A Secure and Highly Available Distributed Store for Meeting Diverse Data Storage Needs *

Subramanian Lakshmanan Mustaque Ahamad H.Venkateswaran

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{subbu,mustaq,venkat}@cc.gatech.edu

Abstract

As computers become pervasive in environments like the home and community, data repositories that can maintain the long term state of applications will become increasingly important. Because of the greater reliance of people on such applications and the potentially sensitive nature of the data manipulated by them, the repository must be highly available and it should provide secure access to data. Furthermore, many different types of data, ranging from private data belonging to a single user to data shared across different users may be stored in the repository. We present the design of a distributed data repository, called a secure store, which can meet the data access needs of diverse applications. We develop protocols that replicate data at multiple servers to enhance availability and work even when a limited number of compromised servers exhibit arbitrary failure behavior. We also discuss how the nature of the data that is stored in the secure store impacts the availability and costs associated with data access.

1 Introduction

As computers become pervasive in the home and community, a variety of computation engines will be deployed to enable emerging applications in such environments. These computers will range from embedded processors to mobile and hand-held devices. For example, the Aware Home that has been built at the Georgia Institute of Technology [8] will support a variety of applications to assist its residents in their daily activities. Such applications in the home and others that are deployed across such connected homes in a community will need to store, access and share a variety of information. Several characteristics of computers that will execute such applications make them unsuitable for storing such information. First, they may be resource poor and may not be able to store long-term data. Second, they can be easily stolen or compromised and hence cannot be entrusted to maintain data that has confidentiality or integrity requirements (e.g., medical or financial records). Third, when data size becomes large, storage management is expensive and prone to errors. Ordinary home users are unlikely to deal with long-term management of such data.

The needs of the applications in environments like the Aware Home motivate a data repository service which is highly available and secure. In the Aware Home, one of the applications being explored enables older residents to stay in the home longer by providing a variety of services, including automatically connecting them with external medical facilities in the event of an emergency. Clearly, the information that is used to make such decisions must be highly available. Also, because of privacy concerns and sensitive nature of the information, access to it must be secured. We call the abstraction that provides such highly available and secure data repository service a *secure store*.

A secure store that can meet the needs of diverse applications will have to meet several requirements. First, it will have to replicate the information to make it highly available and to provide fast access to distributed clients. If the stored data may be updated dynamically, clients must access consistent values for data items. Finally, access to private or sensitive information needs to be controlled. In particular, confidential information must not be disclosed to unauthorized parties, and its integrity should not be compromised even when some of the server nodes become faulty.

Considerable work has been done in building distributed stores which range from distributed file systems to object stores. For example, the Bayou [5] system provides mobile clients access to information via replicated servers and it

*This work was supported in part by NSF grants CCR-9988212 and ITR-0081276.

supports several levels of consistency (called session guarantees) for the data. Such systems primarily focus on high availability (e.g., in Bayou a client can access information if it connects to a single server) and use replication and weak consistency guarantees to achieve this goal. More recently, security of replicated servers has been explored by systems such as Phalanx [1], Fleet [9] and others [3]. However, these systems assume general read/write sharing and strive to provide strong consistency guarantees which results in expensive protocols in wide-area environments.

The main contribution of the paper is the design of a secure replicated store to meet the needs of diverse applications. We present a set of protocols that enable clients to access information securely, with various consistency requirements. The store is implemented by a set of replicated servers, some of which could be compromised by an adversary. The compromised nodes could behave in arbitrary manner (e.g., Byzantine faults). The protocols exploit the nature of the data (e.g, non-shared or shared) and support weaker consistency levels to improve performance. By considering weaker consistency protocols, we demonstrate the cost vs. consistency tradeoffs that are possible. Applications can consider such tradeoffs and choose protocols depending on their consistency needs as well as the resources available to them.

Section 2 motivates our work by presenting applications that require secure and highly available access to different types of information with varying levels of consistency. Section 3 summarizes the considerable amount of related work that exists in the area. Section 4 defines our system model and lists the assumptions made by us in the development of the protocols. The detailed protocols are presented in Section 5. Section 6 discusses performance of the protocols. The paper is concluded in Section 7 with a discussion about future work.

2 Motivation

In this section, we consider three classes of applications that have security as well as consistency requirements for stored data. The traditional security requirements include confidentiality, integrity and availability. A variety of relaxed consistency models have been explored in the past that range from guarantees based on causality to ones that only require that reads of data items return values that are at least as recent as those read in the past.

1. First, we consider applications in which a single user accesses non-shared information. For example, a resident of the Aware Home may save medical records of family members or store tax information in the secure store. Such information is only accessed by the resident. Under certain conditions, copies may be made available to others (e.g., tax preparer or a medical facility) but the information is of non-shared nature.

2. In the second case, we consider applications that involve multiple users. For example, a school may create and store information which may be accessed by many families in the community. In this case, the information is written by a single user but it could be read by many others. This application falls in the general class of applications where a single source disseminates information that is of interest to many others.

3. Finally, we consider applications that manipulate data read and written by multiple users. Many collaborative applications that enable asynchronous interactions across users fall in this category. For example, a group of citizens may collectively develop a plan to address problems in the community over a period of time.

