A Security Infrastructure for Mobile Transactional Systems

Uğur Çetintemel, Peter J. Keleher, and Bobby Bhattacharjee

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
{*ugur, keleher, bobby*}*@cs.umd.edu*

In this paper, we present an infrastructure for providing secure transactional replication support for peer-to-peer, decentralized databases. We first describe how to effectively provide protection against external threats, malicious actions by servers not authorized to access data, using conventional cryptography-based mechanisms. We then classify and present algorithms that provide protection against internal *threats, malicious actions by authenticated servers that misrepresent protocol-specific information. Our approach to handling internal threats uses both cryptographic techniques and modifications to the update commit criteria. The techniques we propose are unique in that they not only enable a tradeoff between performance and the degree of tolerance to malicious servers, but also allow for individual servers to support non-uniform degrees of tolerance without adversely affecting the performance of the rest of the system.*

We investigate the cost of our security mechanisms in the context of Deno: a prototype object replication system designed for use in mobile and weakly-connected environments. Experimental results reveal that protecting against internal threats comes at a cost, but the marginal cost for protecting against larger cliques of malicious insiders is generally low. Furthermore, comparison with a decentralized Read-One Write-All protocol shows that our approach performs significantly better under various workloads.

1. Introduction

Asynchronous approaches (e.g., [5, 18, 22]) for managing replicated data have gained popularity due to their inherent advantages over traditional synchronous solutions in mobile, large-scale, and wide-area environments. Asynchronous approaches that provide the most flexibility are peer-to-peer, decentralized approaches, which can operate under less than full connectivity, easily adapt to frequent changes in group membership, and make few demands on the underlying network topology. These decentralized approaches, also called lazy-group [15], can support the update-anytime-anywhere-anyhow model, which eliminates restrictions regarding where and how updates are performed, effectively facilitating dis- and weakly-connected operation.

Although these approaches have long been used to used to support optimistic, weakly-consistent replication [18, 21, 27, 33], it is only recently that several proposals have extended them to support serialized single-item updates [20], and transactional updates and serializability [2, 9, 16]. With the advent of wireless ad-hoc networking and technologies like Bluetooth, peer-to-peer, decentralized approaches are likely to become increasingly more prevalent in future replicated databases and systems.

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

The infrastructure we present addresses both external and internal security threats. The prime *external* threat is of an unauthenticated server attempting to read or modify data. We prevent this through a combination of cryptography-based mechanisms. Our main focus in this paper is, however, dealing with *internal* threats to security, which is more problematic. Internals threats arise from duly authenticated servers (i.e., insiders) that attempt to cheat by misrepresenting protocol-specific information. As a trivial example, a user of a distributed meeting room scheduler might attempt to falsify votes of other servers in order to ensure that he or she gets a prime reservation. More serious scenarios could arise in collaborative intranet and Internet applications, such as scheduling and workflow applications. Finally, this work has obvious applications in military scenarios (e.g., consider communication among tanks or mobile command posts). We deal with malicious insiders by using cryptographic techniques, as well as modifications to the update commit criteria.

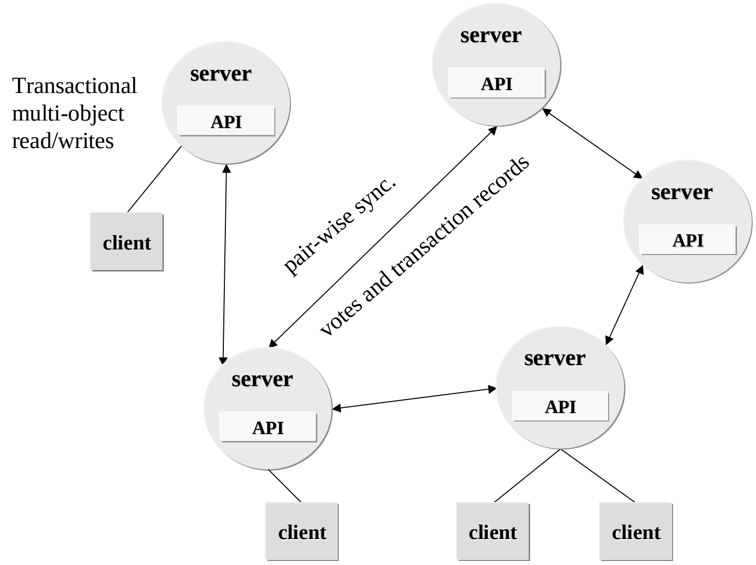

Figure 1: Basic Deno system model

The fundamental idea is to ensure that any protocol-specific information used is correct by using cryptographic techniques when possible, or by explicitly *validating* any such piece of information.

This work is done in the context of Deno [9, 19], a decentralized system that supports transactional replication for mobile and weakly-connected environments. Deno's system model is illustrated in Figure 1. One or more clients connect to each *peer* server, and submit transactions. Servers communicate through *pair-wise* information exchanges. Servers make all commit decisions independently and by using solely local information. The base, non-secure Deno system has been fully implemented. We also implemented the security extensions to the base Deno protocol needed to tolerate malicious insiders. However, we are still building the public-key infrastructure that will be used to address external threats.

Despite the growing need for security infrastructures for managing replicated data, this topic has yet to be well addressed in the decentralized, asynchronous environments that we target. Prior work mainly investigated security issues in traditional synchronous environments (e.g., [25]), and environments where strong connectivity and atomic reliable multicast primitives are available and replication can be globally coordinated by distinguished master servers (e.g., [7, 10, 12, 30, 31]). Studies that address the restrictions of our target environments [24], on the other hand, ignored consistency issues and made assumptions about where and how updates are generated and initially received. Our work aims to provide flexible security mechanisms that eliminate many restrictions of prior work while allowing for strongly-consistent access to decentralized, replicated data. We note that, although our fault model is comprehensive, it does not handle *fully* Byzantine attacks due to its reliance on digital signatures for authenticating the original source of messages that are forwarded in the system.

