Trusted CVS

Muthuramakrishnan Venkitasubramaniam Ashwin Machanavajjhala David Martin
Johannes Gehrke
Department of Computer Science, Cornell University
{vmuthu,mvnak,djm,johannes}@cs.cornell.edu

Abstract

The CVS (Concurrent Versions System) software is a popular method for recording modifications to data objects, in addition to concurrent access to data in a multi-user environment. In current implementations, all users have to trust that the CVS server performs all user operations as instructed. In this paper, we develop protocols that allow users to verify that the server has been compromised, and that it has performed exactly the users' operations on the data. We first show that communication between users is necessary to guarantee that users can detect that the server has been compromised. We then propose efficient protocols that fast enable detection of server integrity under CVS workloads. Our techniques also have applications in the outsourcing model where multiple users own a common database maintained by an untrusted third-party vendor.

1 Introduction

The Concurrent Versions System (CVS) is a popular method for recording modifications to data objects, especially for large software development projects. In current implementations, the data objects are stored and maintained on a CVS server which provides concurrent read and update access to the data to multiple users. However, current implementations of the system require the users to completely trust the CVS server. However, a malicious server can disrupt the functioning of the CVS by violating the integrity and the availability of the system in two ways:

- *Single-User Integrity Violation*: A malicious CVS server can modify data even though the user did not request the server to make any updates.
- *Single-User Availability Violation*: A malicious CVS server may not perform a data update even though the user has submitted a data update.

There are existing solutions which can detect integrity and availability violations such as certificate revocation [6] and authenticated data publishing [2] which involve untrusted servers. In both these solutions, data is stored in an untrusted server and multiple users read from this server. The data is stored in an *authenticated data structure* using which the untrusted server can prove to the users that integrity and availability are not violated [2]. For instance, in the certificate revocation case, the user can verify that the untrusted server has neither maliciously revoked some certificates nor neglected to report any of the revoked certificates. These protocols support multiple users who can read the data, however there is only one user, the data owner, who can update the data on the server.

In systems where multiple users can update files, like in the case of CVS, a malicious server can violate availability even when each user believes that the server has not neglected any of their updates to the data.

- *Multiple-User Availability Violation*: The CVS server can make one set of users believe that the other users are not modifying any files.

In this paper we propose protocols which enable detection of integrity and availability violations by an untrusted server when multiple users can modify the data on the untrusted server. Our solutions will be geared towards constructing a trusted CVS system. Our techniques, however, have wide applicability to any scenario where a common database which is maintained by an untrusted server operated upon by several clients.

Desiderata for Trusted CVS Protocols. Let us first discuss some desiderata for the solutions we are interested in. First, we want our protocols to be able to detect integrity and availability violations as soon as possible in order to limit the amount of rollback that might be necessary. More specifically, we want to be able to formally bound (by some metric) how long it takes to detect a violation. In particular, we will use both time and the number of operations performed after the violation as metrics in our protocols. Second, to be applicable to the CVS context, we require the protocol to be able to detect violations even when the users are inactive for arbitrarily long periods of time. Third, we

want the users to be able perform any operations on the data at the server with only a minimum overhead due to any overhead involved by our protocols compared to a traditional CVS system. Fourth, we want all messages either to be sent to or be received from the server; i.e., we want only minimal external communication among the users. External communication requires synchronization between users that is hard to obtain for CVS users. Fifth and last, we want protocols that do not require much resources at the users, and thus we require that the amount of state maintained by each user is bounded by a small constant amount of memory.

Our Approach. Let us shortly survey our approach to building a trusted CVS.

We model an untrusted CVS by investigating how differently it behaves from a CVS executing on a trusted server. We say that the untrusted CVS server has *deviated* if the users do not receive query results identical to the results returned when using a trusted server. Intuitively, a deviation from correct behaviour implies that either integrity or availability has been violated. Our aim is then to design protocols where users trust each other and can detect deviant behaviour of the server. In particular, we are interested in the case where deviant behavior can be detected within the time that some user performs k further updates on the data. Such a protocol is said to posses the property of k-*bounded deviation detection* (Section 2.2.1). We model the activity of users using the concept of a *workload* and require our protocols to permit workloads wherein users can be offline for long periods of time. We formulize the requirement that our protocol for the untrusted server does not impose too large an overhead for carrying out user operations on the data as compared to a trusted server using the concept of *bounded workload preservation*. In Section 2, we give formal definition of these properties.

Under the assumptions that the CVS server completes every transaction in bounded time, and that the users are partially synchronous (i.e., their local clocks do not drift arbitrarily), we show in Section 3 that in the absence of communication between users, there is no protocol which guarantees bounded deviation detection, bounded workload preservation, and permits typical CVS workloads.

Section 4 contains our main contributions. We propose efficient protocols for implementing a CVS on an untrusted server while guaranteeing k-bounded deviation detection. In the first two protocols, we assume that there is a broadcast channel between the users. The third protocol does not assume existence of external channels of communication, but restricts the kind of workloads allowed — every user needs to perform at least two operations every t time units. The idea behind the first two protocols is that every user verifies that the CVS server has performed a correct state transition while performing an operation, and once in a while the users synchronize using the broadcast channel to check that state transitions can be pieced together to form a single correct execution. The third protocol simulates a broadcast channel using the untrusted server; this is possible since every user is guaranteed to perform at least two operations every once a while. To the best of our knowledge, these three protocols are the first feasible protocols for implementing multi-user data management on an untrusted server efficiently.