In the non-shared case, which is exemplified by the first class, the private nature of the data clearly implies that it is of confidential nature. The consistency requirement is straightforward because the updates are ordered and we only need to identify the most recent update. High availability for the data such as medical records is a must in an emergency situation. In the second class of applications where data is written by one user and read by others, integrity requirements can be easily seen. In particular, readers must be assured that the data they receive is from the legitimate writer and has not been modified by unauthorized parties. Strong consistency is not necessary. In particular, even if a reader may not see the latest value in the current read operation, it is sufficient that future reads return successively more recent data. Application level information can be used to discern when information may be stale.

For the last application class, where shared data items may be updated and read by multiple users, both confidentiality and integrity could be important and access to the information must be controlled. Consistency based on causality may be sufficient when concurrent updates to shared data items are not common or when such updates can be merged. Strong consistency, enforced by some type of synchronization may be unnecessary because users may externally coordinate their updates.

Although we have given example applications that come from the home and community domain, the secure store will be valuable in addressing the needs of applications from other domains which share similar security and consistency needs. Examples of such applications include access to personal information such as email, inventory management across distributed locations, collaborative software development projects, and educational and entertainment applications.

3 Related Work

To build a highly available and secure store, data is usually replicated at several servers. Consistency across multiple replicated copies of data items needs to be maintained in such an environment. Traditionally, it is assumed that replication should be transparent and the consistency provided should ensure the same results that can be obtained from a single, non-replicated copy. Such strong consistency (e.g., one-copy serializability, sequential consistency or linearizability [10]) could result in expensive protocols that cannot scale and is not needed by many applications. We primarily consider applications where weaker notions of consistency suffice. Thus, we discuss related work that addresses weak consistency. There has also been considerable work that deals with malicious servers. We also discuss protocols that tolerate malicious servers in replicated data systems.

Systems that range from distributed file systems to shared memory systems have explored weaker consistency models to enhance performance and scalability. For example, a consistency model based on causality can be defined when shared objects are read and written by distributed users. The causal consistency model [11] permits more efficient implementations and can meet the needs of many applications. Although weaker consistency models may be appropriate for some applications, no single model may meet the consistency needs of all applications. Thus, several consistency levels may have to be provided in the same system [5, 17]. The Bayou system best exemplified this approach. It makes use of replicated servers to provide access to mobile clients. To provide efficient access, Bayou allows a client to read or write a data item from a single server. Once a data item is written at one server, the server can propagate its value to other servers using an asynchronous dissemination scheme. Our approach of supporting multiple levels of consistency is motivated by Bayou-like systems. However, we differ from such systems in several respects. We treat servers as passive data repositories. The burden of maintaining consistency is upon clients that must save and use meta-data to decide what values can be accessed without violating consistency. This approach allows us to limit the power entrusted to servers which is useful when they exhibit malicious behavior. Although malicious server behavior was not addressed by Bayou when the project started, a follow-up paper did address this problem [6]. However, because of Bayou's goal of allowing reads and writes with a single server, this paper explored only safety related concerns (e.g., data corrupted by a malicious server is not accepted by a client) and denial of service due to malicious servers was not addressed. The authors did propose logging and auditing of writes and reads to detect and rectify damage done by malicious servers.

There are two separate approaches that have been explored for replicated servers. In the state machine approach [4], all replicated servers receive and process requests in the same order. A request can be completed as long as certain number of replicas can respond to the request. Schneider's survey paper discusses strategies for tolerating both crash and Byzantine failures [4]. Castro and Liskov [3] gave a practical implementation based on this approach. The algorithm uses message authentication codes and multi-phase protocols to ensure linearizability. Although this implementation is shown to be efficient in the common case when clients and servers have high bandwidth connectivity, the performance of such protocols could suffer in widely distributed environments.

The quorum approach to replicated data management requires that read and write operations be completed with sets of servers. Such sets, called quorums, are chosen to overlap with each other. This ensures that reads return the value of the most recent write and that write operations are ordered. Most of the quorum based protocols only dealt with crash failures. More recently, Reiter and others have considered malicious servers and have explored what are called Byzantine quorums [12]. The Phalanx and Fleet systems [1, 9] are designed using such quorum protocols. Byzantine quorums require more common members than traditional quorums. For example, to tolerate b malicious server failures, at least $2b + 1$ members must be common between any two quorums. These quorums provide strong consistency (safe semantics for concurrent reads and writes) and do not address more flexible consistency levels. Although performance of dissemination protocols is explored by the authors in [13], they are not specific to any consistency level. Consistency levels such as causal consistency, coupled with assumptions about how data is shared, could allow much smaller quorums that could yield improved performance for read and write operations. Another approach that attempts to reduce quorum size makes use of techniques to estimate the number of malicious servers [2]. Thus, the quorum size is dynamically adjusted based on the number of servers that are believed to be faulty at a given time.

Our implementation of a secure and highly available store could benefit from several other techniques that we are not using in this paper. For example, Fray et. al. [18] propose a scheme that fragments the information in a data item and stores it at several servers. In this case, if fewer than a threshold number of servers are compromised, the data item's value cannot be reconstructed and hence cannot be disclosed. Others have also explored such fragmentation based schemes [14, 15].