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

The rest of the paper is structured as follows. Section 2 briefly describes Deno's asynchronous, decentralized replication 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 connectivity. Section 4 describes our flexible approach to handling internal threats, which is the main contribution of this paper, Section 5 describes the Deno architecture, and Section 6 evaluates the effect of our security measures on commit performance using a prototype system. Finally, Section 7 describes related work and Section 8 concludes.

2. Background: Deno

Deno is an object replication system that relies on a decentralized, asynchronous replica management protocol to address 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 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 [14] and epidemic information flow [11], a process where information flows pair-wise 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 [33]); 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., [14, 17, 34]), 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 [3]. The use of weighted voting allows implementations to improve performance by adapting currency distributions to site availabilities, update activity, or other relevant characteristics [8]. 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 servers are routinely disconnected. Second, epidemic protocols remove reliance on network topology: synchronization partners in epidemic protocols can be chosen randomly, eliminating any potential single point of failure.

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 [9].

2.1 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. We assume a model in which the shared state consists of a set of objects that are replicated across multiple servers. Clients submit reads and updates to any Deno server. Synchronization sessions between pairs of servers move newly created update records, together with other protocol information such as votes for such, among the servers.

Updates do not commit globally in one atomic phase because we assume an epidemic style of update propagation and poor connectivity. Asynchronous voting is used to determine which updates actually commit, and in which order. Each server commits updates *independently* using only *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. In the rest of the section, we briefly discuss our replication protocol (full details, which we omit here for brevity, can be found in [9]).

2.2 Peer-to-peer, decentralized replication: The base Deno protocol

2.2.1 Transaction model

A transaction consists of a sequence of read and write operations. 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 commits 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. Two transactions are said to *conflict* if one of the transactions updates an item accessed by the other transaction.

Database states are tracked by associating a version number with each database item. The items in the local copy of the database are modified, and their version numbers incremented, only when update transactions commit. We assume that 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. This model, also employed by Bayou [33], is especially useful in facilitating disconnected operation.

2.2.2 Voting

In Deno, a vote (record) v, contains four pieces of information: (1) *voter*(v), which denotes the server that created v; (2) *trans*(v), which denotes the transaction v is cast for; (3) *curr*(v), which denotes the currency (i.e., weight) held by *voter*(v) when v is created; and (4) *tstamp*(v), which is the local timestamp value of *voter*(v) when it created v. A server, s_a, votes once (and only once) for each transaction that it observes. We denote the vote cast by s_a for transaction t_j as $v_{a,j}$. The primary purpose of this voting process is to provide a total order on the set of transactions observed by a server, i.e., the set of votes that belong to each server can be totally ordered based on vote timestamps. We say that "a transaction t_i precedes transaction t_j at s_a" iff $tstamp(v_{a,i}) < tstamp(v_{a,j})$, and denote this relation by $t_i <_a t_j$. This relation reflects the order in which a particular server desires to commit (i.e., serialize) the transactions it observes. For instance, if $t_i <_a t_j$, then s_a desires to commit t_i before t_j. This ordering information is propagated across the system via pair-wise synchronization sessions as a part of vote records.

2.2.3 Update commitment using local information

We now describe the update commitment process from the perspective of a single server. Each Deno server, s_a, maintains a set of votes, V_a, and a set of transactions, T_a. The latter set, T_a, consists of those update transactions that are known to s_a, have finished execution either locally or remotely, but have yet to be either committed or aborted at s_a.

The candidate transaction of a server s_a, *candidate*(s_a), is the transaction which s_a desires to commit before all others. More formally, if *candidate*(s_a) is t_i, then $t_i <_a t_j$, $\forall\, t_i, t_j \in T_i$, $i \neq j$ (i.e., *candidate*(s_a) is the transaction that comes first in the ordering of s_a). The candidate transaction set at s_a, C_a, contains the candidate transactions of all servers; i.e., $C_a = \{candidate(s_b)\}$, for all servers s_b.

As explained below a candidate transaction *t commits* at s_a when s_a guarantees, using locally available information, that *no other candidate 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, and the protocol guarantees that all servers eventually reach the same commit decisions. Given a server s_a, its vote set V_a, and its candidate transaction set C_a, we define the following:

- *Sum of votes of a candidate transaction* t_i: $\left|votes(\{t_i\})\right| = \sum curr(v_{b,i})$,

s.t., $v_{b,i} \in V_i$, and $candidate(s_b) = t_i$.

- *Unknown votes*: *unknown* = $1.0 - \sum \left|votes(C_a)\right|$.

In other words, *unknown* is the sum of currencies of those servers whose candidate transactions are not yet available. We now define the commit criterion that s_a uses to decide which transactions to terminate (i.e., commit or abort) on the basis of local information. The fundamental idea is to commit a candidate transaction when it is guaranteed that no other candidate transaction can gather more votes.

- *Commit criterion*: A candidate transaction t_i commits iff:
  1. $|votes(\{t_i\})| > 0.5$, or (*Majority case*)
  2. $|votes(\{t_i\})| > |votes(\{t_j\})| + unknown, \ \forall t_j \in C_a$, and $i \neq j$. (*Plurality case*)

The commit criterion states that candidate transaction t_i can commit if it gathers the *plurality* of votes. Case 1 indicates that t_i gathered majority of the votes. Case 2 ensures that no other candidate transaction, which may or may not be known to server s_a, can gather more votes (Note that ties that occur when two candidate transactions gather the same amount of votes can be broken using a simple deterministic comparison between the indices of the servers that created the transactions).

When a candidate transaction *t* commits at server s_a, s_a first incorporates the effects of t_i 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. All candidate transactions whose read items are modified become *obsolete,* and are aborted.

2.2.4 Synchronization

We now discuss how two Deno servers synchronize their states (also called *anti-entropy* [11] in the terminology of epidemic algorithms). A pair-wise synchronization session essentially involves the propagation of (1) transaction records, and (2) votes, that are known to one server and unknown to the other.