We describe related work in Section 5 and we conclude in Section 6.

In closing this introduction, we would like to point out in this preliminary work we exclude all types of failures — for example, unreliable message delivery or crashes of the users or the server. Failures are outside the scope of this paper, and we leave extensions of our protocols to this case to future work.

2 Formalisms

In this section we adopt the framework for multi-agent systems from the book by Halpern et al [3] to formally model our system. In Section 2.1, we describe the basic system model. In Section 2.2 we define the protocol desiderata.

2.1 System Model

In a CVS system, there are $n + 2$ *agents*, namely, the CVS database server, the n users, and the environment. At all times, each agent is associated with a *local state*. The local state of the CVS server includes the data stored at the server. The local state of the users may include the history of modifications requested by the user. The local state of the environment includes the global clock (which the other agents may or may not have an access to), information about messages in transit and everything else that is relevant to the system. A global state is the $(n + 2)$-tuple of local states. The global state describes the system at any given point in time.

The state of the system is constantly changing over time. A *run* of the multi-agent system intuitively represents a possible execution sequence of agents in that system. A run is a function from time to global states. A pair (r, m) consisting of a run r and time m is called a *point*. $r(m)$ denote the state of the system at that point. The time here co-incides with the global clock maintained by the environment. Global states change as a result of actions. Actions are performed in *rounds* – round m takes place between time $m - 1$ and time m. An agent is said to *know* a fact at some point (r, m) if the fact is true at all points in the system where the agent's local state is the same as its local state at (r, m).

Synchrony: An agents local clock is said to "tick" every time its local state changes. An agent's local clock might be slower than the global clock in the environment.

While it is unreasonable to assume that all the users and the server share the same clock as the environment (perfect synchrony), it is usually not the case that the local clocks are arbitrarily slower than the global clock (asynchrony). Partial synchrony is a reasonable middle ground. We assume p-*partial synchrony* where every user's local state changes at least once every p rounds.

Communication: We do not explicitly model messages in our system. Instead, message sent and received by the agents can be modeled by appropriate changes to the local states of the sender and the receiver. We assume local states of the users and the CVS server accommodate message queues from which messages are handled in order. We assume that messages are not lost and are delivered in bounded time. Without loss of generality, we only consider runs of the system where messages are delivered in a single round.

CVS Operations: For simplicity of presentation we assume that only two operations are allowed on the files on the CVS server – *checkout <file names>* and *commit <file names>*. The operation *checkout <file names>*, is a read request which returns the current version of the files on the CVS server. The *commit <file names>* operation commits the changes made by the user in the files specified. To further simplify the model (and to widen the applicability of our result), we consider the CVS server as a database of data items. The checkout operation is modeled as a *read* request on the database. The commit operation is modeled as an *update* request on the database.

CVS operations on a Trusted Server: We first model CVS operations in a system with a trusted CVS server. CVS operation requested by a user are called transactions. The start and end of transactions are marked by a *query action* and a *response action*, respectively. The query action represents a message sent to the server requesting an operation on the data. The response action models a message from the server telling the user that the operation has completed and returning results if any. For simplicity, we assume that at most one query action occurs per round. The trusted CVS server is assumed to execute the requested operations in a serial order mirroring the order in which requests are received. We assume $b_\star$-*bounded transaction time*; i.e., the response action corresponding to every query action occurs within b_* rounds.

Untrusted Server and Deviation: We now can model our untrusted CVS system and formally express our goal. An untrusted server can maliciously violate the integrity and availability of the system. We model violations by the untrusted server using the concept of deviation between runs. Intuitively, an untrusted server is acting maliciously if it does not behave "similar" to the trusted system.

Definition 2.1 (Deviation) *A prefix of a run r* deviates *from a run r' if there is some prefix of r' such that:*

1. *The set of query and response actions that occur in the prefix of r is not identical to the set of query and response actions that occur in the prefix of r', or*
2. *The order in which the query and response actions occur in the prefix of r is different from the order in which they occur in the prefix of r'*

We say that a run r deviates from a run r' *if some prefix of r deviates from r'. We say that the untrusted server* deviates *if some run in the untrusted system deviates from all possible runs in the trusted system.*

The only difference between a run r which does not deviate from a run r' is that the time points at which the query response events happen in the two runs can be different. We will use this notion to define one of our desiderata, bounded workload preservation, in Section 2.2.3.

We have now formally expressed the concept of deviation. We can use this to formally express our goal – to design protocols where users can detect if a run in the untrusted system deviates from all runs in the trusted system.

Workload: We need to define one more concept to enable us to compare the untrusted system with the trusted system. Consider a run r in the trusted system which involves a sequence of operations on the data at the server. We call this sequence of operations on the data a *workload*. Any protocol specifies a set of possible runs R. We say that a protocol *permits* a workload if there exists a run in R wherein the operations on the data occur in the same order as in the workload.

2.2 Protocol Desiderata

Not only do we want our protocols to enable the users to detect deviation, we also want the protocol to have the following desirable properties.

