Secure and Scalable Replication in Phalanx

(Extended Abstract)*

Dahlia Malkhi Michael K. Reiter

AT&T Labs Research, Florham Park, NJ, USA
{dalia,reiter}@research.att.com

Abstract

Phalanx is a software system for building a persistent, survivable data repository that supports shared data abstractions (e.g., variables, mutual exclusion) for clients. Phalanx implements data abstractions that ensure useful properties without trusting the servers supporting these abstractions or the clients accessing them, i.e., Phalanx can survive even the arbitrarily malicious corruption of clients and (some number of) servers. At the core of the system are survivable replication techniques that enable efficient scaling to hundreds of Phalanx servers. In this paper we describe the implementation of some of the data abstractions provided by Phalanx, discuss their ability to scale to large systems, and describe an example application.

1. Introduction

In this paper we introduce *Phalanx*, a software system for building persistent services that support shared data abstractions, such as variables and mutual exclusion, for clients. The properties that distinguish Phalanx from other systems that provide such services are its *scalability* and its *survivability*: Phalanx is designed to scale to hundreds of servers spread across a wide area and to be capable of serving thousands of clients at a time. Moreover, Phalanx can survive the arbitrarily malicious corruption of up to a threshold of its servers and any number of clients while still providing useful services.

The applications at which Phalanx is targeted are large-scale distributed applications that have a need for shared state with intrinsic survivability and security requirements. Applications in this category include, for example:

1. **Public key infrastructures**: Common to many proposals for public key infrastructures (PKIs) are on-line services that support critical functions. These functions may include certificate-generating services that create certificates (i.e., bindings associating attributes to public keys) as in [19, 27], revocation services that enable a client to promptly invalidate her certificate as in [8, 16], and directory services that enable a client to locate the most up-to-date certificate for a name or key, such as X.509 directories [10]. A PKI is a prime example of a system that may need both to survive hostile attempts to penetrate it due to the security requirements it embodies, and to scale to worldwide proportions.
2. **Robust publishing and dissemination**: The Eternity service [1] is a proposed service that would enable a client to publish a document so that anyone can retrieve it, but so that nobody—even the author, or an adversary with the means to mount a military strike against the service—could eliminate the document from existence or otherwise deny access to it. Such a service will inherently require massive replication over a wide area that can survive attempts to corrupt the data it holds. The Eternity service is one (ambitious) example of a broader class of robust publishing and dissemination services that Phalanx is designed to support.
3. **National voting systems**: The AT&T Secure Systems Research Department (of which we are members) was recently tasked with designing an electronic voting system for Costa Rica's national elections. Among the goals of this design was to enable each voter to vote from any of over 1000 voting stations in the country, while still ensuring that no voter identifier could be used to vote multiple times. This application is one instance in which robust mutual exclusion (in this case, for casting the vote for a voter identifier) across a wide area is needed, and indeed this was one of the driving applications in the design of Phalanx.

The scalability and survivability goals of Phalanx, driven by applications such as those above, are different from the

***preprint of paper to appear in 17th IEEE Symposium on Reliable Distributed Systems, October 1998, Purdue University, West Lafayette, Indiana**

goals of any other system of which we are aware. And not surprisingly, known approaches to building fault-tolerant replicated data and shared data abstractions do not suffice for our goals. In particular, the foremost approach for building a survivable service today is *state machine replication* [28], in which every (available) server receives, processes, and responds to every client request; some examples of systems implementing this approach are [29, 26, 11]. Because every server must reliably receive every request, this approach generally does not scale well. Numerous other approaches that support persistent and/or replicated data in a distributed system with tolerance to only benign (crash) failures are not suitable, either. These can be coarsely categorized as group communication systems (see [25]), distributed transactional systems (e.g., Argus [15], and Thor [18]), and shared-memory emulation systems (e.g., Ivy [17] and Munin [3]). Though these systems allow more efficient data access than Phalanx, their simpler access protocols cannot mask the arbitrary corruption of data replicas.