4 System Model and Assumptions

To meet availability goals in the presence of one or more compromised servers, the secure store is implemented by a set of n replicated servers $S_1, S_2..S_n$. Together, these servers offer a data repository service. The secure store is primarily responsible for safe keeping of data written to it and to provide values according to consistency requirements of clients when reads are requested. Other issues such as authorization or who should be able to access a given data item are addressed by additional services.

All requests from clients are authenticated. Hence, we assume the availability of necessary authentication and cryptographic mechanisms. All communication channels between clients and servers are assumed to be secure against eavesdropping, modification and replay attacks. Also, clients and servers are assumed to own a secure private key for which the public key is well known. Key management issues are not addressed in this paper.

We assume a secure authorization mechanism in place. A non-faulty server does not accept a write or a read request from an unauthorized client. This can be effected by using authorization tokens issued to clients by some secure authorization service. We ensure confidentiality of certain kind of data (e.g., data exclusively accessed by a user) but meta-data associated with a data item that is used for consistency maintenance is not confidential.

We assume an upper bound, denoted by b, on the number of servers that can fail or be compromised at any time. Failures could be either crash failures or Byzantine, and faulty servers can behave arbitrarily while executing the secure store protocols. Thus, a compromised server can mount an attack, possibly colluding with other faulty nodes. We assume that during the period of a client's read or write, the system is fairly stable. That is, the set of faulty servers remains within the assumed bound and does not change arbitrarily.

A server maintains meta-data about each data item it stores in addition to its value. The meta-data associated with the copy of a data item at a server depends on the consistency required by the clients. This meta data includes the unique identifier of the writer, timestamp associated with the current value, consistency-related information called $context$ of the writer (defined in a later section), signature of the writer and other relevant information.

Since some servers may not be available at the time a request is made, we cannot assume that all servers must be available to handle a read or write request. It is also desirable to complete a request by contacting a small number of servers from a performance point of view (e.g., response time perceived by a client). Thus, clients may complete updates after communicating with a small subset of servers. We assume that servers keep themselves informed about updates in which they do not directly participate via a gossip or dissemination protocol [7]. A non-faulty server transmits all the updates it has seen to at least one other non-faulty server. A faulty server cannot propagate a non-existent or forged write to other servers since all writes that are propagated have to be accompanied by the signature of the client who wrote the value. We do not make any assumption about the specifics of the gossip protocol and require only that it allows non-faulty nodes to exchange information about their updates.

Although the secure store may contain a large number of data items, we assume that each item belongs to relatively small group of related data items. For example, documents pertaining to a certain topic may define the group of related data items. We assume that consistency is only required across a group of related data items and not across data items that come from different groups. To access the secure store, clients establish a session with or connect to the secure store before they issue read and write requests. A user may have previously interacted with the store and he or she must see the effects of requests issued in earlier sessions. We associate a $context$ with a client to capture its interaction with the store. The $context$ reflects reads or writes that were completed by the client previously. Such $context$ itself could be stored in and retrieved from the secure store.

4.1 Notation

We use the following notation in the description of our protocols. S_i denotes server i, C_i denotes client i and x_i denotes the name of the data item i. Each data item has a unique identifier in the system, denoted by $uid(x_i)$. X denotes a set of data items. v is a value written to or read from a data item. ts denotes a timestamp. One or more timestamps are associated with a value created by a write. A timestamp serves as a unique identifier for a write. In protocols for non-shared and single-writer applications, a timestamp is simply a version number. For the multi-writer applications, timestamps also contain the writers' unique identifier and the digest of the value being written. We use K_i and K_i^{-1} to denote the public and private key of client C_i. $\{data\}_{K_i^{-1}}$ denotes signed digest of data using private key of client i. $d(v)$ denotes the digest of a value v using some agreed-upon digest algorithm. A set of identifiers and timestamps $((uid(x_1), ts_1), ..., (uid(x_m), ts_m))$, called the $context$, is maintained by each client locally for data items in group X. We denote it by $\mathcal{X}$. A $context$ may also appear in the meta-data associated with a value of a data item. We use $\mathcal{X}_{writer}$ to denote the $context$ of the writer at the time the value was written.

4.2 Consistency models

We advocated that the secure store support access to a variety of information with different levels of consistency for shared information. We consider the following consistency models that can meet the needs of many of the applications described earlier.

1. **Monotonic Read Consistency (MRC):**A client who accesses data items using this level of consistency always sees an increasing set of writes to a an object as time proceeds. More specifically, if a client reads a value v for a data item x_i, at a later time when it reads data item x_i again, it is returned v or a value which is newer than v. A value v' is said to be newer than v if the store orders the write that produced v' after the write which stored v. For non-shared data that is accessed by a single client, MRC implies that the client will access the most recent copies of its data items. For shared data, future reads of a reader could return more recent values but are not guaranteed to return the latest value of the object. Thus, consistency guarantees provided by MRC are similar to the monotonic-reads and read-your-writes session guarantees in Bayou.