2.2.1 Bounded Deviation Detection We are interested in protocols which enable users to detect that the untrusted server has deviated from the trusted system preferably sooner than later. In fact, we would like to bound the detection delay in terms of the number of transactions that are requested after the first deviation.

We say that the system exhibits k-*bounded deviation detection* if for any run r in the untrusted system, if a prefix of r of length m deviates from all runs in the trusted system, then some user knows this fact before any user completes more than k transactions that were initiated after round m. (Note that we are using the formal definition of user knowledge from Section 2.)

We only require that some user realizes this fact (as opposed to requiring that the fact become common knowledge amongst all users) because we assume that the first user to detect deviation will leave the system at that point and use

an external mechanism (e.g. law enforcement) to broadcast this information to all other users.

The motivation behind bounding delay this way is that no user will lose more than k transactions after the server has deviated. We could required a stronger condition – the protocol should enable deviation detection before any k further operations are performed on the data, and not k operations per user. We only give protocol for the weaker requirement.

2.2.2 CVS Workloads Ideally, we would like the untrusted system to handle a variety of workloads typically found in CVS applications. For example, we want our system to guarantee bounded deviation detection in cases where some users sleep (i.e., go offline) indefinitely since this often seems to be the case with actual CVS users in real life. Unfortunately, as we shall see in the next section, some naturally occurring classes of workloads make it difficult or even impossible for protocols to provide bounded deviation detection.

2.2.3 Bounded Workload Preservation We show a simple protocol which motivates the workload preservation requirement. Consider a system with an untrusted server and a single user. Techniques from authenticated data publishing ensure that integrity and availability are not violated [2]. In these protocols, every time the data is modified by the server (on a request by the data owner), the server returns a *verification object* back to the owner. The owner can use this verification object to verify that single-user integrity and availability violations have not occurred.

We can use a simple extension of this protocol for the mutli-user case as follows. The protocol forces users to update the data only at pre-specified time points (say, on the hour) and only in a pre-specified order. All users know the initial state of the data. The first user performs an update, and verifies that the server has not deviated using the verification object. Then the user signs this verification object and the update performed and stores it on the server. This goes on in a token passing style cycling through the users. If a user does not have an operation, a signature of a null message is stored. Since the signatures are not forgeable, and since users are guaranteed to store a signed object in their turn, we have simulated the single user protocol. Thus if the single user protocol can detect deviation, so can this multi-user protocol.

We want protocols that permit any user to be able to do anything in the untrusted system that she could do in the trusted system, with only a reasonable delay in verification overhead. In a workload where a user performs two operations in succession, the above protocol forces the user to wait for all the other users to write null records to the server before performing her second operations! We want to disallow such protocols which drastically slow down certain workloads, and we make this formal as follows.

We say that the system exhibits c*-workload preservation* if for each possible run r_* in the trusted system there is some run r in the untrusted system such that

1. run r does not deviate from r_*, and
2. the number of rounds between any two events is at most the number of rounds between those events in r_* plus c times the number of query and answer events that occur between the two events. (Intuitively, c represents the overhead of integrity verification per ordinary transaction.)

2.2.4 Absence of External Communication In a CVS system on a trusted server, all communication is only between the users and the server. There is no communication between users. For an untrusted server, we also want protocols that do not require communication between the users. We call any communication between users *external communication*.

More precisely, we say that a system has *no external communication* if the local state of each user in round $m+1$ is a function of the user's local state in round m and the messages received by the user from the server in round $m+1$, in case there are any. This means that the local state of each user is conditionally independent of the local states of the other users given the state of the server. Hence, if the users want to communicate with each other, they can do so only through the server.

2.2.5 Bounded Local State with Users We want protocols that bound the local state at each user by a small constant amount of memory. This means, for example, that the protocol should not expect the users to maintain the complete history of the requests that they ever posed to the server.

3 Necessity of External Communication

Though all the desiderata outlined in the previous section are very reasonable and desirable, we will show in this section that there is no protocol with all of these desiderata. More precisely, we show that in the absence of reliable external communication, there is no protocol which permits CVS workloads and guarantees bounded workload preservation and k-bounded deviation detection.

To show the necessity of reliable external communication, we introduce a class of workloads which we call *partitionable workloads*. These workloads are very reasonable in a CVS system. We then show that there is no protocol which guarantees k-bounded deviation detection and bounded workload preservation, while permitting a partitionable workload in the absence of reliable external communication.

3.1 Partitionable Workloads

We motivate this class of workloads with an example. Consider a code-base in CVS which is jointly edited by a programmer in the US and a programmer in China. The two programmers work quite independently. They work at different times (due to the time difference) and on different parts of the source code, occasionally changing some common header files. It is therefore quite reasonable to assume that the programmer in the US changes a common header file, say Common.h and goes offline. Before this programmer comes back up, the programmer in China might make a change which is dependent on Common.h. We call the change made by the Chinese programmer *causally dependent* on the change made by the US programmer. The Chinese programmer might go on to make $k+1$ other changes before the American comes back up. We describe such a situation abstractly as follows.