In Deno, synchronization is controlled via version vectors [26]. In our model, each server s_a maintains an *n*-element version vector, vv_a, where *n* is the number of servers, that describes the number of events of each other server *seen* by s_a. Element $vv_a[b]$ is a scalar count of the number of s_b's events that have been seen at s_a. In our case, there are two types of events of interest: transaction pre-commits, and vote creations. A *pre-commit* event occurs whenever an update transaction completes its local execution on the server where it executed and is ready to participate in the global voting process. A *vote creation* event occurs whenever a server casts a vote.

In more detail, each server s_a maintains a serial order, called *local order,* on all pre-commits and vote creations. We denote event *l* of s_a as e_a^l. As information about events is always propagated in local order, if s_a's version vector is vv_a, then s_a has seen all events $e_b^1 \dots e_b^{vv_a[b]}$, for all s_b, $b = 1 \dots n$. We assume a unidirectional, two-message *pull* synchronization, although other modes are possible [11, 20]. When s_a pulls information from s_b, the following actions take place:

1. Server s_a sends vv_a to s_b.
2. Server s_b responds with all events e_k^l s.t. $l > vv_a[k]$ and $l \le vv_b[k]$, for all $k = 1 \dots n$.
3. Server s_a incorporates the new events in the same order that they originally occurred by first applying the voting rule and the commit rule for all relevant transactions, and updating vv_a to the pairwise maximum of vv_a and vv_b

For purposes of exposition, we assumed *n*-dimensional vectors in the above description. Our implementation uses a set representation for the version vector; i.e., $vv_a = \{(b, cnt_b), (c, cnt_c) \dots\}$, where each pair consists of a server id, *b*, and a count, cnt_b, specifying the number of s_b's events seen by s_a. Using this synchronization protocol, the local orders at each server are propagated to the rest of the system.

2.2.5 Consistency and correctness issues

In this section, we briefly discuss the consistency and correctness related issues in the base Deno protocol. Full details and the correctness proofs, which are omitted for brevity here, can be found in [9].

The base Deno protocol described earlier forces all update transactions to commit in the same order at all servers, thereby providing strong consistency and serializability, where each query serializes with respect to both queries and update transactions. Strong consistency is characterized by an acyclic serialization graph, prohibiting both *update transaction cycles* (i.e., cycles involving only update transactions) and *multi-query cycles* (i.e., cycles involving multiple queries and one or more update transactions) [3, 4, 13], thereby guaranteeing globally-serializable executions.

We also developed a *weak-consistency* version of our protocol [9] that commits an update when it is guaranteed that no other *conflicting* transaction can gather more votes. We showed that relaxing the level of consistency among replicas in this manner yields performance improvements in many cases [9]. This weak-consistency protocol supports a form of update consistency [3, 4, 13] where each query serializes with respect to all update transactions, but possibly not with other queries. However, this protocol does ensure that queries always observe transactionally-consistent database states. We do not consider the weak-consistency protocol further in this paper, and use the base strong-consistency protocol since; (1) the level of consistency supported is orthogonal to the main theme of this paper; and (2) we want to establish a fair comparison with another decentralized protocol that provides strong-consistency (see Section 6).

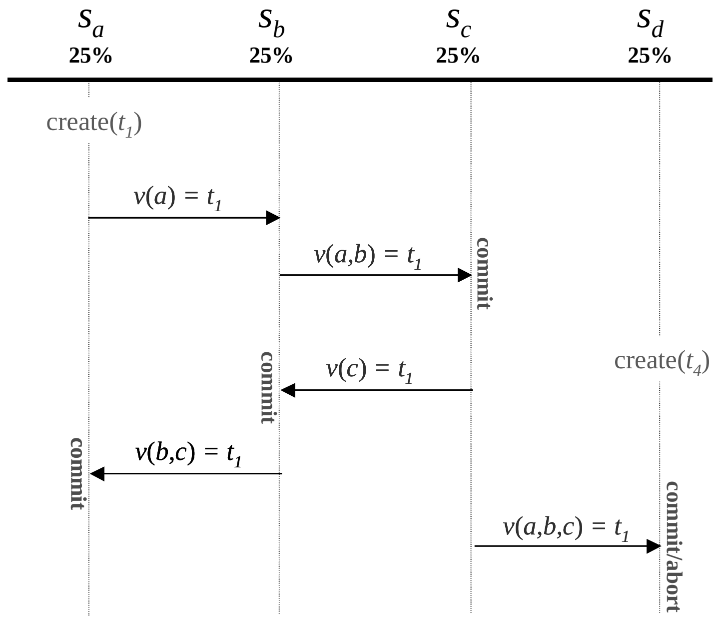

Figure 2: Illustrating update commitment

2.3 Protocol illustration

We illustrate our protocol in Figure 2 with an example scenario. The system has four servers, all with currency of 0.25. Server s_a creates a new update, t_1, votes for it, and sends a message describing t_1 and its vote to s_b via a synchronization session. Server s_b votes for t_1, and then later transfers notice of t_1 and both votes to s_c. After adding its own vote, s_c can commit t_1 because it has gathered a plurality. Later synchronization sessions move the votes back to s_b and s_a, which also reach the same commit decision.