Phalanx therefore is constructed using a different approach in order to meet its scalability and survivability goals. At the foundation of Phalanx are novel *quorum* constructions that enable clients to complete operations on shared data objects after interacting with only a typically small subset (quorum) of servers, with no centralized locking or management, and with no server-to-server interaction. Quorums can be surprisingly small—e.g., comprised of only $O(\sqrt{n})$ out of a total of n servers—and thus client access protocols can efficiently scale to hundreds and possibly even thousands of servers. Quorum systems can be constructed to tolerate failures ranging from benign to fully arbitrary [20, 23], and for either type of failure can provide either strict consistency guarantees or only probabilistic ones [24]. Phalanx allows clients to dynamically tune the quorums they use for their application needs.

Quorums are fundamental to Phalanx, and all objects are implemented using quorum-based protocols. The basic operation performed by a Phalanx client is a *quorum remote procedure call* (Q-RPC; see Figure 1), which is the equivalent of performing the same RPC to each server in some quorum. Over Q-RPC, Phalanx servers implement an object store that is accessed by object stubs residing at clients. One of the purposes of this paper is to describe the implementation of two types of objects implemented at this layer:

1. **Shared variables**: A shared variable supports read and write operations by clients. For most classes of failure assumptions, we guarantee *atomic* updates [14, 22], and under the most severe failure scenarios—readers, writers, and (a limited number) of servers are all untrusted—then we provide only *safe* semantics [14, 22].

2. **Mutual exclusion**: A mutual exclusion object grants a right to at most one client out of some number of contending clients. Phalanx supports *at-most-one* mutual exclusion and *exactly-one* mutual exclusion objects, differing in the conditions which guarantee that some client succeeds in gaining the right.

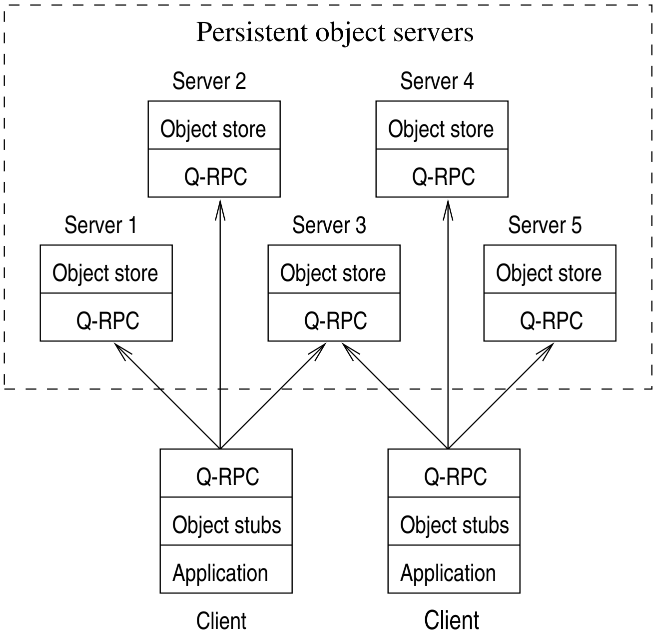

Figure 1. Phalanx system architecture

Using these and other objects, we are presently developing higher-level services and applications for clients, including transactions on multiple objects. In Section 6, we demonstrate one such application, an electronic voting system built over Phalanx.

2. System model and preliminaries

We presume a system model consisting of *servers* and *clients* that need not necessarily be distinct. A *correct* client or server is one that obeys its functional specification. One that deviates from its specification is *faulty*. Faulty clients or servers can exhibit arbitrary (Byzantine) behavior, including collaborating with other faulty clients and servers. We assume that at most b servers fail. Throughout this paper it is convenient to further partition the faulty clients into two sets: those that fail benignly and the *dishonest*. Formally, the only property that this partition must have is that any client that ever suffers a "truly Byzantine" failure—i.e., any failure that cannot be classified as a crash, omission, or timing failure [4]—must be classified as dishonest. The *honest* clients are all those that are not dishonest, i.e., all correct and benignly faulty ones.

As mentioned in Section 1, our protocols leverage the power of *quorum systems* to make operations as efficient as possible. A quorum system $\mathcal{Q}$ is a set of subsets of servers with the property that for any $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. Intuitively, this property can be used to ensure that consistency is preserved across multiple operations performed at different quorums. For example, supposing only benign failures for the moment, if a client reads a variable at a quorum of servers, then because there is a server in that quorum that also received the last-written value of the variable, then the client will be sure to obtain it. Of course, in our setting, simply requiring a non-empty intersection between two quorums may not suffice, since all servers in that intersection may be "truly Byzantine" faulty. Therefore, in the following sections we will make use of several variations of quorum systems that are better suited to our environment.