We say that the system *permits unboundedly partitionable workloads* if, for every number k, there is a partitioning of users into sets A and B and two runs r_A and r_B in the untrusted system and rounds $m < m_A < m_B < m'$ such that:

1. r_A and r_B have some common prefix up to round m (see Figure 1).
2. the prefix of r_A up to m' does not deviate from some run in the trusted system.
3. the prefix of r_B up to m' does not deviate from some run in the trusted system.
4. some transaction t_2 in r_B that gets issued and completed strictly between rounds m_A and m_B is causally dependent on the existence of some transaction t_1 in r_A that gets issued and completed strictly between rounds m and m_A, and
5. in r_B, some user (in B) issues and completes at least $k+1$ transactions between rounds m_B and m'
6. in r_A, no user in B issues or completes any transactions (neither operations nor verifications) between rounds m_A and m'
7. in r_B, no user in A issues or completes any transactions (neither operations nor verifications) between rounds m_A and m'

Under such a workload, an untrusted server can mount a partitioning attack where the US programmer is led to believe that change to Common.h ($t1$) has been completed and the Chinese programmer is led to believe that t_1 did not happen. This is illustrated in Figure 1 as run r. Run r deviates from all possible runs in the trusted system. This deviation and cannot be discovered until the US programmer comes up, by which time the other programmer has performed more than k operations. Hence, k-bounded deviation cannot be guaranteed.

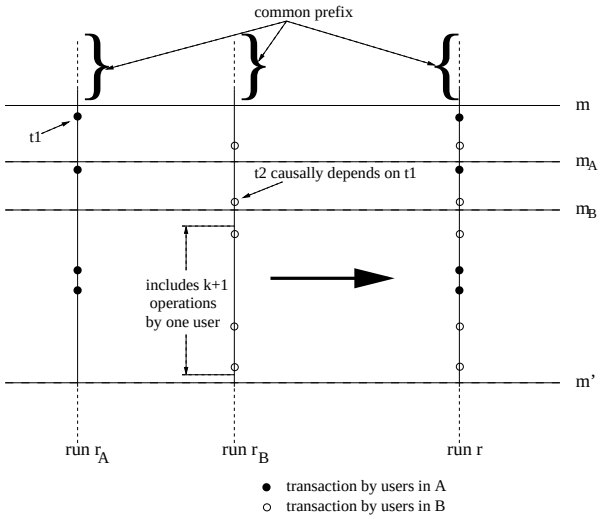


Figure 1. The Partition Attack

Theorem 3.1 *If the untrusted system lacks external communication, and permits unboundedly partitionable workloads, then k-bounded deviation detection is not possible for any k.*

The proof of this theorem is omitted due to space constraints.

4 Protocols

In this section, we present three protocols that guarantee deviation detection. From Theorem 3.1, we know that there are no protocols that can guarantee bounded deviation detection while at the same time permitting typical CVS workloads without the use of external communication. Our first protocol guarantees bounded deviation detection while permitting less restrictive workloads by using external communication. Our second protocol improves on the first in efficiency. Finally, our third protocol guarantees deviation detection within bounded time (as opposed to within a bounded number of operations) with no external communication, by restricting the permitted workload.

Before describing our protocols, we first describe the data structure we will build upon to verify that the server has not violated integrity – Merkle Trees [7].

4.1 Merkle Trees

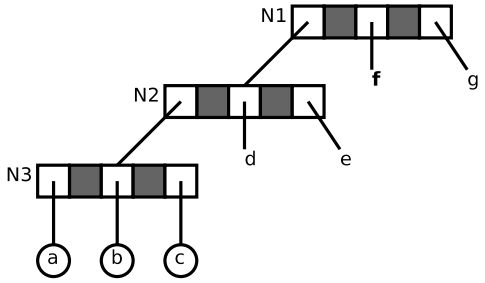
A Merkle Tree [7] is a B$^+$-tree with digests. In a B$^+$-tree [15], the leaf nodes of the tree contain data, and the internal nodes contain keys and tree pointers. Each internal node has up to m keys and $m+1$ pointers to children, where $m+1$ is the maximum permissible branching factor of the B$^+$-tree.

In a Merkle Tree, each node also stores a *digest*. The digest stored in a leaf node is the hash of the data stored at that node. A collision intractable hash function, for example as described in [2], is used. The digest stored in an internal node is a hash of the concatenation of the digests of the node's children. Figure 2 shows a root to leaf path in a Merkle Tree. $N3$ is a leaf node and $N2, N1$ denote internal nodes. $N1$ has no parent node and hence is the root of the tree. $a, b, \ldots, g$ denote digests. Digests a, b and c are hashes of values stored in node $N3$. Digests d and e are the digests stored at the siblings of node $N3$. The digests that will be stored in each of the nodes $N1, N2$ and $N3$ are shown in the table in the Figure 2. The hash of the root of this tree is called the *root hash*.

A change in any of the data values in the tree will change the digests on the path from the corresponding leaf node to the root node. For example, if a value in the node $N3$ is updated, the digests at $N3, N2$ and $N1$ change; the other parts of the tree are not affected. Assume that after the update in $N3$, the new hashes of $N3$'s data values are a', b and c, respectively,, and if we know d, e, f and g, then we can recompute the digests along the path and thus compute the new root hash. Since the height of the tree is bounded by $O(\log n)$ and the number of siblings of any node is bounded by a constant m, for a single update we only need to know $O(\log n)$ other digests to recompute the root hash.