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

3. External security threats

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

3.1 Authentication

A principal (server) is authenticated into the system by identifying itself to a distinguished server acting as the certificate authority (CA). We assume a priori that all servers trust the CA, and know the CA's public key. The CA responds with an *access certificate* that specifies the principal's rights in the system. Certificates may provide either *read* or *read/write* permission for a given database, and may contain a timestamp that delimits the certificate's lifetime. Since a certificate is signed by the CA, any server with

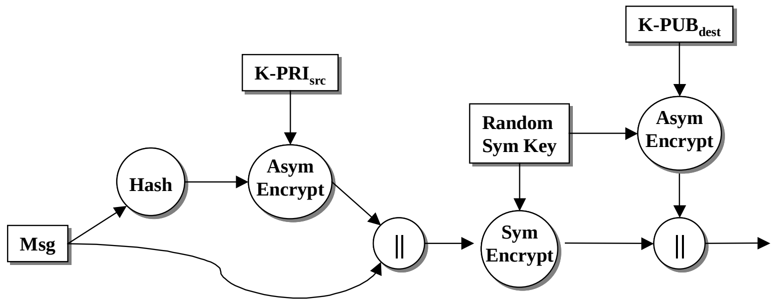

Figure 3: Integrity with privacy: We use symmetric encryption (i.e., Triple-DES) to encrypt the message, and asymmetric encryption (i.e., RSA) to encrypt the symmetric key and sign the message. K-PRI$_{src}$ and K-PUB$_{dest}$ are private and public keys of the source and destination of the message, respectively (double bars indicate concatenation).

the CA's public key can verify that the certificate is valid, and certificates can not be forged. Note that we assume a priori that all servers trust the CA, and know the CA's public key.

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

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

We solve this problem by allowing the CA to issue *ticket-granting tickets* (TGT), analogously to Kerberos [32]. A TGT gives the bearer a limited ability to make and grant new certificates for resources and properties. In our architecture, use of a TGT requires direct confirmation from the user. Note the TGT's can be used to generalize the system to include a hierarchy of CA's. This not only provides load-balancing for access to the CA's, but increases the chances that a CA is available when needed.

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

3.2 Integrity and privacy

Figure 3 shows Deno's approach to providing both integrity and privacy guarantees for communicated data. Note that this method is very similar to the method used in PGP. Integrity is provided by appending a message authentication code (MAC) to each message, which in this case is the MD5 hash of the message signed by encrypting with the source's private key. Privacy is provided by encrypting the message and the MAC with a randomly generated, one-time *session key*. The session key is then encrypted with the destination's public key, and the concatenation of the encrypted session key, MAC, and message is sent to the destination.

The use of peer-to-peer one-time session keys allows us to avoid the *key changing* problem incurred by secure multicast trees [31] (note that these peer-to-peer keys need not be one time; instead they may be cached and re-used later). Secure multicast trees generally use a single session key for the entire group. Any change in group membership requires the session key to be changed. The key must be changed when a new server, s_n, is added to the group because we do not want s_n to be able to read messages that were sent prior to its joining (we assume that s_n might have recorded prior encrypted messages even though it could not read them). Similarly, the key should be changed when s_n leaves the group because we do not want s_n to be able to read messages that are sent after it leaves the group. The use of peer-to-peer session keys, thus, eliminates a similar need in our model.

4. Internal security threats

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

4.1 Malicious actions

Before we classify the actions a malicious intruder can take, we should note that malicious servers can always commit arbitrary transactions to their *local* databases without even advertising the transaction to other servers. Malicious servers can also remain within the protocol framework and issue updates that, if committed, obscure or undo the effects of other updates. This type of behavior can only be handled in an application-specific manner and is beyond the scope of this work. Under certain circumstances, even a single malicious server can accomplish a denial-of-service attack by refusing to vote its currency. This attack is handled by the Deno's normal *currency revocation* mechanism [8] used to recover from benignly-failed servers.

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

4.1.1 Currency misrepresentation

The problem here is of a server misrepresenting the amount of currency it has available for voting purposes. This is possible since Deno servers can perform *peer-to-peer currency exchanges* to migrate currency allocations towards a target distribution. A peer-to-peer exchange is used by two servers to reallocate their currency between them. Although this local operation enables light-weight replica creation, retirement, and dynamic currency redistribution [8], and it poses a unique security problem in that it cannot be directly verified by other servers.

We make this operation secure by requiring each currency exchange to be formalized as an update. A currency transfer from s_i to s_j is only considered complete when the corresponding *exchange update* is committed. Note that such exchange updates are commutative with respect to all other updates and are generally committed faster than ordinary updates.

4.1.2 Vote misrepresentation

There are two types of vote misrepresentation:

1. *Misrepresenting non-local votes*: A malicious server s_m misrepresents or forges some other server s_a's vote to a third server s_b. This can happen, for instance, when s_a and s_b are connected through s_m, s_a reports its vote to s_m and s_m forges this vote and reports a different vote for s_a to s_b. This type of malicious behavior is prevented by requiring each server to sign its votes using a suitable digital signature technique. The worst a malicious server can do then is to never report s_a's vote to s_b. Since our symmetric, peer model does not impose any specific connectivity requirements, this behavior can only delay committing of transactions, but cannot affect correctness.
2. *Misrepresenting local votes*: The second vote misrepresentation is more difficult to guard against and can quite easily be used to violate all correctness guarantees. In this case, a server (possibly signs) and illegally votes its own currency more than once for multiple transactions. Consider the example shown in Figure 4. Assume that server s_m is malicious. If s_m tells s_a that it votes for x, and s_b that it votes for y, then both destinations reach the conclusion that their candidates have more than 50% of the vote and can be committed. Furthermore, securely signed votes do not help in this case since s_m can properly sign its own vote for any transaction. In the rest of this section, we investigate approaches to detecting such malicious servers, and develop an algorithm that guarantees correctness at all non-malicious servers.

4.2 Approaches for handling internal threats

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

4.2.1 Secure update commitment

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

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

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

In general, *votes*($\{t_i\}$) consists of validated votes, *valid*($\{t_i\}$), and unvalidated votes, *unvalid*($\{t_i\}$). Note that we consider votes cast by the local server to be validated votes. We denote the currency of any vote v in *votes*($\{t_i\}$) by $|v|$. Similarly, we denote the total currency for a set V of votes by $|V|$, e.g., $|votes(\{t_i\})|$ denotes the sum of the currencies of all votes cast for $t_i \in T$. Finally, let *unvalid*(δ, T) be the set of δ elements with the largest currency in *unvalid*(T). If we consider all votes in the base Deno system to be validated, then the base commit criterion for t_i can be stated as in the top row of Table 1, where *unknown* is defined as 1 - $|votes(\{C\})|$ and C is the set of candidate transactions.

