Separating Agreement from Execution for Byzantine Fault Tolerant Services

Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, Mike Dahlin
Laboratory for Advanced Systems Research
Department of Computer Sciences
The University of Texas at Austin

Abstract

We describe a new architecture for Byzantine fault tolerant state machine replication that separates agreement—ordering requests—from execution—processing requests. This separation yields two fundamental and practically significant advantages over previous architectures. First, it reduces replication costs because the new architecture can tolerate faults in up to half of the state machine replicas that execute requests. Previous systems can tolerate faults in at most a third of the combined agreement/state machine replicas. Second, separating agreement from execution allows a general *privacy firewall* architecture to protect confidentiality through replication. In contrast, replication in previous systems hurts confidentiality because exploiting the weakest replica can be sufficient to compromise the system. We have constructed a prototype and evaluated it running both microbenchmarks and an NFS server. Overall, we find that the architecture adds modest latencies to unreplicated systems and that its performance is competitive with existing Byzantine fault tolerant systems.

1 Introduction

This paper explores how to improve the trustworthiness of software systems by using redundancy to simultaneously enhance integrity, availability, and confidentiality. By integrity, we mean that service must process users' requests correctly. By availability, we mean that a service operates without interruption. By confidentiality, we mean that the system only reveals the information a user is authorized to see.

Our goal is to provide these guarantees despite malicious attacks that exploit software bugs. Despite advances in applying formal verification techniques to practical problems [25, 33], there is little near-term prospect of eliminating bugs from increasingly complex software systems. Yet, the increasing importance of software systems makes it essential to develop systems that can tolerate attacks that exploit such errors. Furthermore, the prospect of deploying increasing numbers of systems as access-anywhere services raises both challenges—access-anywhere can mean attack-from-anywhere—and also opportunities—when systems are shared by many users, they can devote significant resources to hardening themselves to attack.

Recent work has demonstrated that Byzantine fault tolerant (BFT) state machine replication is a promising technique for using redundancy to improve integrity and availability [34] and that it is practical in that it adds modest latency [11], can proactively recover from faults [12] and can make use of existing software diversity to exploit "opportunistic n-version programming" [36].

Unfortunately, two barriers limit widespread use of these techniques to improve security. First, although existing general BFT systems improve integrity and availability, they hurt confidentiality. In particular, although either increasing the number of replicas or making the implementations of replicas more diverse reduces the chance that an attacker compromises enough replicas to bring down a service, each also increases the chance that at least one replica has an exploitable bug—and an attacker need only exploit the weakest replica in order to compromise confidentiality. Second, BFT systems require more than three-fold replication to work correctly [7]—for instance, tolerating just a single faulty replica requires at least four replicas. Even with opportunistic n-version programming and falling hardware costs, this replication cost is significant, and reducing the number of replicas would significantly increase the practicality of BFT replication.

In this paper, we explore a new general BFT replication architecture to address these two problems. The key principle of this architecture is to *separate agreement from execution.* Correct state machine replication relies on first agreeing on a linearizable order of all requests [26, 27, 39] and then executing these requests on all state machine replicas, and existing BFT state machine replication systems tightly couple these two functions. Cleanly separating agreement and replication yields two fundamental advantages.

First, our architecture reduces replication costs because the new architecture can tolerate faults in up to half of the state machine replicas responsible for executing requests. In particular, while our system still requires $3f+1$ *agreement replicas* to order requests using f-resilient Byzantine agreement, it only requires a simple majority of correct *execution replicas* to process the ordered requests. This distinction is crucial because in practice execution replicas are likely to be much more expensive than agreement replicas both in terms of hardware—because of increased processing, storage, and I/O—and, especially, in terms of software. When n-version (or opportunistic n-version) programming is used to eliminate common-mode failures across replicas, the agreement nodes are part of a generic library that may be reused across applications, while the cost of replicating execution code must be paid by each different service.

Second, separating agreement from execution leads to a practical and general *privacy firewall* architecture to protect confidentiality through Byzantine replication. In existing state machine replication architectures, a voter co-located with each client selects from among replica replies. Voters are co-located with clients to avoid introducing a new single point of failure when integrity and availability are the goals [39], but when confidentiality is also a goal, existing architectures allow a malicious client to observe confidential information leaked by faulty servers. In our system, a redundant set of privacy firewall nodes restricts communication from the execution nodes (where confidential information is manipulated and stored) to filter out incorrect replies before they reach the agreement nodes or the clients.