Since an insert or delete operation affects only $O(\log n)$ elements of the B$^+$-tree, given the operation (insert or delete) and the digests of the $O(\log n)$ siblings of the affected nodes, it is possible to recompute the root digest of the Merkle Tree before and after the operation has been performed using $O(\log n)$ digests.

We now describe how the above schema enables a single user to verify operations on the database. We assume that the current root digest $\mathcal{M}$ is known to the user. Given an update query, Q, the server returns the new root hash and the digests of the $O(\log n)$ nodes required to compute the old and new root digests. We call these $O(\log n)$ digests the *verification object of update* Q, denoted by $v(Q, D)$. Using $v(Q, D)$, the user recomputes the old root digest and verifies that it is equal to $\mathcal{M}$. By doing so the user verifies that the verification object returned by the server is correct. The user then computes the new root digest of the tree and compares it with the returned new root hash. This helps to check if the update was performed correctly. Finally, the user sets $\mathcal{M}$ to the new root digest to be ready to verify the next operation.



Node	Digest
$N3$	$h(a \parallel b \parallel c)$
$N2$	$h(N3 \parallel d \parallel e)$
$N1$	$h(N2 \parallel f \parallel g)$

Figure 2. A path in a Merkle Tree

We will use some notation from this section in the remaining discussion. We denote by $\mathcal{M}(D)$ the root digest of the database D. Recall that given a query Q, $Q(D)$ represents the answer to the query and, $v(Q, D)$, the verification object for query Q on D.

4.2 Protocol I

In this protocol, when a user submits a query, the server is required to return a message containing a root digest signed by the last user to perform an operation. The user then verifies the signature, calculates the new root digest, and returns a signed copy of it to the server. Thus, the user verifies its operation on the database. Notice, however, that this does not prevent a partition attack, where the server returns an out-of-date signed root digest (see Section 3). Our protocol therefore requires users to communicate every k operations in order to verify that the server produced the current states for all operations. We now describe the protocol in detail.

We assume the existence of a public key infrastructure, for example as in [4]; it is used to verify digital signatures. We write $sign_i(x)$ for the result of user i signing message x. In this protocol,

- The server maintains the count of number of operations performed on the database, denoted by ctr.
- User i maintains the count of the total number of operations it has performed, denoted by $lctr_i$, and the last value of ctr it has seen, denoted by $gctr_i$.

At the beginning of the protocol, $lctr_i$, $gctr_i$ and ctr are initialized to 0. We denote the initial state of the database by D_0. The root digest of the initial database state is $\mathcal{M}(D_0)$, and is assumed to be common knowledge. Some user j

Notation	Description
$Q(D)$	Answer to query Q
$v(Q, D)$	Verification object
ctr	Total number of operations performed
j	User with the most recent operation on D
sig	$sig_j(\mathcal{M}(D) \parallel ctr)$

Table 1. Notation

is elected to sign $h(\mathcal{M}(D_0) \parallel 0)$ and send the result (i.e., $sign_j(h(\mathcal{M}(D_0) \| 0))$) to the server. When any user i issues a query Q, the server returns $(Q(D), v(Q, D), ctr, j, sig)$ where, $Q(D)$ is the answer to query Q, $v(Q, D)$ is the associated verification object, and $sig = sig_j(\mathcal{M}(D) \parallel ctr)$ is the root hash concatenated with the counter ctr signed by user j. Table 1 gives an overview of this notation.

Given the verification object, $v(Q, D)$, the user can compute $\mathcal{M}(D)$ and hence can also compute the value of $h(\mathcal{M}(D) \parallel ctr)$. The user then verifies that sig is indeed $h(\mathcal{M}(D) \parallel ctr)$ signed by user j. We call such a sig *legitimate* because it cannot be forged by the server. If sig is legitimate, the user increments its local operation count $lctr_i$, updates $gctr_i = ctr + 1$, computes the new root digest $\mathcal{M}(D')$, and sends a signed copy of $h(\mathcal{M}(D') \| ctr+1)$ back to the server. If sig is not legitimate, the user terminates and reports an error.

The first user to complete k operations announces a "sync-up" message on the broadcast channel. On receiving the "sync-up" message, all users broadcast their local operation counts $lctr_i$ after completing their current transactions. Furthermore, they do not start a new transaction between the "sync-up" message and broadcast. User i reports success if $gctr_i = \sum_k lctr_k$. If all users report the check unsuccessful, they terminate and report an error. If the server is not malicious, there is some user i for whom $gctr_i = \sum_k lctr_k$.

Theorem 4.1 *Protocol I guarantees bounded deviation detection and bounded workload preservation.*

Proof (sketch): Given a legitimate sig (i.e., $sig = sign_j(h(\mathcal{M}(D) \| ctr))$), our assumptions on hash functions make it intractable for the server to find a database D^{bad} and ctr' such that $sig = sign_j(\mathcal{M}(D^{bad}) \parallel ctr')$. Hence, the server can not forge a legitimate state. Thus every increment in ctr is accompanied by an increment in $lctr_k$ for some user k. The total number of increments seen is captured by $\sum_k lctr_k$. If the server presents the same legitimate sig for two different operations, then a single increment in ctr is accompanied by two increments in the $\sum_k lctr_k$, once for each of the two operations. This would mean that at the synchronization step, $gctr_i$ would not be equal to $\sum_k lctr_k$ for all i.