To make use of quorums, clients communicate to servers via a *quorum remote procedure call*. Given a quorum system, a client's invocation of Q-RPC(m), where m is a request, returns responses from a quorum of servers to the request m. To do this, Q-RPC(m) sends m to servers as necessary to collect responses from a quorum, and then returns these responses to the client. The Q-RPC module provides additional interfaces, e.g., that enable a calling routine to specify servers to avoid because those servers have been detected to be faulty (e.g., based on responses they returned to other Q-RPCs), or that enable a calling routine to issue a query to a partial quorum to complete a previous Q-RPC in which faulty servers returned useless (e.g., syntactically incorrect) values. For the purposes of this paper, however, we omit these interfaces from further discussion. In our protocols, correct servers never send messages to other servers, and correct clients never send messages to other clients.

We assume the existence of trapdoor one-way functions [5], which are sufficient for constructing digital signature schemes. In our protocols, we will often assume that a client or server possesses a private key known only to itself with which it can *digitally sign* messages, and that any other client or server can verify the origin of that signed message. Not all messages will be signed; we will explicitly indicate that the message m is signed by u by denoting it $\langle m \rangle_u$.

3. Shared variables

We begin by describing the implementation of *shared variables* in Phalanx. A shared variable supports *read* and *write* operations on it, by which clients can read and update the variable, respectively.

A variable x is represented in our system by a single copy at each server. That is, every server u maintains a copy x_u of the variable x and an associated timestamp $t_{x,u}$, initially zero. In our protocol descriptions that follow, we assume that there is a set W_x of *writers* that have permission to write to x, and that servers and readers can test whether a given client is a member of W_x (e.g., W_x could be encoded in the name of x or stored in a separate variable). During a write by some $w \in W_x$, the timestamp $t_{x,u}$ is updated at some servers u. Our protocols require that writes by different writers result in different timestamps, and thus for each writer w there is a known set T_w of timestamps that does not intersect $T_{w'}$ for any other $w' \in W_x$. The timestamps in T_w can be formed, e.g., as integers appended with the name of w in the low-order bits.

There are two different variable implementations that Phalanx supports, namely one that assumes that the writers of a variable are honest (Section 3.1) and (a more expensive) one that makes no assumptions about writers (Section 3.2). These two protocols build upon two similar but weaker protocols in [20].

As a practical matter, we note that shared variables can be used to store arbitrary contents, e.g., large files. In the case of large files, though, it may be desirable to read and write only portions of the file rather than the entire contents of such "variables". This can be performed with minor changes to our protocols below, but in this case, our protocols achieve (only) safe variable semantics.

3.1. Honest writers

When all writers of a variable are assumed to be honest, the main job of the variable read and write protocols is to ensure that faulty servers and readers cannot mislead honest clients. Suppose that each $w \in W_x$ holds a private signing key and each possible reader of the variable holds the corresponding public key. Then, the task of masking the behavior of faulty servers becomes particularly simple. It suffices for w to digitally sign its updates to the variable, and for all clients to employ a quorum system that ensures that some *correct* server is in the intersection of a write quorum and a subsequent read quorum. This correct server can then forward the digitally signed update to the reader, enabling it to verify w's digital signature on the value and adopt this value.

The class of quorum systems that is required in this case is called a b-*dissemination* quorum system [20]. A b-dissemination quorum system is one that ensures that the intersection of any two chosen quorums contains at least one correct server despite up to b faults. More precisely, a quorum system $\mathcal{Q}$ is a b-*dissemination quorum system* if for every $Q_1, Q_2 \in \mathcal{Q}$, $|Q_1 \cap Q_2| \geq b + 1$. Note that if it is presumed that no servers are dishonest ($b = 0$), then a b-dissemination quorum system reduces to a regular quorum system.