Unfortunately, eliminating incorrect replies is not sufficient to protect confidentiality because covert channels may encode arbitrary information by exploiting other aspects of a protocol. For example, a faulty node F in a medical database could answer yes/no questions by replying/not replying when a client's request is rejected because of insufficient access. Then, when a client asks "does Greenspan have cancer?" and receives a reply "Request rejected," it could inspect the reply certificate and know the answer is yes (no) if F is (is not) included in the authenticating set of servers. The privacy firewall architecture systematically applies a simple principle to filter covert channels based on message contents or ordering: *it ensures that the stream of replies is a deterministic function of the stream of requests.* In particular, the architecture eliminates nondeterminism by (1) placing the agreement nodes *outside* of the privacy firewall and the execution nodes *inside* to ensure that request ordering decisions are made without access to confidential information, (2) transmitting replies out of the privacy firewall in sequence number order, and (3) using threshold signatures [14] to prevent server membership from acting as a source of nondeterminism.

We have constructed a prototype system that separates agreement and replication and that implements a privacy firewall. Overall, we find its performance to be competitive with previous systems [11, 12, 36]. For example, for the modified Andrew-500 benchmark [36, 19], our system with the privacy firewall turned on is only about 25% slower than Rodrigues et al.'s BASE system [36]. When used without the privacy firewall, our new architecture reduces the number of state machine replicas by a third. And, although the privacy firewall can be a source of significant overhead due to its reliance on threshold cryptography, this overhead can be controlled by passing batches of replies through the privacy firewall rather than signing each reply. When moderate batch sizes are coupled with the new architecture's reduced state machine replication cost, a request under the privacy firewall architecture costs less than under existing architectures when application requests consume more than about 5ms of processing.

The main contribution of this paper is to present the first study to systematically apply the principle of separation of agreement and execution to (1) reduce the replication cost and (2) enhance confidentiality properties for general Byzantine replicated services. Although in retrospect this separation is straightforward, all previous general BFT state machine replication systems have tightly coupled agreement and execution, have paid unneeded replication costs, and have increased system vulnerability to confidentiality breaches. Note that a similar principle is noted by Lamport in his recent discussion of Paxos [28] which distinguishes *proposers* (our clients), *acceptors* (our agreement machines), and *learners* (our execution machines). Our system differs from Paxos in that it addresses Byzantine faults, and it is the first system to apply this principle to reduce replication costs and to provide confidentiality.

In Section 2, we describe our system model and assumptions. Then, Section 3 describes how we separate agreement from execution and Section 4 describes our confidentiality firewall architecture. Section 5 describes and evaluates our prototype. Finally Section 6 puts related work in perspective and Section 7 summarizes our conclusions.

2 System model and assumptions

We consider a distributed asynchronous system where nodes may operate at arbitrarily different speeds and where there is no *a priori* bound on message delay [26]. We also assume an unreliable network that can discard, delay, reorder, and alter messages. More specifically, our agreement and replication protocols ensure safety regardless of timing, crash failures, message omission, message reordering, and message alterations that do not subvert our cryptographic assumptions defined below. Note, however, that it is impossible to guarantee liveness unless some assumptions are made about synchrony and message loss [16]. Our system makes progress if the relatively weak *bounded fair links* assumption holds. We define a bounded fair links network as a network that provides two guarantees. First, it provides the *fair links* guarantee: if a message is sent infinitely often to a correct receiver then it is received infinitely often by that receiver. Second, there exists some delay T such that if a message is retransmitted infinitely often to a correct receiver according to some schedule from time t_0, then the receiver receives the message at least once before time $t_0 + T$; note that the participants in the protocol need not know the value of T. This assumption appears reasonable in practice assuming that network partitions are eventually repaired.

We assume a Byzantine fault model for machines and a strong adversary. Faulty machines can exhibit arbitrary behavior. For example, they can crash, lose data, alter data, and send incorrect protocol messages. Furthermore, we assume an adversary who can coordinate faulty nodes in arbitrary ways. However, we restrict this weak assumption about machine faults with two strong assumptions. First, we assume that some bound on the number of faulty servers is known; for example a given configuration might assume that at most f of n servers are faulty. Second, we assume that no machine can subvert the assumptions about cryptographic primitives described in the following paragraphs.

Our protocol assumes cryptographic primitives with several important properties. We assume a cryptographic *authentication certificate* that allows a subset containing k nodes out of a set $\mathcal{S}$ of nodes to operate on message X to produce an authentication certificate $\langle X \rangle_{\mathcal{S},\mathcal{D},k}$ that any node in some set of destination nodes $\mathcal{D}$ can regard as proof that k distinct nodes in $\mathcal{S}$ *said* [8] X. To provide a range of performance and privacy trade-offs, our protocol supports three alternative implementations of such authentication certificates that are conjectured to have the desired properties if a bound is assumed on the adversary's computational power: public key signatures [35], message authentication code (MAC) authenticators [11, 12, 41], and threshold signatures [14]. The procedure constructing and verifying authentication certificates with these primitives are straightforward and are detailed in an extended technical report [4].