Description of Protocol I

1: **Initialization:**
2: Some user j is elected to sign $h(\mathcal{M}(D_0) \parallel 0)$ and send it to the server.

1: **Query Operation:**
2: User i **sends** query Q^i to the server.
3: Server **sends** Φ to user i, where $\Phi = (Q^i(D), v(Q^i, D), ctr, j, sig)$
4: **if** sig is legitimate **then**
5: User sets $lctr_i \leftarrow lctr_i + 1$ and $gctr_i \leftarrow ctr + 1$.
6: User **sends** $sign_i(h(\mathcal{M}(D') \parallel ctr + 1))$ to server.
7: **else**
8: User terminates and reports and error.
9: **end if**
10: Server sets $ctr \leftarrow ctr + 1$.

1: **Synchronization:**
2: All users broadcast $lctr_i$ to the other users
3: User i broadcasts success if $gctr_i = \sum_k lctr_k$.
4: If no user broadcasts success they terminate and report an error.

The protocol guarantees bounded deviation detection because the synchronization is guaranteed to be performed when the first user completes k operations. The communication overhead due to verification process occurs in step 6 of the query operation where an extra message is sent from the user to the server. This is a constant communication overhead for each operation and hence the protocol guarantees bounded workload preservation ■

4.3 Protocol II

In Protocol I, we notice that after the server responds to a query, it waits for the user to return the signature of the current root digest in another message. Only after receiving this signature, the server can answer the next query. This additional blocking step affects throughput in systems with frequent updates. Also, the protocol requires a public key infrastructure. In this section, we give a protocol that avoids that extra message while still guaranteeing bounded deviation detection. Furthermore, this protocol does not use digital signatures, and hence does not need a PKI.

In this discussion, we associate the state of the database with the value $h(\mathcal{M}(D) \| ctr)$. We notice that every state of the database is seen by at least by two users, except the initial state and current state which is seen by one user. While synchronizing every k operations, if the users could verify this property, they are guaranteed that the states were part of a single sequence, each transition of which is seen by exactly one user.

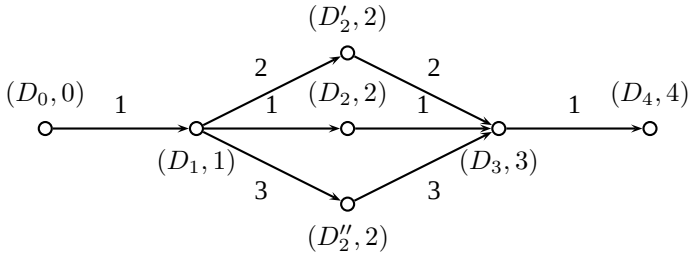


Figure 3. Scenario 2

A first attempt towards this, would be for user i to keep a register σ_i, that contains the XOR of all states they see. During synchronization, they XOR all their registers. All states that occur twice, would cancel out. Only the XOR of the first and last state would remain.

This however does not work for a simple reason. Take the scenario depicted in Figure 3. The nodes are represented as states and counter values. The node labels must be interpreted as $h(\mathcal{M}(D) \parallel ctr)$. The label on the edge of a transition depicts the user that validated that transition. In this case the XOR's of all intermediate nodes cancel out to give the first and last, since the intermediate nodes have even degree. Thus, the untrusted system can violate availability by replaying the same state to multiple users.

We however notice that in this graph, there are multiple transition with the same end state (eg: $(D_3, 3)$). Since each user maintains $gctr_i$, which is the last seen ctr value, the two transitions cannot be seen by the same user i. This is because, user i would detect an error when the server produces the same ctr value for two of its operations. Hence, the transitions were seen by different users. Suppose, we tag the new state $h(\mathcal{M}(D')\|ctr+1)$ of a transition with the user that performed the transition, i.e. if user j saw the transition from (D, ctr) to $(D', ctr+1)$ then the state $(D', ctr+1)$, is represented as $h(\mathcal{M}(D')\|ctr+1\|j)$. This would force only a single transition into a state. We claim that, this is enough to guarantee that the graph is a directed path from the initial state to the last state. We summarize the conditions of the graph in Lemma 4.1. Since our protocol satisfies the conditions of the Lemma, we only need that the users perform the XOR operation every k operations to get Theorem 4.2.

Lemma 4.1 *Consider a directed graph $G(V, E)$ with the following properties,*

P1. There are no isolated vertices

P2. In-degree of every vertex is at-most 1.

P3. There are no directed cycles.

P4. Exactly two vertices have an odd total degree (sum of indegree and outdegree). One of these vertices has indegree 0.