In order to provide resilience against malicious servers, the non-secure commit criterion is modified as in the second row of Table 1. The left hand side of the inequality provides a

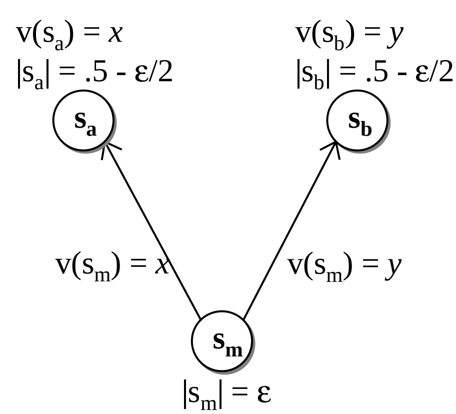

Figure 4: Vote misrepresentation: By telling s_a and s_b different votes, s_m can cause them to commit conflicting updates ($|s|$ is the currency held by s and $v(s)$ is the transaction s votes for).

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

Table 1: Commit criteria

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

In order to validate a vote for transaction t_i from a server s_b, a server s_a must ensure that all other servers in the system have seen the same vote. Thus, server s_a must collect *receipts* of the votes cast by s_b to all other servers. A *receipt* of server s_b's vote from server s_c is a statement of the form "Server s_b votes for transaction t_i", securely signed by server s_c using an appropriate digital signature. Server s_a considers a particular vote valid if and only if it has received receipts for that vote from all other servers in the system or if the vote is cast by server s_a itself. In order to validate a vote, a server s_a does not need to establish a peer-to-peer connection with all other servers in the system — instead, receipts for votes from any server can be forwarded by any other server in the system. Since strong cryptographic primitives protect the receipts, even malicious servers will not be able to alter the contents of the receipt. Malicious servers may corrupt or discard receipts: corrupt receipts will be detected by the server validating the receipt, while discarded receipts will be treated as any lost message. In the worst case, malicious servers may be able to affect the liveness properties of the algorithm, but once again, we have been able to restore the safety guarantees[1].

When a server detects a malicious vote while performing validation, it marks the corresponding server as malicious, ignores all further votes from that server, and initiates the currency revocation mechanism [8] to cancel the voting rights of the malicious server. If the server already committed an update incorrectly using a malicious vote — which can happen only if the degree of tolerance set by the server is less than the actual number of malicious insiders — the server has to rollback the effects of the update.

4.2.2 Correctness of the secure commit criterion

We now provide a proof sketch for the correctness of the secure commit criterion. The correctness proof for the non-secure commit criterion (i.e., δ=0) can be found in [9].

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

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

4.2.3 Examples and discussion

In this section, we illustrate, via a set of examples, some of the more subtle properties of the secure plurality algorithm. We begin with a simple example of applying the secure protocol to the three server case shown in Figure 4. We had shown earlier that if server s_m is malicious, in the base protocol, under appropriate circumstances, it could cause the committed views of servers s_a and s_b to diverge arbitrarily far *even if it held arbitrarily small amount of currency in the system.* Now we show that even if server s_m is malicious and holds arbitrarily *large* amounts of currency in the system, it cannot cause a *single* incorrect commit at either servers s_a or s_b, as long as servers s_a and s_b operate under the assumption that there are malicious servers in the system. Assume s_m holds an arbitrary amount (say c_m) of currency. Once again, assume the rest of the currency is distributed equally between servers s_a and s_b (the analysis for the other cases are analogous and is omitted for brevity).

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

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

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

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

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

5. Deno architecture

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. The basic Deno API [19] supports operations for creating objects, creating and deleting object replicas, and performing reads and writes on the shared objects in a transactional framework.

Figure 5 illustrates the basic Deno server architecture. The *Server Manager* is in charge of coordinating the activities of the various components. It also 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 interest from other servers. This component implements synchronization policies that specify when and with which server to synchronize. The *Trans Manager* is responsible for the local execution of transactions. It maintains a transaction pool that contains all *active* transactions known to the server. The *Storage Manager* provides access to the object store that stores the current committed versions of all replicated objects at the server. The object store is currently implemented as an 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 layered atop UDP/IP. Deno consists of ~13,000 lines of multi-threaded C++ code, and has a footprint of ~170KB.

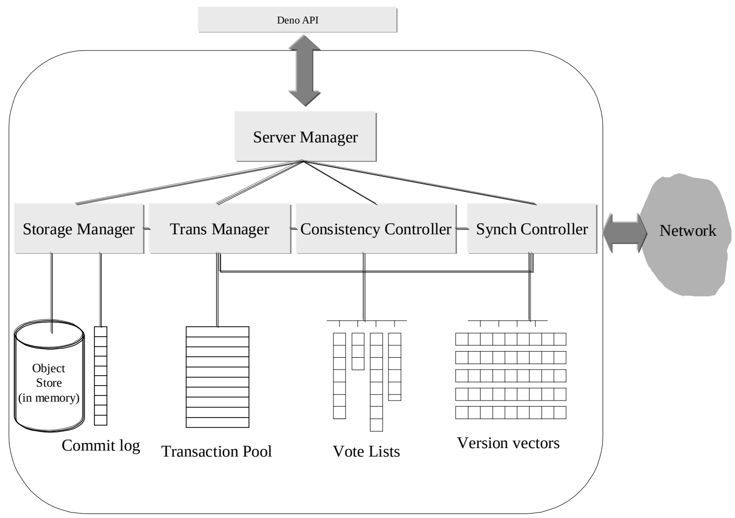

Figure 5: Deno prototype

6. Performance evaluation

6.1 Experimental environment and performance metrics

Using our prototype system, we now evaluate the cost of the proposed security mechanisms for Deno's decentralized voting protocol and a ROWA-type decentralized protocol (described below). We performed the experiments on a 16 node Linux cluster with each node running a copy of the Deno server. Each node contains two 400 MHz Pentium II processors and 256 MBytes of RAM. We note that 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 the investigated protocols.

