidemic update. Thus, once we incur the overhead of validating a single vote to guard against a single malicious server, the additional cost of guarding against any number of malicious servers is low. This explains why the commit delay does not significantly increase as the number of malicious insiders is increased from one to four. In fact, when we repeated this experiment with transaction rate equal to 0.8 and with the protocol guarding against the maximum number of malicious servers (15 in the 16 node case), we found the average commit delay to be about 16 rounds, within 1 round of the commit delay for guarding against a single malicious node.

In order to quantify the overhead of the internal security we computed the fraction of transactions that require at least one validated vote to commit. Intuitively, we expect this fraction to increase with the transaction generation rate and with the number of malicious nodes to guard against. In Figure 8 we plot the fraction of transactions in the system that at commit time had gathered at least one validated vote against increasing transaction rate. This metric is easier to measure and is a good upper bound on the number of transactions that *require* validated votes.

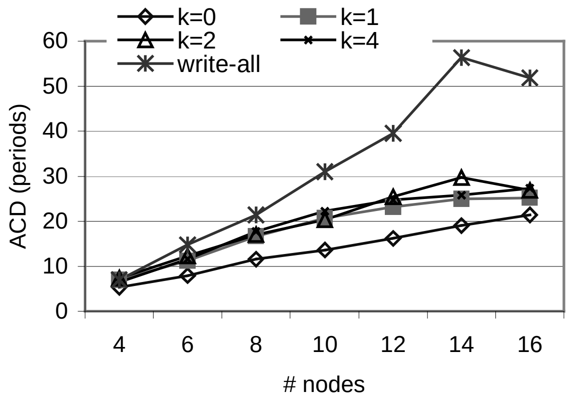

Figure 9: Scalability

This upper bound is loose, (within 10% in our test cases). Note that even under relatively heavy transaction rates, only about 50% of the committed transactions required validated votes.

5.1.3 Scalability

Figure 9 shows the change in commit delays as the number of nodes in the system is varied from 4 to 16. The transaction generation rate was fixed at 0.5. For this moderate transaction rate, adding the security increased the average commit delay by less than three rounds of communication. As is evident from the plot, the Write-all Scheme does not scale nearly as well. By contrast, the support for malicious servers does not hurt scalability at all. In fact, if there is any trend at all in the relationship between the runs with malicious servers and those without, it is that they are converging. Our future work will certainly include verifying this trend for more servers.

5.2 Improving on public-key signatures

The specific requirements for securing a vote and a vote receipt are as follows:

1. There must be enough information in each vote to authenticate the sender,
2. The authentication scheme must be non-repudiable, and
3. The messages must contain a strong message integrity check.

Public-key digital signatures provide all the primitives required to implement these three requirements. Individually signing a large number of votes and receipts using a conventional digital signature scheme such as RSA can be computationally prohibitive. Instead, any set of votes and receipts can be signed together as a single message.

In order to quantify the overhead of our proposed additions to Deno, we experimented with the execution times for signing a 16 byte hash code using 512 bit keys using the RSAref library from RSA Security Inc[4]. On a 440 MHz UltraSparc-IIi processor running Solaris 2.6, in single user, highest priority mode, producing the signature takes on average about 328 milliseconds (code compiled using the Sun WorkshopPro version 5.0 compiler). Thus, even if a large number of votes are signed in aggregate, public-key digital signature schemes can have a relatively high computational overhead, especially if regular RSA signatures are used on the typical small hand-held devices on which Deno may be used. In the next section, we show how to use a secure multicast technique described by Canetti et. al. in [18] to replace the digital signature authentication with *message authentication codes* (MACs).

5.2.1 Probabilistic Version

A MAC takes a secret key and a message (or a message hash) and generates an unforgeable value. The basic idea behind the probabilistic scheme is the following. Each sender is given a set of l "send keys". The sender authenticates a message by first creating a hash of the message using a secure hash algorithm, e.g., MD5. The sender then creates l keyed-MACs of the original hash by using its l send keys. (A keyed-MAC is computed by computing a hash of the message appended with a key). The server transmits the original message and all l keyed-MACs.