Given a b-dissemination quorum system, operations on the shared variable are implemented as follows:

Write: For a writer $w \in W_x$ to write the value v into x, it

1. performs a Q-RPC to obtain a set of timestamps $\{t_{x,u}\}_{u \in Q_1}$ currently held at a quorum Q_1 of servers
2. chooses a timestamp $t \in T_w$ greater than the highest timestamp value, and
3. performs a second Q-RPC to send $\langle v, t \rangle_w$ to each server in some quorum Q_2.

Read: For a client to read x, it

1. performs a Q-RPC to obtain a set of signed value/timestamp pairs $\{\langle v_u, t_u \rangle_{w_u}\}_{u \in Q_1}$ currently held at a quorum Q_1 of servers
2. chooses the pair $\langle v, t \rangle_w$ such that $w \in W_x$ and, among all such pairs, that has the highest timestamp t
3. performs a second Q-RPC to "write back" $\langle v, t \rangle_w$ to servers, to guarantee that $\langle v, t \rangle_w$ is stored at some quorum Q_2, and
4. returns v as the result of the read operation.

In either case, when a server u receives a pair $\langle v, t \rangle_w$ such that $w \in W_x$, $t \in T_w$, and $t > t_{x,u}$, then it stores v, t and w's signature. In practice, the "write back" Q-RPC of a read operation can be optimized to send $\langle v, t \rangle_w$ to a server only if it did not already respond with $\langle v, t \rangle_w$ in the previous Q-RPC.

The digital signature by the writer serves two purposes. First, it prevents a faulty server from forging a value and convincing a client to accept it, since a correct client accepts only values signed by a writer. Second, it prevents a faulty reader from forging a value and convincing a server to accept it when it writes its chosen value back at the end of the read protocol.

Informally, our protocol above guarantees that writes to the variable are *atomic*, so that read and write operations throughout the system appear to occur sequentially. To properly present the semantics provided by our protocols and prove their correctness requires extending the standard treatment of concurrent objects in benign failure environment [14, 9] to an environment with Byzantine clients, which is beyond our scope here. The interested reader is referred to [22] for a formal treatment of this subject. Due to space limitations, we omit a proof for our protocol here.

3.2. Dishonest writers

In this section, we address the case that writers may be dishonest. A natural question to ask is: why should we attempt to provide *any* variable semantics in this case? After all, a dishonest writer could presumably repeatedly write a variable with different values so that no two variable reads ever return the same contents. However, we would like to restrict the damage done by a faulty client so that if it eventually stops, the system will be in (or will return to) a consistent state. Additionally, in some applications, variables can be written only a limited number of times or with only a limited frequency. For example, in the Eternity service briefly described in Section 1, the act of publishing a file is the *only* time that it can be written, i.e., it can be written once. Similarly, in Section 6 we describe another application in which a variable is written at most once. In such cases, a dishonest client should not be able to publish a variable so that different readers see different variable contents. Another example would be an intrusion detection application in which audit logs are consolidated at Phalanx at regular intervals for dissemination to analysis programs; between checkpoints, variables are rendered unwriteable. Our variable implementation presented here at least ensures that between writes, readers will have consistent variable semantics on which to base their analysis efforts.

The first difficulty that arises in treating dishonest writers is that the writer might send different updates to different servers. To guard against this, an *echo* protocol is employed in all write operations to guarantee agreement on the written value among the servers. The echo protocol borrows its name and functionality from the echo protocol in Rampart [26]. An echo protocol works as follows: To send an update to the servers, a writer first obtains signed "echoes" for the update from a quorum of servers. It then writes the new value with the attached set of echoes to each server in some quorum. Since any two quorums intersect, it is not possible for a writer to obtain a full quorum of echoes for each of two different values with the same timestamp, provided that no correct server echoes more than one value with the same timestamp. As a result, the value written with a given timestamp is unique.

Though this suffices to ensure unique updates to the servers, it does not suffice to ensure that readers can identify the correct value. In the previous section, readers relied on the digital signature of the writer to filter out forged values from faulty servers. In the case of a dishonest writer, however, this signature is useless, and so we rely instead on correct servers to mask out the forged values of faulty servers. This is achieved by making use of a *b-masking quorum system* [20], where a quorum system $\mathcal{Q}$ is a *b-masking quorum system* if for every $Q_1, Q_2 \in \mathcal{Q}$, $|Q_1 \cap Q_2| \geq 2b+1$. Again, if it is presumed that no servers are dishonest ($b = 0$), then a b-masking quorum system reduces to a regular quorum system.