Then, G is a directed path. ■

Description of Protocol II

1: **Query Operation:**
2: User i sends query Q^i to the server.
3: Server returns to user i, $(Q^i(D), v(Q^i, D), ctr, j)$ and increments ctr by 1.
4: User i reports error if $ctr \leq gctr_i$.
5: User i computes $\mathcal{M}(D)$ and $\mathcal{M}(D')$.
6: User i updates local value

$$\begin{aligned} \sigma_i &= \sigma_i \oplus h(\mathcal{M}(D) \parallel ctr \parallel j) \\ &\quad \oplus h(\mathcal{M}(Q(D)) \parallel ctr+1 \parallel i) \\ last_i &= h(\mathcal{M}(Q(D)) \parallel ctr+1 \parallel i) \\ gctr_i &= ctr+1 \end{aligned}$$

1: **Synchronization :**
2: All users broadcast σ_i to the other users
3: User i broadcasts success if $h(\mathcal{M}(D_0) \parallel 1) \oplus last_i = \bigoplus_k \sigma_k$.
4: If no user broadcasts success they terminate and report an error

---

Theorem 4.2 *Protocol II guarantees bounded deviation detection and bounded workload preservation.*

Proof(sketch): We visualize the state of the database as $h(\mathcal{M}(D) \parallel ctr \parallel i)$ where D is the database seen by user i for some query, as a node in a graph. This graph contains nodes corresponding to all the states the users saw. A directed edge from node $u = h(\mathcal{M}(D) \parallel ctr \parallel i)$ to $v = h(\mathcal{M}(D') \parallel ctr+1 \parallel j)$ occurs if for some query by user j, the server returned $(v(Q, D), ctr, i)$ and $\mathcal{M}(D') = \mathcal{M}(Q(D))$. For every edge there is some user who has seen the transition from one node to the other. The initial state of the database is a special node represented by $s = h(\mathcal{M}(D_0) \parallel 1)$.

Assuming that the synchronization step completed successfully, we show that the graph representing the states of the database satisfy the properties of Lemma 4.1.

P1. There are no isolated vertices. This is true, since all nodes in the graph are part of some transition and have an edge corresponding to that transition.

P2. In-degree of every vertex is at-most 1. Suppose u, v and w are three nodes such that (u, w) and (v, w) are two edges (therefore $indegree(w) > 1$). Let

$$\begin{aligned} u &= hash(\mathcal{M}(D_u) \parallel c_u \parallel o_u) \\ v &= hash(\mathcal{M}(D_v) \parallel c_v \parallel o_v) \\ w &= hash(\mathcal{M}(D_w) \parallel c_w \parallel o_w) \end{aligned}$$

Since (u, w) and (v, w) are directed edges, $c_w = c_u + 1$ and $c_w = c_v + 1$ which means $c_u = c_v$. Further user o_w must have performed both these operations. This means the publisher produced the same counter value ($o_u = o_v$) to the owner o_w twice which would have been detected at step 4 in that Query operation.

P3. There are no directed cycles. Since the counter value increases along every directed edge there cannot be any loop.

P4. Exactly two vertices have an odd total degree. One of these vertices has in-degree 0. During synchronization, when all σ_i's are cumulatively XOR'ed, nodes with even degree cancels out. $last_j$ is a state in the graph and is forced to have an odd degree for some user j. Further the initial state is forced to have odd degree too. A third vertex cannot have odd degree because it would appear in the XOR and fail in the check. Hence only two nodes can have odd degree.

From Lemma 4.1, it follows that the graph is a single directed path. This means that, the server acted correctly, if the synchronization step terminated successfully. Since, we are guaranteed that the synchronization step occurs when the first user completes k operations this protocol guarantees bounded deviation detection. This guarantees bounded workload preservation because because it has no extra messages for every operation. ■

4.4 Protocol III

Protocols I and II assume that there is a broadcast channel among the users. When the number of users is large it is unrealistic to assume a broadcast channel among all users. Furthermore, it requires that all users to be online simultaneously whenever the synchronization step is done.

In this section, we present a third protocol that guarantees bounded detection where users do not have any external communication channels. However, we restrict the workload that the protocol permits. We assume that every user performs at least two operations every t time units. The advantage of this solution over the previous two protocols is that the users are not required to be online simultaneously.

As in Protocol II users maintain the registers σ_i and $last_i$.

For this protocol we shall define one epoch to be t time units. We assume that every user performs at least two operations every epoch. We illustrate the protocol in terms of epochs with a time line shown in Figure 4. Consider three consecutive epochs $e, e+1$ and $e+2$. User i in its first operation in epoch $e+1$ (at A) is informed by the server that it entered a new epoch. It takes a back up of its last state from epoch e. This is stored in the server in the second operation of epoch $e+1$ (at B). At the end of epoch $e+1$ it is guaranteed that all users have stored their states

Description of Protocol III

1: **Query Operation:**
2: User i sends query Q^i to the server. If this is the second operation in a new epoch then along-with the query the local states of the previous epoch is sent with a signature.
3: Server returns to user i, $(v(Q^i, D), ctr, j)$ and increments ctr by 1.
4: User i checks if $ctr > ctr_i$ else reports error
5: User i computes $\mathcal{M}(D)$ and $\mathcal{M}(D')$.
6: User i updates local value

$$\begin{aligned} \sigma_i &= \sigma_i \oplus h(\mathcal{M}(D) \parallel ctr \parallel j) \\ &\quad \oplus h(\mathcal{M}(Q(D)) \parallel ctr+1 \parallel i) \\ last_i &= h(\mathcal{M}(Q(D)) \parallel ctr+1 \parallel i) \end{aligned}$$