2. **Causal Consistency (CC):** MRC only ensures that for a given data item, a client never receives values older than the ones read in the past. It does not impose restrictions across related data items. Consider the case when a client writes value v_2 to a data item x_2 based on value v_1 of data item x_1 that it has read. Informally, if another client reads value v_2 for x_2, CC ensures that the client is guaranteed not to read a value for x_1 that is older than v_1. The notion of "older" is precisely defined based on the happens before order for read and write operations to shared data items [11]. This relation can order read and write operations across a set of related data items. In particular, if o_1 and o_2 are two writes to data item x which assign values v_1 and v_2 to x respectively, v_1 is said to be causally overwritten by v_2, if o_1 causally precedes o_2. A secure server that supports CC ensures that no read operation returns a causally overwritten value.

Although we focus on protocols for MRC and CC, clearly these protocols may not be able to meet the consistency needs of all applications. In particular, some applications may require strong consistency. In this case, existing protocols can be used. For example, the replicated state machine approach based protocol in [3] can be used to ensure that all client operations appear to execute in a total order. MRC and CC do not address how quickly values written by a certain client become available to others. Although models that address timeliness do exist, their implementation in an asynchronous distributed system is infeasible. We do assume that MRC and CC will return newer values eventually when clients continue to read the data.

5 The Protocols

We consider data that is accessed by the three classes of applications we listed in Section 2. We first present protocols for the first two classes of applications, namely non-shared and single-writer applications. In these applications, since a single client is responsible for writing the data, we assume that the client is honest and follows the protocol as expected without trying to compromise the integrity of the service provided by the store. For the third class of applications which allow data items to be updated by multiple clients, if there are no malicious clients, the same protocol works by extending the timestamp to include the uid of the writer. However, the protocol needs to be modified to handle malicious clients.

The secure store not only provides safe storage for long-term state but also makes data highly available to the clients. To determine what values are consistent for a given client, we associate a *context* $\mathcal{X}_i$ with client C_i that captures its past interactions with the store. Such $context$ itself can be stored in the secure store and read when a client starts a session. We say that a client *connects* with the store to initialize its $context$. Before going off line, by executing a *disconnect* operation, the client writes back its updated $context$. Writing and reading of $context$ require special treatment, irrespective of the consistency level required by the application. We first discuss how $context$ is represented and how clients read and write it. This is followed by the protocols that are used to read and write other data.

5.1 Context and its management

In replicated data systems, since all copies may not be identical (an update writes new values only at a subset of the servers), version numbers or timestamps are used to determine which copies have the latest values. Such timestamps must monotonically increase as updates are done. They can be read either from clocks that advance between successive updates or can be read from logical clocks that are advanced as read and write operations are executed. When consistency needs to be ensured across a set of related data items and a client accesses several such items, timestamps have to be kept for each of the data items. The $context$ of a client includes the unique identifiers of data items accessed in a given session and the timestamps associated with the data items. Although the secure server may potentially store a large number of data items, in a given session, we assume that a client only accesses a small number of such items. This implies that the $context$ maintained by a client at a given time will not be large.

Consistency dependencies in the CC model can arise because of operations that are executed on several data items that belong to a related group. For example, a client C_i may write value v to data item x based on the value v' of another data item x' that it read in the past. Another client that reads v for x and then requests a copy of x' must see either v' or a more recent value of x'. To ensure this, we need to associate meta-data with values of data items that are stored at the servers. Such meta-data for value v must reflect the fact that it potentially has a causal dependency on the write that produced value v' of item x'. In particular, the meta-data stored with v not only has the timestamp of x but also the timestamps of other related data items to capture causally preceding updates to them. In fact, the set of timestamp values of the data items that are related to x and x' is precisely the information that is needed to maintain CC. This information is captured by a client's $context$. Thus, if $X = \{x_1, x_2, ..x_m\}$ is a related group of data items, the $context$ associated with data items in X at client C_i is $\mathcal{X}_i$ = $((uid(x_1), ts_1), ..., (uid(x_m), ts_m))$. As discussed later, the $context$ evolves as ts_j's increase when reads and writes requested by C_i complete. If an x_i is written back to the store, the client's $context$ associated with X is written with the data value. Timestamp vectors similar to $context$ defined by us are also used in many systems that have been developed for weakly consistent replicated data [5].

A client's $context$ reflects the accesses it has completed. Thus, when a new session is initiated by the client, it must initialize its $context$ to reflect the interactions it had completed in the last session. A client may deal with a number of $context$ objects over a period of time. Clients may be resource poor and information stored at the client site could be compromised easily. Hence, we choose the approach in which a client saves its $context$ in the secure store along with a signed digest of the $context$. The signature ensures that a malicious server cannot alter the $context$ information.