The protocol here employs a b-masking quorum system $\mathcal{Q}$ to perform read and write operations as follows:

Write: For a writer $w \in W_x$ to write the value v, it

1. performs a Q-RPC to obtain a set of timestamps $\{t_{x,u}\}_{u \in Q_1}$ currently held at a quorum Q_1 of servers

2. chooses a timestamp $t \in T_w$ greater than the highest timestamp value

3. performs a second Q-RPC to send $\langle v, t \rangle$ to servers, to obtain signed echoes for $\langle v, t \rangle$ from a quorum Q_2 of servers

4. performs a third Q-RPC to forward the echoes from Q_2 and write $\langle v, t \rangle$ at a quorum of servers.

Read: For a client to read a variable x, it

1. performs a Q-RPC to obtain $\{\langle x_u, t_{x,u} \rangle_u\}_{u \in Q_1}$, i.e., the value/timestamp pairs currently held at a quorum Q_1 of servers, each pair digitally signed by the server that holds it

2. discards any value/timestamp pair returned by b or fewer servers, and chooses from the remaining the pair $\langle v, t \rangle$ (that occurs in $b + 1$ responses) with the largest timestamp t. (If no such pair exists, the client returns $\perp$ as the result of the read operation.)

3. performs a second Q-RPC to "write back" $\langle v, t \rangle$ along with $b + 1$ server signatures for it, and

4. returns v as the result of the read operation.

In the write protocol, a server generates a signed echo for $\langle v, t \rangle$ only if $w \in W_x$, $t \in T_w$, and it has not previously echoed $\langle v', t \rangle$ for any $v' \neq v$. In a write protocol and the write-back phase of a read, each server u modifies x_u and $t_{x,u}$ only if t is greater than the value it already holds in $t_{x,u}$ and if the request is accompanied by signed echoes for $\langle v, t \rangle$ from a quorum of servers (in the write protocol) or $b + 1$ server signatures on $\langle v, t \rangle$ (in the write back).

As in the case for honest writers above, we informally state here that the protocols above achieve a *safe read/write variable* semantics, so that the variable eventually stabilizes to a consistent state after writes complete, but may return arbitrary values to a reader while being written.

We also note that in the case of honest writers, the protocol in this section can be used to implement safe variable semantics without requiring signatures by clients. Moreover, in that case, the "echo" phase can be omitted.

4. Mutual exclusion

In addition to the need for shared state, many applications need mechanisms to coordinate updates to that state or to otherwise enforce mutual exclusion for other operations. In this section, we outline two classes of mutual exclusion objects that we have developed for Phalanx. The two classes of mutual exclusion objects that we describe here differ in their liveness properties. Each type provides a "contend" operation that either succeeds or fails. The first type guarantees that if a client is alone in contending for mutual exclusion, then that client succeeds, though no client might succeed if multiple clients contend for it. Accordingly, we call this "at-most-one" mutual exclusion, indicating that at most one (but possibly zero) clients succeed. The second type ensures that some client succeeds no matter how many (but at least one) contend for mutual exclusion; we call this "exactly-one" mutual exclusion.

4.1. At-most-one mutual exclusion

In some applications, one may need to achieve exclusive access to certain resources, but contention for those resources may *a priori* be unlikely or indicative of foul play. For example, one of the goals in an electronic election system may be to ensure that each voter identifier can be used to cast a vote only once. Achieving this may require enforcing mutual exclusion on attempts to cast a vote for each voter identifier, thereby precluding its use at multiple voting stations. Contention for a voter identifier indicates an attempt to use the voter identifier multiple times, and the vote can be (and probably should be) delayed until this contention is resolved. For such circumstances, here we describe the implementation of at-most-one mutual exclusion in Phalanx.