In order to support the required cryptographic primitives, we assume that correct nodes know their private keys (under signature and shared signature schemes) or shared secret keys (under MAC authenticator schemes) and that if a node is correct then no other node knows its private keys. Further, we assume that if both nodes sharing a secret key are correct, then no other node knows their shared secret key. We assume that public keys are distributed so that the intended recipients of messages know the public keys needed to verify messages they receive.

Note that in practice, public key and threshold signatures are typically implemented by computing a cryptographic digest of a message and signing the digest, and MACs are implemented by computing a cryptographic digest of a message and a secret. We assume that a cryptographic digest of a message X produces a fixed-length collection of bits $D(X)$ such that it is computationally infeasible to generate a second message X' with the same digest $D(X') = D(X)$ and such that it is computationally infeasible to calculate X given $D(X)$. Because digest values are of fixed length, it is possible for multiple messages to digest to the same value, but the length is typically chosen to be large enough to make this probability negligible. Several existing digest functions such as SHA1 [40] are believed to have these properties assuming that the adversary's computational power is bounded.

To allow servers to buffer information regarding each client's most recent request, we assume a finite universe of *authorized clients* that send authenticated requests to the system. For signature-based authenticators, we assume that each authorized client knows its own public and private keys, that each server knows the public keys of all authorized clients, and that if a client is correct, no other machine knows its private key. For MAC-based authenticators, we assume each client/server pair shares a secret and that if both machines are correct, no other node knows the secret. The system tolerates an arbitrary number of faulty clients in that non-faulty clients observe a consistent system state regardless of the actions of faulty clients. Note that a careless, malicious, or faulty client can issue disruptive requests that the system executes (in a consistent way). To limit such damage, applications may implement access control algorithms that restrict which actions can be taken by which clients. For simplicity, our description assumes that a client sends a request, waits for the reply, and sends its next request, but it is straightforward to extend the protocol to allow each client to have k outstanding requests.

The basic system replicates applications that behave as *deterministic* state machines. Given a current state C and an input I, all non-faulty copies of the state machine transition to the same subsequent state C'. We also assume that all correct state machines have a $checkpoint(C)$ function that operates on the state machine's current state and produces sequence of bits B. State machines also have a $restore(B)$ function that operates on a sequence of bits and returns a state machine state such that if a correct machine executes $checkpoint(C)$ to produce some value B, then any correct machine that executes $restore(B)$ will then be in state C. We discuss how to abstract nondeterminism and minor differences across different state machine replicas [36] in Section 3.1.4 below.

3 Separating agreement from execution

Figure 1(a) illustrates a traditional Byzantine fault tolerant state machine architecture that combines agreement and execution [11, 12, 36]. In such systems, clients send authenticated requests to the $3f + 1$ servers in the system, where f is the maximum number of faults the system can tolerate. The servers then use a three-phase protocol to generate cryptographically verifiable proofs that assign unique sequence numbers to requests. Each server then

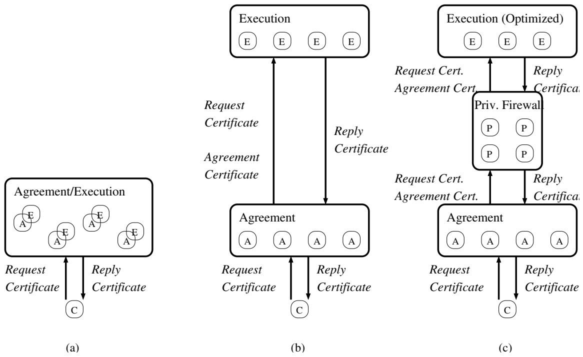


Fig. 1: High level architecture of (a) traditional Byzantine fault tolerant state machine replication, (b) separate Byzantine fault tolerant agreement and execution, and (c) new optimizations enabled by the separation of agreement and execution.

executes requests in sequence-number order and sends replies to the clients. A set of $f + 1$ matching replies represents a *reply certificate* that a client can regard as proof that the request has been executed and that the reply is correct.

Figure 1(b) illustrates our new architecture that separates agreement and execution. The observation that enables this separation is that the agreement phase of the traditional architecture produces a cryptographically-verifiable proof of the ordering of a request. This *agreement certificate* can be verified by any server, so it is possible for execution nodes to be separate from agreement nodes.

Figure 1(c) illustrates two enhancements enabled by the separation of execution and agreement.