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 by sending a *pull* request to another randomly selected server. Each server generated transactions according to a global transaction rate (specified relative to a synchronization period). Each transaction accessed and modified up to five data items. In all experiments, currency is uniformly distributed across servers, and 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 decentralized scheme, `write-all`, which is an epidemic "Read-One, Write-All" (ROWA) [3] protocol modeling the other peer-to-peer decentralized transactional protocol in the literature. This protocol commits transactions after ensuring that all other servers are ready to commit. Therefore, a transaction has to be propagated to all the servers before it can be committed. In terms of the voting terminology, `write-all` commits a transaction when the trans-

Parameter	Description		Setting
Synch Period (*SP*)	Mean synchronization period	(uniform)	0 – 5 (secs)
Transaction Rate (*TR*)	Mean transaction generation rate	(uniform)	0 – 5 (trans/synch period)
Num Servers (*n*)	Number of Deno servers		3 – 30
Trans Size	Number of items updated by a transaction	(uniform)	0 – 5
Degree of tolerance (δ)	Number of malicious servers that can be tolerated		0 – n-1

Table 2: Primary experimental parameters and settings

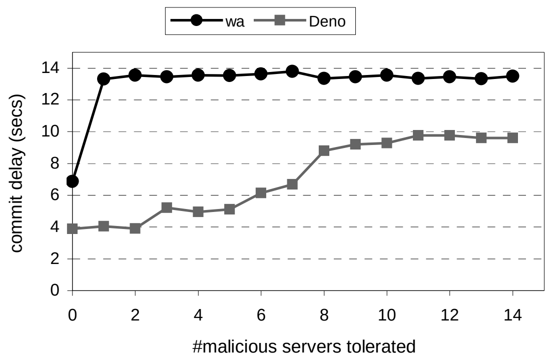

Figure 6: Commit delays vs. degree of tolerance, *n*=15, *TR*=0.01, *SP*=2.0

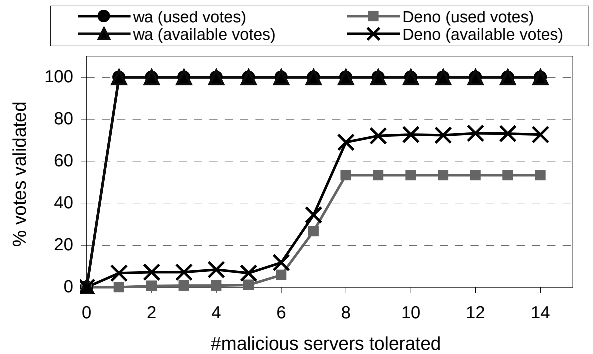

Figure 7: Percentage of validations required/available for commit vs. degree of tolerance, *n*=15, *TR*=0.01, *SP*=2.0

action gathers all the votes in the system. A similar ROWA-type epidemic protocol was proposed by Agrawal *et al.* [2]. We also implemented a secure version of `write-all`, which also uses a straightforward adaptation of the vote validation technique described in Section 4.2.

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

Before presenting our results, we would like to note that individually signing a large number of votes and receipts using a conventional digital signature scheme such as RSA can be computationally expensive: the execution times for signing a 16 byte hash code using 512 bit keys using the RSAref library from RSA Security Inc. (see www.rsa.com) is approximately 330 msecs in our environment. Instead, any set of votes and receipts can be signed together as a single message. Furthermore, probabilistic techniques that trade off signature quality to computational overhead can also be used to decrease this computational cost by several orders of magnitude (e.g., [6]).

6.2 Commit delays vs. degree of tolerance to malicious servers

Figure 6 shows the average commit delays for very small transaction generation rates (i.e., no update contention), for Deno and `write-all`, with varying degrees of tolerance to malicious servers, i.e., δ. On the x-axis we vary δ from 0 (non-secure system) to n-1 (max-security system). The curve for Deno follows an *S*-shape; initially increasing gradually with increasing δ, making a significant jump in the vicinity of $\delta = n/2$, and then essentially staying flat afterwards. As long as δ is smaller than $n/2$, servers do not need to use validated votes to commit an update; it simply is enough to gather sufficient unvalidated votes. For instance, assuming no update contention and n set to 15, a single update can be committed with 11, 12, 13, 14 unvalidated votes when δ is 3, 4, and 5, respectively. However, when δ is more than half the servers, it is not possible to commit updates without the use of validated votes. Vote validation is a relatively costly operation, as it involves obtaining receipts from the other servers in the system. This explains the sudden increase in commit delays as δ exceeds $n/2$. After this point, commit delays continue to increase as more validated votes are required for commit. At the point where half of the all votes are validated, updates can immediately commit, which is the reason why commit delays for Deno essentially stays constant for relatively large δ values.

The figure also depicts commit delays for non-secure `write-all` ($\delta = 0$), and secure write-all ($\delta >$ 0). Since secure `write-all` requires all votes to be validated for commit, it cannot support intermediate degrees of tolerance as Deno. We observe that for all degrees of tolerance, Deno commits updates signifi-

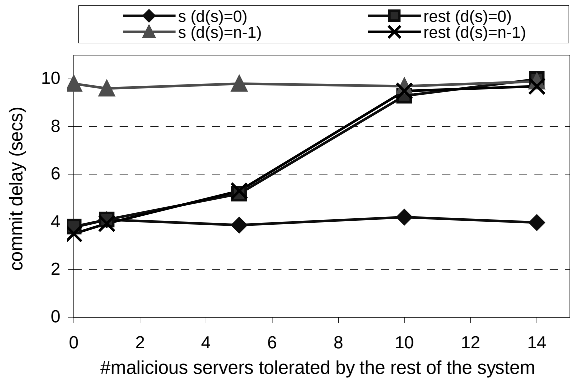

Figure 9: Supporting non-uniform degrees of tolerance at individual servers, *n*=15, *TR*=1.0, *SP*=2.0