To implement an at-most-one mutual exclusion object x, each server u maintains a variable x_u, initially set to "free". The protocol makes use of a b-dissemination quorum system $\mathcal{Q}$ over the servers (see Section 3.1). To contend for x, a client digitally signs a request to contend and sends it via a Q-RPC. When a server receives a properly signed request from a client, it returns its value of x_u and, if x_u is presently free, assigns x_u to be the signed request from the client. The client succeeds if it receives responses from every server in some quorum, and none of these responses is a properly signed request by another client. It is easy to verify that this mutual exclusion protocol achieves the following properties: (i) at most one client succeeds; (ii) if c is honest and succeeds, then c contended for mutual exclusion; and (iii) if only one client contends and that client is correct, then that client succeeds.

If each server digitally signs its free response to a client, then a client can use the collection of signed free responses from a quorum of servers as proof that it succeeded in its contention for mutual exclusion. That is, the client can use this as an access token to access the objects guarded by the mutual exclusion object.

4.2. Exactly-one mutual exclusion

The objects of Section 4.1 lack any guarantee of some client succeeding, except in the case that only a single correct client contends for mutual exclusion. In this section, we describe a mutual exclusion object guaranteeing that some client will succeed no matter how many contend for it.

Central to the implementation of these objects in Phalanx are *consensus objects*. Abstractly, a consensus object is a shared object to which a client can *propose* a value and receive a single value in return. The consensus object returns the same value to each client, and the returned value is one proposed by some client. That is, the returned values satisfy the following properties: (i) If any honest client receives v, then all correct clients receive v; and (ii) if any honest client receives v, then some client proposed v. It is well-known that no protocol satisfying (i) and (ii) can guarantee that clients receive a value in a finite number of steps [7]. Phalanx thus employs randomization to ensure that correct clients receive the consensus value with probability one. In the case that clients are presumed to be honest, we employ a randomized protocol based on one due to Aspnes and Herlihy [2], and for the general case (i.e., when clients can be dishonest), we use our own protocol [21]. In either case, these consensus protocols terminate in a expected number of operations polynomial in the number of clients.

With consensus objects in hand, implementing exactly-one mutual exclusion is straightforward. Each mutual exclusion object is represented by a consensus object. To contend for mutual exclusion, a client proposes its own identifier as the consensus value. If it receives its own identifier as the consensus value, then it has succeeded in its contention for mutual exclusion.

Simpler emulations of exactly-one mutual exclusion objects using shared variables can be achieved using adaptations of Dijkstra's mutual exclusion algorithm [6] or Lamport's bakery algorithm [12] to the case of faulty clients; see [22]. However, the consensus-object-based implementation described here, while somewhat more complex, offers certain advantages in practice. In particular, it enables a Phalanx server to ascertain, based on local data only, which client has succeeded in its contention (see [21]), e.g., to enforce access control to guarded objects. Due to the quorum structure of our system, the aforementioned simpler solutions to exactly-one mutual exclusion provide less information to servers and thus less capability for these added features. Due to space constraints, here we omit further discussion of these issues.

5. Scale

The scalability that we believe Phalanx can achieve is based on two factors: efficient protocol design that enables Phalanx to scale well as the number of clients grows, and the use of quorum systems that enable Phalanx to scale well as the number of servers grows. In this section, we briefly consider these two factors.

We first note that in our design, clients need never communicate with one another, nor do servers communicate among themselves, and thus, interaction is limited to occur between clients and (quorums of) servers. In examining the variable implementations of Section 3, it should be clear that growth in the number of clients should have minimal impact on the latency of our protocols. Servers process a small constant number of messages and perform a small constant number of computations (in particular, digital signatures and verifications) per read or write operation. As such, growth in the number of clients results in at most a linear growth in the load on each server for variable operations, and with a proper choice of quorum system (see below), this growth can be slow in practice. Our implementation of at-most-one mutex objects also share this property. Our implementation of exactly-one mutex objects does not scale as elegantly, because in the consensus object implementations that we employ [2, 21], the amount of work a client must do to obtain the consensus value is a function of the number of clients that have proposed values for it. However, we expect that in the applications we envision, mutex objects will seldom be contended for by more than a handful of clients at any time.