To ensure that a read of the $context$ returns its latest value, strong consistency needs to be guaranteed for the read and write operations that access $context$. To ensure such consistency and availability in the presence of up to b faulty servers out of a total of n, we use a quorum based approach when $context$ is read or written. In particular, we require that the reading or writing of $context$ information be done with at least $\lceil (n+b+1)/2 \rceil$ servers. This ensures that at least one non-faulty server, to which the last $context$ information was written, will participate in $context$ read. The client can choose the most recent $context$ value that has a valid signature from the values returned by the servers in the quorum.

```
Read $C_i$'s $context$ on session initiation:
    let $X$ be the related group of data items
    that $C_i$ wants to access in the session;
    request $C_i$'s $context$ associated with $X$ and signature
    from all servers;
    wait for at least $\lceil (n+b+1)/2 \rceil$ responses;
    check if a $context$ is valid by verifying its signature;
    $\mathcal{X}_i$ = latest valid $context$ returned by some server;

Store $C_i$'s $context$ on completion of its session:
    let $\mathcal{X}_i$ be $C_i$'s current $context$ for data items in $X$ ;
    send $\{\mathcal{X}_i, \{\mathcal{X}_i\}_{K_i^{-1}}\}$ to $\lceil (n+b+1)/2 \rceil$ servers;
```

Figure 1. Context acquisition and storage

As can be seen in Figure 1, $context$ read or write can be completed as long as $\lceil (n+b+1)/2 \rceil$ servers participate in the quorum. Since a valid signature is required, faulty servers can only misbehave by either not responding or sending an old value of the $context$. Given that the latest value received from a server is chosen as the client's $context$, the $context$ from a non-faulty server that participated in the most recent write will be chosen. Here, latest value is that vector which has the highest timestamp for every data item in the group. Notice that we require only $b+1$ servers in the intersection of two quorums whereas masking quorums require $2b+1$ servers in the intersection. Our optimization is possible because we can choose the latest valid $context$ from a single server while masking quorums require that a value appear in the response of at least $b+1$ servers for it to be chosen.

If a client successfully writes its $context$ prior to session termination, at least one of the servers that responds to the next $context$ acquisition request will return the client's latest $context$. If the client fails either before this is done or while it is writing the $context$, the quorum intersection is not guaranteed to return the most recent $context$. In fact, the context stored in the meta-data of data items could be more recent than the $context$ written by the last successfully terminated session. In this case, a more expensive protocol is used to reconstruct the context. The client will have to read the timestamps associated with all data items in a group X for which $context$ needs to be reconstructed. These items must be read from all servers. Only the faulty servers may choose not to respond to the client request. From these values, the latest valid timestamp for each data item is used to reconstruct the client's $context$ for data items in X.

5.2 Protocols for non-shared and single-writer applications

Once a session is started and a client initializes its $context$, it could issue read and write operations for the data items that it is authorized to access. The client is responsible for accessing consistent data based on the $context$. Similar to the protocol in Figure 1, we assume that the operations are executed by client C_i. Since this section considers data that is written by a single client, C_i is the only one that executes write operation on the data items. Other clients can read shared data that is written by C_i. We assume that for a given set of data items, either MRC or CC consistency is specified at the time of their creation. Thus, the same data item cannot be accessed with MRC consistency requirement at one time and CC consistency at another time.

```
let $\mathcal{X}_i = ((uid(x_1), t_1), ....., (uid(x_m), t_m))$;
Write($x_j, v$):
    increment $t_j$ in $\mathcal{X}_i$ to current clock value;
    if CC is required then
        write-message := {"write",$uid(x_j)$,
        $\mathcal{X}_i$, $v$, $\{uid(x_j), \mathcal{X}_i, v\}_{K_i^{-1}}$ };
    elseif MRC is required then
        write-message := {"write",$uid(x_j)$, $t_j$, $v$,
        $\{uid(x_j), t_j, v\}_{K_i^{-1}}$ };
    endif;
    send write-message to at least $b + 1$ servers;

Read($x_j$):
    let $t_j$ be the timestamp associated with $x_j$ in $\mathcal{X}_i$;
    send ($uid(x_j)$,$t_j$) to $b + 1$ or more servers;
    receive replies from these servers that includes
    the meta-data of $x_j$ ;
    let $t_r$ be the highest timestamp for data item $x_j$
    among the replies ;
    if $t_r \geq t_j$ then
        choose the server which sent $t_r$ in its reply;
        send {"read", $uid(x_j), t_r$} to chosen server;
        receive $M = \{t_r, \mathcal{X}_{writer}, v,$
        $\{uid(x_j), t_r, \mathcal{X}_{writer}, v\}_{K_i^{-1}}$ }
        accept $v$ if the signature is valid;
        if MRC consistency is required then
            update $t_j$ in $\mathcal{X}_i$ to $t_r$ when $t_r > t_j$;
        if CC consistency required
            update each timestamp in $\mathcal{X}_i$ to max of value
            in $\mathcal{X}_i$ and the corresponding value in $\mathcal{X}_{writer}$;
    else
        contact additional servers or try later
```

Figure 2. Read and write protocols executed by client C_i

Figure 2 shows the protocols for reading and writing when the data is written by a single client. For monotonic read consistency, only the current version number or timestamp of the data item is stored with its value. Since the timestamp of this data item monotonically increases as values are read and written, successive reads of a client will return newer values. Since a client writes its *context* at the end of a session, a future session will also return the most recent value seen by the client or a newer value.