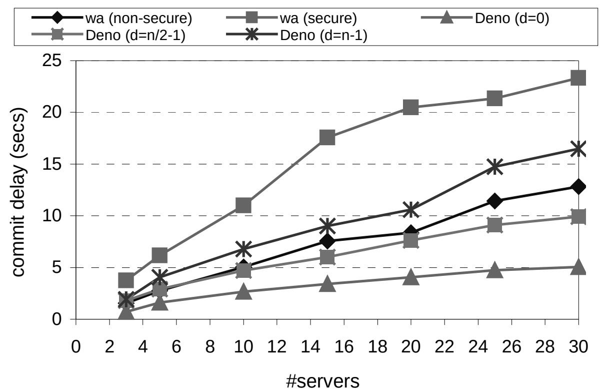

Figure 8: Scalability, *TR*=0.01, *SP*=2.0

cantly faster than `write-all`, reducing the commit delays of `write-all` by 40% and 30% for non-secure (δ = 0) and maximum security cases (δ = n-1), respectively. The most dramatic improvement, 60%, occurs when 0 < δ < n/2, since, in this region, Deno commits updates *without* validating any votes, whereas `write-all` has to validate *all* votes before committing an update.

Figure 7 provides more insight by plotting the percentage of validated votes used and those that are available at commit time at each server, averaged over all commits across all servers. As we expected, no validated votes are used at commit by Deno when δ < n/2. In this region, validated votes available at commit time at each server are non-zero, because each server considers its own vote as validated by default. Notice that Deno requires at most 50% of the votes to be validated for supporting any degree of tolerance. On the other hand, `write-all` requires 100% votes to be validated to tolerate any number of malicious servers, thereby incurring relatively large commit delays.

6.3 Performance implications of supporting non-uniform degrees of tolerance

We now investigate the performance impact of using different degrees of tolerance at different servers. We expect that the commit performance of each server be independent from the degrees of tolerance supported by others, since each Deno server makes all commit decisions entirely *independently* and using only *local* information. To demonstrate the validity of this premise, we conducted an experiment where we let a single server, *s*, use a degree of tolerance, δ(*s*), different from that used by the rest of the servers, δ(*rest*).

Figure 9 presents commit delay results for *s* and the rest of servers (averaged) for the cases where δ(*s*) = 0 and δ(*s*) = n-1, as we vary δ(*rest*) — note that *d* refers to δ in the figures. When we consider the commit delay curve for *s* when *k*(*s*)=0, we observe that the curve remains essentially flat regardless of the degree of tolerance used by the other servers. Even when δ(*rest*) is set to n-1, the performance of *s* is not affected at all. The same observation holds for the case where δ(*s*) = n-1. It is evident that the commit performance of a server is not affected by the performance of the rest of the system. The commit delay curves for the rest of the system for δ(*s*) = 0 and δ(*s*) = n-1 illustrate the complementary case. We observe that the two curves are essentially identical, revealing that the performance of the system as a whole is not affected by the degree of tolerance set by *k*. It is therefore clear that the degree of tolerance used by a server does not adversely affect the performance of other servers and vice versa.

6.4 Scalability

Figure 8 shows commit delays for Deno with various degrees of tolerance for malicious servers, and for secure and non-secure versions of `write-all`, as the number of servers is varied from 3 to 30. As expected, the commit delays increase as the system size increases for both Deno and `write-all`. Non-secure Deno (i.e., δ = 0) demonstrates the best scalability, followed by Deno with δ = n/2-1. Recall that at this tolerance degree, commits do not require any validated votes, eliminating the need to contact every other server in the system. This also explains why Deno with $\delta = n/2-1$ performs better than even the non-secure `write-all`, which needs to contact all servers to commit an update. We also observe that Deno with max-security (i.e., $\delta = n-1$) commits updates faster than secure `write-all`(since Deno needs to validate at most half the votes), and the difference between the two approaches increases with increasing system size.

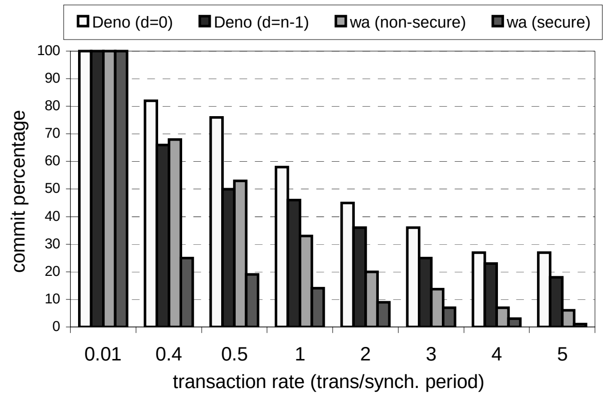

Figure 10: Update contention effects, *n*=15, *SP*=2.0

6.5 Update contention effects

We now investigate the effects of update contention on Deno and `write-all`. Figure 10 plots commit percentage results for varying transaction generation rates. The figure shows that all approaches suffer from the increased transaction rate due to the global update consistency requirement that at most one out of a set of conflicting transactions can commit (the transactions that are aborted can be restarted depending on application semantics). Under very small transaction rates, $TR \in [0.0\text{-}0.01]$, all protocols perform fairly well, committing all updates. With increasing transaction rates, however, commit percentages drop for all protocols significantly. We observe the most dramatic fall for secure `write-all`: at a transaction rate of 0.4, the commit percentage of secure `write-all` is ~25%, whereas the commit percentages of the other protocols are all above 65%. Notice that beyond a transaction rate of 0.5, *max-security* Deno has a higher commit percentage than even the *non-secure* `write-all`. The reason for this interesting result lies in the fundamental difference in the way Deno and `write-all` treat conflicting updates. Deno's voting algorithm can globally pick a single update out of a set of conflicting updates to commit. The `write-all` protocol clearly lacks such a mechanism and thus has to abort all conflicting updates. Due to this behavior, beyond some update contention, the `write-all` approaches, both secure and non-secure, are not able to commit any updates (beyond 6 and 10 transactions/synch period, respectively). On the other hand, Deno approaches continue to make progress and commit updates regardless of the update generation rate (not shown).