Scalability with growth in the number of servers is primarily dictated by the quorum system used, as quorum systems exist with a wide array of properties [20, 23, 24]. For example, there are dissemination constructions whose resilience is $b = \frac{\sqrt{n}-1}{2}$ and that have quorum sizes as small as $O(n^{3/4})$ (so, e.g., for $n = 1000$, quorum size is < 200). Moreover, probabilistic constructions exist with quorums of size $O(\sqrt{n})$, that simultaneously have outstanding availability. These latter constructions admit a certain well-defined probability of inconsistency in any given operation.

The best quorum system to use can differ from protocol to protocol. Thus we are constructing Phalanx to be flexible as to the quorums it uses to maintain objects, and to allow even switching between quorum systems dynamically at run time. In particular, different Phalanx objects can be maintained simultaneously using different quorum systems.

For particular applications, it may be possible to further tune the quorum system used to enhance the performance of our protocols. For example, if it is known that variable reads far outnumber variable writes, then it should be possible to employ a quorum construction with distinguished read quorums and write quorums that optimize reads at the cost of more expensive writes.

6. Example application: A distributed voting system

As discussed in Section 1, the design of Phalanx was partly driven by the requirements of an electronic voting system for Costa Rica that we helped to develop. In this section we briefly describe a simplified voting application that we have built using Phalanx. This application both captures many of the requirements of the Costa Rican elections and demonstrates some of the features of Phalanx.

In our voting system, there are n polling stations $u_1, \ldots, u_n$ that are geographically distributed but that can communicate over a network. It is presumed that n can be large; e.g., in the case of the Costa Rican elections, $n > 1000$. During the course of an election, each voter can visit a polling station to cast her vote. The goals of the election system can be stated informally as follows.

1. At the end of the election, there is a consistent per-station, per-candidate vote tally that is available to all polling stations.
2. The total number of votes cast in the election is at most the total number of voters who actually voted.
3. If polling station u is correct and voter v voted at u, then v's vote is counted correctly in the tally.
4. If polling station u is correct, then all that is revealed about votes cast at u are (i) which voters voted at u, and (ii) how many votes were cast for each candidate at u. No information about individual votes is revealed (beyond what can be inferred from (i) and (ii)).

We admit the possibility, albeit unlikely, that a polling station could be corrupted (i.e., fails), in which case it could inhibit or change the votes cast by the voters that visit it. Nevertheless, the properties above limit and isolate the effects of a faulty station so that, for example, it can be determined if it could have any effect on the final outcome of the elections.

A voter is named by a voter identifier (VID). It is presumed that each voter is given her VID (e.g., on a voting card) during some registration process that is outside the scope of our system. Each VID is a large, unpredictable string that is not known to the polling stations. However, each polling station holds, for each VID, an *access tag* $\langle h(\mathsf{VID}), \{h(f(s_i, \mathsf{VID}))\}_{1 \leq i \leq n}\rangle$ where h is a one-way collision-resistant function, f is a pseudo-random function, and s_i is a large random value known only to u_i. These access tags are installed at each polling station by the authority overseeing the elections. Upon receiving a VID from a voter, the polling station can determine whether it is valid by seeing if $h(\mathsf{VID})$ is the first component of any access tag. However, because h is one-way, these access tags are not helpful in allowing a polling station to predict VIDs. Below we will see that these access tags are also used to ensure property 3 above.

The polling stations in our system act both as Phalanx servers and as clients of the system that make use of Phalanx-replicated objects. As clients, the polling stations employ one at-most-one mutual exclusion object (see Section 4.1) per VID, which is named by the value $h(\mathsf{VID})$ and for which station u_i is allowed to contend only if it includes $f(s_i, \mathsf{VID})$ in its request to contend. More precisely, the voting protocol proceeds as follows: When a voter visits a polling station, she enters her voter identifier VID and vote at the polling station. The polling station u_i takes the following steps, in order.

- It computes $y_1 = h(\mathsf{VID})$ and $y_2 = f(s_i, \mathsf{VID})$.
- It confirms that there is an access tag of the form $\langle y_1, S\rangle$ (where S denotes some set). If not, then VID is not a valid voter identifier, and u_i rejects it.
- It contends for the mutex object named y_1, including y_2 in its signed request to contend. If this fails (i.e., some server returns a signed request from another server u_j containing a y_2' such that $h(y_2') \in S$), then this VID has already been used in this election and u_i rejects it.
- It accepts the voter's vote and stores it in a *local* tally.