Since servers simply act as passive stores for signed information, a faulty server cannot modify either the meta-data or the value of the data item in an undetectable way. Thus, it can either not respond to a request, or respond with old data or data that is corrupted. A client can detect old or corrupted data by verifying the signature and examining the associated meta-data. By writing the data to at least $b + 1$ servers, we ensure that at least one non-faulty server receives the data and will store it correctly. However, such a non-faulty server to which the last data value was written may not be among the $b + 1$ servers that are contacted by a subsequent read operation. In this case, the data supplied by the non-faulty server may be stale according to the *context* of the requesting client and it will not be accepted. To increase the likelihood that a client's read request does include a non-faulty server with current value of the data item, we add a dissemination component to the protocol presented so far. Many dissemination protocols exist [7, 13] that allow servers to exchange data values. New data values could be sent to one or more servers at a frequency that can be tuned according to the needs of the clients or the resources available to the servers. A server receiving such a message can verify that the update was signed by an authorized client. The server receiving the message updates its value if the timestamp of the received value is greater than the timestamp of the data value stored at the receiver. We omit the details of the dissemination protocol but these details and a correctness argument can easily be constructed based on implementations that exist for systems that support shared objects that provide causal consistency [11].

The protocol presented in Figure 2 may not return a value with a timestamp that is greater or equal to the data item's timestamp in the client's *context*. Several options exist for handling this case. For example, additional servers may be contacted or the client can try the operation at a later time when the new value may have been disseminated to the servers that it contacted. Thus, the cost of a read operation will depend on the dissemination protocol as well as the frequency with which data items are updated.

The correctness of the protocol follows from two observations. First, no malicious server can modify any data item since all data items are signed. Second, consistency is enforced by the client accessing the data. Since a single client writes the data (both non-shared and shared data that others only read) and the writer monotonically increases the timestamps on updates, timestamps in client *context*s or in the meta-data stored with object values can always be ordered. In the write protocol, the meta-data is included for computing the signature and non-faulty servers forward the entire write message. Thus, a malicious server can neither successfully disseminate spurious data values nor can it change the meta-data associated with values.

Although the protocol discussed does not address confidentiality, it is easy to provide for the data values in the non-shared case. The owner or writing client can store all its data items in encrypted form. When the owner changes its key, it reads the data items, re-encrypts and stores them back. The meta-data associated with a data item, however, must be in plaintext because servers must use it to update their values when the dissemination protocol is run. A malicious server can thus disclose the meta-data associated with a data item but not its value. Since the timestamp only needs to increase monotonically, the writer can increase it on each write by some random amount. That will ensure that others cannot guess how many times the data item has been updated. For the single writer and multiple readers case, confidentiality can be ensured using a similar scheme. However, the key that is used to encrypt the data values must be distributed to readers, perhaps along with the authorization credentials that are needed to access a data item. Servers do not know this key and hence, malicious servers cannot disclose any information to unauthorized clients. If there is a change in the set of clients that has access to the data, key distribution and management schemes similar to those discussed in secure multicast communication [16] have to be employed.

5.3 Protocols for multi-writer applications

We now consider the case when shared data items are both read and written by multiple clients. Because we only provide MRC and CC consistency, if the writers generate ordered timestamps for their updates, the protocol in Figure 2 will still be correct since the timestamps stored with data values will define an order that will be consistent with causality. However, because the writers can generate values independently, ordered timestamps cannot be guaranteed. This could create several problems for the earlier protocol. First, without any coordination between writers, two distinct values v_1 and v_2 for data item x could have the same timestamp. In the protocol in Figure 2, it is possible for reads of x by a client to return v_1 followed by v_2 and then v_1 again. This is clearly undesirable. Servers that participate in the dissemination protocols can also get confused when distinct values of x have the same timestamps.

The protocol in Figure 2 can be adapted for multiple writer case by changing how timestamps are associated with data items. In particular, with a time t, we also include the unique identifier of the client that generated t to create the timestamp. A malicious client cannot use the timestamp of a different client. This is because the signature associated with writes includes timestamps as well, and the key used to sign should match the uid of the client in the timestamp. To prevent a malicious client from using one timestamp for two different values it writes, we also include the digest of the value written in the timestamp. Thus, a timestamp becomes a 3-tuple ($time$,$uid(C_i)$,$d(v)$). Two timestamps $ts_1 = (t_1, uid_1, d_1)$ and $ts_2 = (t_2, uid_2, d_2)$ are first ordered based on the value of the time associated with the timestamp. If t_1 and t_2 are the same, the timestamps are ordered based on the uids of clients contained in them. If the uids are same as well, the digests should be the same. Otherwise, the writer is deemed to be faulty. In this case, clients accessing this data item can be informed that the value cannot be assumed to be correct. The augmented timestamps are associated with data values and the signed digest reflects both the timestamp as well as the value.

It should be noted that malicious clients can write garbage values. This cannot be prevented and must be detected at the application level. However, a malicious client C_1 could include spurious entries in a context as part of a write. These entries could be arbitrarily high and any client C_2 which reads this write would update its local context with such high timestamps. These timestamps may not correspond to any valid write residing in any of the servers. Thus client C_2 would look for values with greater timestamps and would not find such values. This could have a cascading effect if C_2 writes values with this corrupted context. Soon the whole set of clients would see this easy denial of service attack. Clearly such a situation is not desirable.