Each receiver s_u has a subset of the send keys: R_u. The receiver verifies the message by computing MACs of the hash for each key in R_u and comparing the results with the MACs that were appended to the message. A message can only be forged if the forger clique knows all of the keys in R_u; the probability of this occurring is one of the base parameters in the protocol and can be set to any value.

Given l keys and w malicious nodes, using the analysis given in [18], it can be shown that the value of q, the probability of failure is:

$$\left(1-\frac{1}{w+1}\left(1-\frac{1}{w+1}\right)^{w}\right)^{l}.$$

In our case, the Deno servers are both senders are receivers. All votes and receipts are messages that have to be signed. Consider the case when we are willing to tolerate one malicious server, and a probability of 1/1000 that messages can be forged. The number of primary keys, l, required is 25, and the resulting system would required 78 microseconds to generate all the MACs (once again on the 440 MHz UltraSparc machine using the same compiler). Thus, in this case, the MAC-based scheme is 4200 *times* as fast as a public-key signature system. We should note that in this case, each message would have to carry approximately 450 bytes of extra signature information not present in public-key signatures. The number of keys required to reduce the probability of forgery down to 1 in 10, 000 is 33. The results for withstanding more malicious servers is similar: for four malicious servers and the probability of forgery equal to 1 in 10, 1000, 108 MACs would need to be generated. This computation takes 340 microseconds (964 times faster than public-key signatures) and would add about 2000 bytes of extra key-information to each message.

6. Related work

The work related to this paper falls into two distinct categories: weakly-connected transactional systems and security for groups and elections. Many existing asynchronous "update-anywhere" protocols use the epidemic model [7, 9, 10, 17, 19, 20]. Many epidemic systems take an optimistic approach and use reconciliation-based protocols (e.g., Ficus [20], Lotus Notes [7]) that are only viable in non-transactional single-item domains.

Bayou [10] 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. As mentioned before, Agrawal *et al*. [17] proposed a pessimistic "Read-One, Write-All" (ROWA) [16] 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 [9] to improve availability and performance. Bayou protects against external threats using authentication and access control certificates.

The security protocols described here are related to traditional security in group communication protocols. Prior work in securing group communication protocols can also be divided into schemes that can withstand external threats only and schemes that are

[4] See http://www.rsa.com

able to withstand some internal threats. Ensemble[21] is the third-generation of group communication protocols from Cornell University and is a follow-up to the Horus [22] system. Ensemble is designed to be secure against external threats [23] and uses similar public-key mechanisms as Deno. Ensemble assumes internal threats do not exist, and defines protocols to handle security properties for multiple partitions of the group. Due to its reliance on an underlying atomic multicast protocol, Ensemble has to provide mechanisms for multicast group re-keying. Due to the epidemic nature of Deno's communication structure, we can completely ignore such key management [24] and rekeying [25] issues that are inherent to secure multicasting in wide-area networks.

The Rampart system [26] is also layered atop the atomic reliable multicast primitive. However, unlike Horus or Ensemble, Rampart is secure against external and internal threats. In [26], a protocol for reliable and atomic multicast is presented that is resilient of Byzantine attacks by at most one third of all the nodes the system. This protocol requires much stronger connectivity and reliable multicast primitives than is required by Deno. However, since Rampart is able to withstand all manner of Byzantine behavior, the liveness of the system is not affected by malicious nodes try to mount a denial-of-service attack by not responding to protocol requests. As mentioned before, the Deno countermeasure in this situation is to treat the non-responsive servers as failed than to revoke currency as explained in servers [14].

Our work is different from existing secure voting protocols in two major ways. By design, many existing secure voting protocols provide voter privacy and rely on a small number of central facilities for counting votes [27, 28]. In Deno, voter privacy is not an issue and the weakly connected nature of the underlying network makes reliance on central authorities untenable.

7. Conclusions

We have presented a complete infrastructure for protecting a highly-available, decentralized replication system against attacks, both external and internal. External attacks refer to those by entities that have not been authenticated into the system. This class of attack is prevented using traditional public-key techniques.

Internal attacks are much more interesting. An internal attack refers to one in which an entity has been authenticated into the system, and then tries to subvert the protocol by misrepresenting others' votes or lying about its own. The former case can also be handled by public-key signatures. The latter case, however, can only be addressed by modifying the transaction commit criteria to take the "trust" level of votes into account.