7: If the server indicates start of new epoch then user i takes backup of local states σ_i and $last_i$. The user retrieves the stored local states from the epoch before last and verifies consistency, i.e. checks if there exists k such that

$$h(\mathcal{M}(D_e) \parallel l_e) \oplus last_i = \bigoplus_k \sigma_k$$

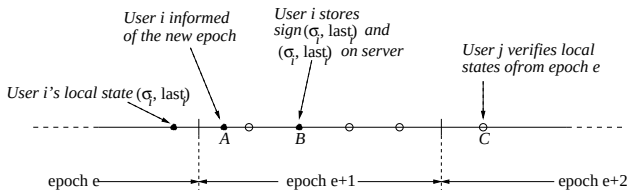


Figure 4. Epochs in Protocol III

from epoch e at the server. Some user j, in epoch $e+2$ (at C), takes the local states of all users from epoch e and runs the synchronization check from protocol II, i.e. checks if $h(\mathcal{M}(D_e) \parallel l_e) \oplus last_i = \bigoplus_k \sigma_k$ for some i. For each epoch, a particular user can be assigned to do the synchronization check. For verification the user needs the initial state, $h(\mathcal{M}(D_e)||l_e)$. This it can compute from the $last_i$ values of the previous epoch.

Theorem 4.3 *Protocol III guarantees detection within two epochs assuming that every agent performs at least two operations every epoch.*

The proof of this theorem is omitted due to space constraints.

Unlike protocols I and II, in this protocol, all the users need not be online simultaneously. We note that this protocol guarantees that a fault by the server will be detected within two epochs. Hence, this protocol gives a guarantee

with respect to time, while the first two protocols guarantees with respect to the number of operations by a user.

5 Related Work

Hash trees were developed by Merkle and used for efficient authentication of a public file [7, 8] as well as a digital signature construction in [9]. Merkle's hash tree can be described as an authenticated dictionary data structure, allowing efficient proofs of membership or non-membership of elements in the set. Authenticated dictionaries were adapted and enhanced to manage certificate revocation lists in [5, 11]. Authenticated dictionaries were adapted to relations in [2], where algorithms based on Merkle trees and refinements in [11] are proposed for authenticating relations and verifying basic relational queries. They assume either the data is static or very infrequently updated. This is because they require the root of the Merkle tree published for every update. Extension to semistructured data was investigated in [1], and implementation using B-trees considered in [14]. Authenticating queries using the techniques above may require revealing some data items that are not in the query answer, or other information about the database instance [13]. It is the goal of [10, 12] to provide authenticated answers while also maintaining certain secrecy properties. Recently, techniques for proving the actual query execution for not only range queries but also more complicated compute-intensive data-mining queries were proposed [16].

6 Conclusions

We have studied the problem of implementing a CVS-like multi-user data management system on an untrusted server. We have formalized the notion of what it means for an untrusted system to "behave like" a trusted system by providing definitions of deviation detection and workload preservation. We have shown that external communication is necessary to guarantee bounded deviation detection if we are required to support workloads that are typical to CVS applications. We proposed efficient protocols that use external communication in the form of a broadcast channel.

Possible future directions are (1) to extend these protocol to detect exactly when the fault occurred (2) to find protocols where the clients do only constant amount of work as compared to proportional to the number of users in the system and (3) to address failures. In future work, we plan to extend our protocols to address these issues.

Acknowledgments. We thank Alan Demers and Jayavel Shanmugasundaram for helpful discussions. This work was supported by NSF Grants IIS-0330201, IIS-0133481, IIS-0121175, and by an E-Science grant and a gift from Microsoft Corporation. Any opinions, finding, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of xml documents. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 136–145. ACM Press, 2001.

[2] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security*, pages 101–112, 2000.

[3] Ronald Fagin, Joseph Y. Halpern, Moshe Y. Vardi, and Yoram Moses. *Reasoning about knowledge*. MIT Press, Cambridge, MA, USA, 1995.

[4] R. Housley, W. Ford, W. Polk, , and D. Solo. Internet x.509 public key infrastructure certificate and crl profile. 1999. IETF RFC2459, http://www.ietf.org/rfc/rfc2459.txt.

[5] Paul C. Kocher. On certificate revocation and validation. In *Financial Cryptography*, pages 172–177, 1998.

[6] Naor M and Nissim K. Certificate revocation and certificate update. Technical Report MCS99-05, Weizmann Institute of Science, 1999.

[7] Ralph C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, Information Systems Laboratory, Stanford University, 1979.

[8] Ralph C. Merkle. Protocols for public key cryptosystems. In *Symp. Security & Privacy*, pages 122–134, 1980.

[9] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[10] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *FOCS*, 2003.

[11] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *USENIX Security Symp.*, 1998.

[12] Rafail Ostrovsky, Charles Rackoff, and Adam Smith. Efficient consistency proofs for generalized queries on a committed database. In *ICALP*, 2004.

[13] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2005. ACM Press.

[14] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *ICDE*, 2004.

[15] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[16] Radu Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.