Hence we make the following modifications to our earlier protocol. A non-malicious server should start reporting a write to any requesting client only after the causally preceding writes, as reflected in the accompanying context, arrive at the server. A client that does a read should make read requests to 2b+1 servers and accept a value as valid only if b+1 or more servers reply with the same value. This is to mask the effect of malicious servers that may report a value even before the causally preceding writes have arrived. Finally, we require that non-malicious servers log the writes and report a set of latest writes for a particular data item so that a client can choose a common value from b+1 lists. This makes sure that a value being over-written is still available while the new value is being disseminated to atleast b+1 non-malicious servers. Old values could be erased from the log once a server learns that a new value is available at atleast 2b+1 servers.

Confidentiality can be achieved as in the case of single writer case by sharing a key that servers do not know. The writers must coordinate with each other to acquire this key and if the key is changed, it must be made available to all writers after the data stored at the servers is encrypted again with the new key. Malicious servers might still retain the old data, encrypted with the old key. If the old key is compromised, confidentiality is lost. We are currently exploring methods to address such confidentiality issues.

6 Performance

The performance of a secure store can be characterized based on a number of metrics. First, when a client needs to read or write data items, the communication necessary with server nodes requires network bandwidth and impacts the response time of the client operation. The response time could vary with the number of servers that need to participate in the execution of the operation. Second, cryptographic operations such as generation of digests and signing them could introduce computational overhead. In this Section, we discuss both message and computational costs of our protocols that affect the latency seen by the client while executing read and write operations and compare them with those of some of the protocols described in Section 3.

We distinguish operations that are used to acquire and store $context$ data from those to read and write other data items. The quorum sizes are different in these two cases. The size of $context$ data depends on two factors: (1) the number of related data items in a given group, and (2) the number of groups from which data items are accessed in a session. For example, all documents containing tax related information for a given year could be considered related. An application may access tax documents as well as documents that store information about medical bills. Once the size of $context$ data is determined, its acquisition requires round-trip communication with $\lceil (n + b + 1)/2 \rceil$ servers, where n is the total number of servers and b is the constant bound on the number of servers that could be faulty. The $context$ data also needs to be stored at the same number of servers when it is written on session termination. Thus, the message costs and the response time of $context$ read and write operations depends both on the total number of servers and the number of servers that could exhibit malicious behavior. In particular, a total of $2 * \lceil (n + b + 1)/2 \rceil$ messages will be exchanged between the client and the servers to retrieve or store the $context$.

Prior to storing the $context$, its digest must be computed and signed with the client's private key. Non-faulty servers need to verify the signature to ensure that they do not overwrite their $context$ data with spurious information. Thus, the computational overhead at the time $context$ is written includes one signature and $\lceil (n + b + 1)/2 \rceil$ signature verifications. When $context$ is read, the latest $context$ is chosen and only its signature needs to be verified. Thus, in the best case, $context$ acquisition requires just one signature verification. It should be noted that $context$ is accessed only on session creation and termination.

The cost of read and write operations for non-context data depends on both the quorum size as well as on the rate at which new values are propagated among servers. A write operation can complete for all types of data (non-shared, shared with MRC or CC consistency) by communicating with $b + 1$ servers. This gives a total of $b + 1$ messages for write operation. Since the operations can be completed by communicating with only $b + 1$ servers, their response time will be better than the response time of $context$ operations. In the best case, the message cost and response time of read operations could also be the same as write operations. This will be the case when one of the server that responds to the read quorum request has copy of the data item that is consistent according to the client's $context$. However, if the desired data value has not been propagated to the servers in the read quorum, either additional servers must be contacted or read must block until the needed data value is disseminated to one of the servers in the quorum. The dissemination protocol would require additional communication between the servers. The frequency of such communication will depend on the resources available to servers as well as the read response time desired by the applications.

Similar to $context$ information, servers store signed data items. Thus, each write requires the signing of the digest of the value and the meta data being written by the writer. Servers in the quorum must verify such signature. For reads, signature verification is done by the client. Thus, the computational overhead of writes includes $b+1$ signature verification and additional overhead of signing the data value by the client that writes the value. For a read operation, a client needs to verify only the signature of the final value it accepts. Since b will be much smaller than n, the overhead of signing and signature verification will be significantly lower than other quorum based protocols.

The figures in the preceding paragraphs change from $b + 1$ to $2b + 1$ for the malicious clients case. Clients do not have to do signature verification for a read now since non-malicious servers do the validation before reporting. However, a data value being written is not available for the clients until the write and the causally preceeding writes reach atleast b+1 non-malicious servers. The malicious clients case requires additional overhead at the server side to maintain a history of a limited number of writes for each data item.

Both the communication and computational overheads of our protocols are better than the quorum protocols that provide strong consistency. For example, if majority quorums are used, Byzantine quorums require communication with $\lceil (n + 2b + 1)/2 \rceil$ servers for both read and write operations. Although improved quorum design can reduce their sizes [12], a minimum quorum size of $\sqrt{n}$ is necessary. We can achieve lower overhead even for $context$ operations by using an improved quorum design. For non-context data, the message cost of writes is much lower and when writes are infrequent, most reads will access data that has been disseminated to all servers. In this case, the average cost of reads will be close to the costs of writes. Similar to message overheads, the computational overheads of strong consistency quorums include signature verifications that are proportional to the size of the quorums. Thus, by providing weaker consistency when appropriate, significant communication and computational savings can be realized.