Our results that show that this modification of the commit criteria, while potentially expensive, still results in much more efficient transaction commits that the best competing approach from the literature. Our results also suggest that, far from hurting system scalability, the effect of the security modifications on system performance actually diminishes with increasing system size.

Finally, we show that a technique originally proposed for the secure multicast domain can be used to replace use of digital signatures with message hashes throughout our infrastructure. This replacement reduces the overhead of digital signatures in our system by at least a factor of 1000.

8. References

[1] P. J. Keleher, "Decentralized Replicated-Object Protocols," in *The 18th Annual Symposium on Principles of Distributed Computing (PODC '99)*, May 1999.

[2] P. Keleher and U. Cetintemel, "Consistency Management in Deno," *The Journal on Special Topics in Mobile Networking and Applications (MONET).*

[3] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *ACM Transactions on Computing Systems*, vol. 10, pp. 360-391, November 1992.

[4] Y. Breitbart and H. F. Korth, "Replication and Consistency: Being Lazy Helps Sometimes," in *Proc. of the Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997.

[5] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, Consistency, and Practicality: Are These Mutually Exclusive?," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998.

[6] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silbershatz, "Update Propagation Protocols for Replicated Databases," in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Philadelphia, PA, 1999.

[7] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and L. Greif, "Replicated Document Management in a Group Communication System," in *Proc. of the*

Conf. on Computer Supported Cooperative Work, 1988.

[8] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRESS," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 188-194, May 1979.

[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. of the Symposium on Principles of Distributed Computing*, 1987.

[10] 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. of the ACM Symposium on Operating Systems Principles*, 1995.

[11] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, vol. 4, pp. 180-209, 1979.

[12] D. K. Gifford, "Weighted Voting for Replicated Data," in *Proc. of the ACM Symposium on Operating Systems Principles*, 1979.

[13] S. Jajodia and D. Mutchler, "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database," *ACM Transactions on Database Systems*, vol. 15, pp. 230-280, 1990.

[14] U. Cetintemel and P. J. Keleher, "Light-Weight Currency Management Mechanisms in Deno," in *The 10th IEEE Workshop on Research Issues in Data Engineering (RIDE2000)*, February 2000.

[15] U. Cetintemel, P. Keleher, and M. Franklin, "Support for Speculative Update Propagation and Mobility in Deno," UMIACS UMIACS-TR-99-70, Oct. 29, 1999 1999.

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

[17] D. Agrawal, A. E. Abbadi, and R. Steinke, "Epidemic Algorithms in Replicated Databases," in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1997.

[18] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast Security: A Taxonomy and Efficient Constructions," in *Proceedings of INFOCOM '99*, 1999.

[19] M. Rabinovich, N. H. Gehani, and A. Kononov, "Scalable Update Propagation in Epidemic Replicated Databases," in *International Conference on Extending Database Technology (EDBT)*, 1996.

[20] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek, "Perspectives on Optimistically Replicated Peer-to-Peer Filing," *Software—Practice and Experience*, vol. 28, pp. 155-180, February 1998.

[21] M. Hayden, "The Ensemble System," Cornell Univerisity TR-98-1695, 1998.

[22] R. v. Renesse, K. Birman, and S. Maffeis, "Horus, a fexible Group Communication System," *Communications of the ACM*, 1996.

[23] O. Rodeh, K. P. Berman, M. Hayden, Z. Xiao, and D. Dolev, "Ensemble Security," Cornell Univerisity TR-98-1703, 1998.

[24] H. Harney and C. Muckenhirn, "Group Key Management Protocol (GKMP) Architecture.," RFC 2094, 1997.

[25] C. Wong, M. Gouda, and S. Lam, "Secure Group Communications using Key Graphs," in *ACM SIGCOMM*, 1998.

[26] M. K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," in *2nd ACM Conference on Computer and Communications Security*, 1994.

[27] A. Fujioka, T. Okamoto, and K. Ohta, "A practical secret voting scheme for large-scale elections," in *Advances in Cryptology --- AUSCRYPT'92, Lecture Notes in Computer Science*, 1992.

[28] L. Cranor and R. Cryton, "Sensus: A security-conscious electronic polling scheme for the Internet," in *Hawaii International Conference on System Sciences*, 1997.