7. Related work

The work related to this paper falls into two distinct categories: weakly-connected (transactional) systems and security for groups and elections. Most existing asynchronous *update-anywhere* protocols use the epidemic model (e.g., [2, 11, 18, 27, 28, 33]). Many epidemic systems take an optimistic approach and use reconciliation-based protocols (e.g., Ficus [27], Lotus Notes [18]) that are only viable in non-transactional domains. Agrawal *et al*. [2] proposed a decentralized *Read-One, Write-All* [3] approach that was the first decentralized, epidemic protocol to ensure strong consistency and serializability. More recently, several work [9, 16, 20] investigated the integration of decentralized voting protocols with the epidemic communication mainly to achieve increased availability and performance with respect to primary-copy and ROWA-type epidemic approaches. However, none of these proposals addressed the security issues we study in this paper.

Liu *et al.* [23] addressed the issue of backing out malicious but committed transactions. This work is complementary to our study in this paper, and can be used by a server that incorrectly committed an update by not supporting a sufficient degree of tolerance. Agrawal and El Abbadi [1] used quorums to preserve confidentiality of replicated data despite the disclosure of the contents of a *threshold* of the repositories. Ray *et al*. [29] presented a locking-based, advanced secure commit protocol for multi-level secure distributed databases. Malkhi and Reiter [25] investigated Byzantine failures in quorum systems in a synchronous, non-epidemic setting.

The work most closely related to ours is that by Malkhi *et al*. [24], which provides an analytical treatment of epidemic-style update diffusion (i.e., propagation) algorithms that are tolerant of Byzantine faults. This work assumes that less than t replicas fail and each update is initially received by at least t non-malicious replicas. Each non-malicious replica commits an update only it receives the update from t others. Although our protocols do not handle *fully* Byzantine failures (since we use digital signatures for authenticating forwarded protocol information), we do not make any assumptions about how and where updates are generated and initially received. We also integrate our security extensions on top of decentralized consistency protocols (i.e., voting and ROWA) and provide strong-consistency, whereas [24] does not address consistency issues and assumes a non-transactional setting.

The security protocols described here are also related to traditional security in group communication systems and protocols. Ensemble [31] addresses only external security threats, whereas Rampant [30] is designed to handle Byzantine attacks. Castro and Liskov [7] described a practical replication algorithm for tolerating Byzantine attacks in asynchronous environments. These protocols are commonly based on primary-copy models to coordinate replica management and require much stronger connectivity and reliable multicast primitives than is required by Deno.

Finally, our work is different from existing secure election protocols (e.g., [10, 12]) 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. In Deno, voter privacy is not an issue and the weakly connected nature of the underlying network makes reliance on central authorities untenable.

8. Conclusions

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

We evaluated the cost of our security protocols using the Deno prototype replicated system. The experimental results revealed that: (1) protecting against internal threats comes at a cost, but the marginal cost for protecting against larger cliques of malicious insiders is generally low; (2) our decentralized protocol performs, scales, and handles update contention significantly better than a ROWA-based decentralized protocol; (3) our approach allows servers to trade off performance and the degree of tolerance to malicious insiders, and; (4) individual servers can support different degrees of tolerance without adversely impacting the performance of other servers, allowing servers to set arbitrary degrees of tolerance based on their individual requirements and resources.

9. References

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

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

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

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

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

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

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

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

[9] U. Cetintemel, P. J. Keleher, and M. J. Franklin. Support for Speculative Update Propagation and Mobility in Deno. University of Maryland, UMIACS-TR-99-70, Oct. 29, 1999.

[10] L. Cranor and R. Cryton. Sensus: A Security-Conscious Electronic Polling Scheme for the Internet. In *Proc. Hawaii Intl. Conf. on System Sciences*, 1997.

[11] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th ACM Symp. on Principles of Distributed Computing (PODC)*, Vancouver, 1987.

[12] 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.

[13] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database System. *ACM Transactions on Database Systems*, 7(2):209-234, June 1982.

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

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

[16] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi. Epidemic Quorums for Managing Replicated Data. In *Proc. 19th IEEE Intl. Performance, Computing, and Communications Conf. (IPCCC)*, Phoenix, 2000.

[17] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Transactions on Database Systems*, 15(2):230-280, 1990.

[18] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozie, and L. Greif. Replicated Document Management in a Group Communication System. In *Proc. Conf. on Computer Supported Cooperative Work*, 1988.

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

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

[21] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc.13th ACM Symp. on Operating Systems Principles (SOSP)*, October 1991.

[22] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computing Systems*, 10(4):360-391, November 1992.

[23] P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases*, 8(1):7-40, 2000.

[24] D. Malkhi, Y. Mansour, and M. Reiter. On Diffusing Updates in a Byzantine Environment. In *Proc. 18th IEEE Symp. on Reliable Distributed Systems (SRDS)*, Lausanne, 1999.

[25] D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *Proc. 29th ACM Symp. on Theory of Computing*, 1997.

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

[27] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on Optimistically Replicated Peer-to-Peer Filing. *Software--Practice and Experience*, 28(2):155-180, February 1998.

[28] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable Update Propagation in Epidemic Replicated Databases. In *Proc. Fifth Intl. Conf. on Extending Database Technology (EDBT)*, Avignon, 1996.

[29] I. Ray, E. Bertino, S. Jajodia, and L. Mancini. An Advanced Commit Protocol for MLS Distributed Database Systems. In *Proc.3rd ACM Conf. on Computer and Communications Security*, New Delhi, 1996.

[30] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proc. Second ACM Conf. on Computer and Communications Security (COMSEC)*, Fairfax, 1994.

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

[32] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. of the Winter 1988 USENIX Conf.*, 1988.

[33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in a Weakly Connected Replicated Storage System. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 1995.

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