The state machine based implementation of Byzantine fault-tolerance reported in [3] provides better performance than earlier systems. This is primarily due to lower computational overheads of message authentication codes that are used in place of signatures. Although this approach offers computational savings, it has significantly higher message overhead. For example, the multi-phase protocols require $O(n^2)$ messages. This could lead to high response time for operations, specially in an environment where communication latencies are high across the server replicas.

In our future work, we plan to explore the performance of our protocols using both simulations as well as actual implementations. However, it is clear that flexible consistency levels, which are appropriate for many applications, can offer significant performance improvements.

7 Conclusions

The pervasive nature of computers, particularly in the home and community environment, will lead to new applications that will create and manipulate sensitive information about users and the environments in which they work and live. Secure storage of such information will be a key requirement. Furthermore, because of increased reliance by humans on services provided by such applications, the information that they need must be highly available. Although consistency requirements will exist, their data access and sharing needs will differ from traditional applications. We propose the design of a secure repository to store and access such data that has a variety of sharing patterns.

We propose an approach in which servers are primarily repositories of data, and clients are responsible for accessing consistent values of data items. Our primary contribution is the integration of techniques that allow the secure store to tolerate malicious behavior of some compromised servers with consistency models that provide weaker but acceptable guarantees for several classes of applications. This results in more efficient protocols, leading to improved access times for read and write operations.

We are currently working on a large group project that is addressing a variety of applications in the home and community. The secure store will be implemented to maintain information that will be generated and manipulated by these applications. We plan to build our store using the protocols discussed in this paper. Such an implementation will allow us to measure many of the performance improvements that are possible when only weaker consistency, as permitted by the sharing patterns of the applications, is provided instead of strong consistency. We will also explore secure applications that benefit from such a store in the future.

References

[1] Malkhi D., Reiter M., “Secure and Scalable Replication in Phalanx”, *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[2] Alvisi L., Malkhi D., Pierce E., Reiter M., Wright R., “Dynamic Byzantine Quorum Systems”, *Proc. International Conference on Dependable Systems and Networks*, June 2000.

[3] Castro M., Liskov B., “Practical Byzantine Fault Tolerance”, *Proc. 3rd Symposium on Operating Systems Design and Implementation, New Orleans*, February 1999.

[4] Schneider F., “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”, *ACM Computing Surveys*, Vol. 22, No. 4, December 1990.

[5] Terry D., Demers A., Peterson K., Spreitzer J., Theimer M., Welch B., “Session Guarantees for Weakly Consistent Replicated Data”, *Proc. International Conference on Parallel and Distributed Information Systems*, Austin, Texas, September 1994.

[6] Spreitzer J., Theimer M., Petersen K., Demers A., and Terry D., “Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems”, *Proc. 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.

[7] Demers A., Greene D., Hauser C., Irish W., Larson J., Shenker S., Sturgis H., Swinehart D, and Terry D., “Epidemic algorithms for replicated database maintenance”, *Proc. 6th Symposium on Principles of Distributed Computing*, pp. 1-12,1987.

[8] The Aware Home Research Initiative, http://www.cc.gatech.edu/fce/ahri/, 2000.

[9] Scalable and Survivable Data Replicatiion : The Fleet Project, http://www.bell-labs.com/user/reiter/fleet/.

[10] Herlihy M. and Wing J., “Linearizability: A Correctness Condition for Concurrent Objects”, *ACM Transactions on Programming Languages*,12(3), July 1992.

[11] Ahamad M., Neiger G., Burns J., Hutto P., Kohli P., “Causal Memory: Definitions, Implementations and Programming”, *Distributed Computing journal*, Springer-Verlag, Aug. 1995.

[12] Malkhi D., Reiter M., “Byzantine quorum systems”, *Proc. 29th ACM Symposium on Theory of Computing*, May 1997.

[13] Malkhi D., Mansour Y., and Reiter M., “On diffusing updates in a Byzantine environment”, *Proc. 18th IEEE Symposium on Reliable Distributed Systems*, October 1999.

[14] Rabin M., “Efficient dispersal of information for security, load balancing and fault tolerance, *Journal of the ACM*,36(2):335-348, 1989.

[15] Alon N., Kaplan H., Krivelevich M., Malkhi D., Stern J., “Scalable Secure Storage when Half the System is Faulty”, *Proc. 27th International Colloquium on Automata, Languages and Programming*, 2000.

[16] Wong C., Gouda M., and Lam S., “Secure group communication using key graphs”. *ACM SIGCOMM’98*, September 1998.

[17] Yu H. and Vahdat A., “Design and Evaluation of a Continuous Consistency Model for Replicated Services”, *Proc. Operating Systems Design and Implementation*, October 2000.

[18] Fray JM., Deswarte Y. and Powell D., “Intrusion-tolerance using fine-grain fragmentation-scattering”, *Proc. 1986 IEEE Symposium on Security and Privacy*, Oakland (CA), April 1986.

[19] Lakshmanan S., Ahamad M., H. Venkateswaran, “A Secure and Highly Available Distributed Store for Meeting Diverse Data Storage Needs”, Tech. Report GIT-CC-00-41, Georgia Institute of Technology, March 2001.