Each server, upon receiving $\langle y_1, y_2\rangle$ in u_i's signed request to contend for the mutex object named y_1, finds the access tag $\langle y_1, S\rangle$ and verifies that $h(y_2) \in S$. If so, the server interprets this request as a request to contend for the object named y_1.

The second type of shared object employed by polling stations is an untrusted-writer variable (see Section 3.2) per polling station per candidate. At the end of the election, u_i writes its tally for candidate c into the designated shared variable. Each such tally variable is "write once", i.e., can be written at most once. Once all polling stations have done this, any polling station can compute the overall election results by reading all shared variables and tallying the individual scores from each polling station.

Briefly, the properties described above for our system are achieved as follows. Property 1 is achieved due to the consistency of values written to the untrusted-writer shared variables that, in this case, can be written only once. Property 2 can be ensured during a post-election audit by verifying that each polling station reports a number of votes that is at most the total number of voter identifiers for which it succeeded in its mutual exclusion contentions. Property 3 holds because unless a voter visits another polling station, no other station can contend for her voter identifier (due to the properties of the access tags) and so the correct station that she does visit can complete her vote. Property 4 holds because the only information recorded in shared state is (i) for which voter identifiers have polling stations contended and (ii) the tally for each candidate per polling station.

7 Conclusion

In this paper we introduced the Phalanx software system for the construction of survivable and scalable data repositories. The distinguishing features of Phalanx are its implementation of strong data abstractions in a scalable way using untrusted servers and clients. The applications for which we are targeting Phalanx include critical components of large scale public-key infrastructures, publishing and dissemination services, and national election systems. In this paper we focused on the protocols used to implement some of the basic data abstractions that Phalanx supports, in particular shared variables and mutual exclusion objects. We argued for the scalability of Phalanx based on the efficiency of these protocols and the novel use of quorum systems at the core of Phalanx, and we described a prototype voting application that we have built with the system.

References

[1] R. J. Anderson. The Eternity Service. In *Proceedings of Pragocrypt '96*, 1996.

[2] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11:441–461, 1990.

[3] J. K. Bennet, J. B. Carter and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[4] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation* 18(1), pages 158–179, 1995.

[5] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory* IT-22(6):644–654, November 1976.

[6] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM* 8(9):569, September 1965.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2):374–382, April 1985.

[8] M. Gasser, A. Goldstein, C. Kaufman and B. Lampson. The Digital distributed system security architecture. In *Proceedings of the 12th NIST/NCSC National Computer Security Conference*, pages 305–319, October 1989.

[9] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463–492, July 1990.

[10] International Telegraph and Telephone Consultative Committee (CCITT). The Directory – Authentication Framework, Recommendation X.509, 1988.

[11] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, pages 317–326, January 1998.

[12] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM* 17(8):453–455, August 1974.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[14] L. Lamport. On interprocess communication (part II: algorithms). *Distributed Computing* 1:86–101, 1986.

[15] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 111-122, 1987.

[16] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[17] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[18] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of SIGMOD '96*, pages 318–329, June 1996.

[19] P. V. McMahon. SESAME V2 public key and authorization extensions to Kerberos. In *Proceedings of the 1995 Internet Society Symposium on Network and Distributed System Security*, pages 114–131, February 1995.

[20] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing* 11(4), to appear. Preliminary version appears in *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 569–578, May 1997.

[21] D. Malkhi and M. Reiter. Survivable consensus objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[22] D. Malkhi, M. Reiter and N. Lynch. A correctness condition for memory shared among Byzantine clients. Submitted for publication.

[23] D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 249–257, August 1997.

[24] D. Malkhi, M. Reiter, A. Wool and R. Wright. Probabilistic quorum systems. Submitted for publication. Preliminary version appears in *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 267–273, August 1997.

[25] D. Powell, editor. Group communication. *Communications of the ACM* 39(4), April 1996.

[26] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[27] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright. The Ω key management service. *Journal of Computer Security* 4(4):267–287, 1996.

[28] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

[29] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully. Principal features of the VOLTAN family of reliable node architectures for distributed systems. *IEEE Transactions on Computers*, 41(5):542–549, May 1992.