1. We can separately optimize the agreement and execution clusters. In particular, it takes a minimum of $3f+1$ servers to reach agreement in an asynchronous system that can suffer f Byzantine faults [7]. But, once incoming requests have been ordered, a simple majority of correct servers suffices to mask Byzantine faults among execution servers—$2g+1$ replicas can tolerate g faults. Note that the agreement and execution servers can be separate physical machines, or they can share the same physical machines.

   Reducing the number of state machine execution replicas needed to tolerate a given number of faults can reduce both software and hardware costs of Byzantine fault tolerance. Software costs are reduced for systems using n-version programming to reduce the correlation of failures across replicas because the $3f+1$ agreement replicas run a generic library that can be reused across applications and only $2g + 1$ application-specific state machine implementations are needed. Hardware costs are reduced because fewer replicas of the request are processed, fewer I/Os are performed, and fewer copies of the state are stored.

2. We can insert a *privacy firewall* between the agreement nodes and the execution nodes to filter minority answers from replies rather than sending these incorrect replies to clients. This filtering allows a Byzantine replicated system to protect confidentiality of state machine data in the face of faults. Note that this configuration requires agreement and execution nodes to by physically separate and to communicate only via confidentiality nodes.

The rest of this section describes a protocol that separates agreement from replication. Section 4 then describes the privacy firewall.

3.1 Inter-cluster protocol

We first provide a cluster-centric description of the protocol among the client, agreement cluster, and execution cluster. Here, we treat the agreement cluster and execution cluster as opaque units that can reliably take certain actions and save certain state. In Sections 3.2 and 3.3 we describe how individual nodes act within these clusters to ensure this behavior.

3.1.1 Client behavior

To issue a request, a client sends a request certificate to the the agreement cluster. In our protocol, request certificates have the form $\langle REQUEST, o, t, c\rangle_{c,\mathcal{A},1}$ where o is the operation requested, t is the timestamp, c is the client that issued the request; the message is certified by the client c to agreement cluster $\mathcal{A}$ and one client's certification is all that is needed.[1] A correct client issues monotonically increasing timestamps; if a faulty client's clock runs backwards, its own requests may be ignored, but no other damage is done.

After issuing a request, a client waits for a reply certificate certified by at least $g + 1$ execution servers. In our protocol a reply certificate has the form: $\langle REPLY, v, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ where v was the view number in the agreement cluster when it assigned a sequence number to the request, t is the request's timestamp, c is the client's identity, r is the result of the requested operation, and $\mathcal{E}$ is the set of execution nodes of which at least $g + 1$ must certify the message to c.

If after a timeout the complete reply certificate has not been received, the client rebroadcasts the request to all agreement nodes. Castro and Liskov suggest two useful optimizations for reducing the communication between the client and the replicated service [11]. First, a client initially sends a request to the agreement server that was the primary during the view v of indicated in the most recent reply received; retransmissions go to all agreement servers. Second, a client's request can designate a specific agreement node to send the reply certificate or can contain the token *ALL* indicating that all agreement servers should send. Again, the client's first transmission can designate a particular server, and retransmissions should designate *ALL*.

[1] Note that our message formats and protocol closely follow Castro and Liskov's [11, 12].

3.1.2 Agreement cluster behavior

The agreement cluster's job is to order requests, send them to the execution cluster, and relay replies from the execution cluster to the client. The agreement cluster acts on two messages—the intra-cluster protocols discussed later will explain how to ensure that these actions are taken reliably.

First, when the agreement cluster receives a message m containing a valid client request certificate $\langle REQUEST, o, t, c\rangle_{c,\mathcal{A},1}$ the cluster proceeds with three steps, the first of which is optional.

1. Optionally check $Cache_c$. $Cache_c$ is an optional data structure that stores the reply certificate for the most recent request by client c. Note that when the privacy firewall (Section 4) is not used, the cache is a performance optimization only, it is required for neither safety nor liveness, and any $Cache_c$ entry may be discarded at any time. Conversely, the agreement cluster is required to maintain a cache of all clients' last replies when the privacy firewall is used. If the reply certificate's timestamp matches that of the request certificate, send the reply to the client and end message processing.

2. If the timestamp on the incoming request exceeds the timestamp of c's last request, generate an agreement certificate that binds the request to a sequence number n. In our protocol, the agreement certificate is of the form $\langle COMMIT, v, n, d, \mathcal{A}\rangle_{\mathcal{A},\mathcal{E},2f+1}$ where v is the current view, n is the sequence number assigned, d is the digest of the client's request ($d = D(m)$), and the certificate is authenticated by at least $2f+1$ of the agreement servers $\mathcal{A}$ to the execution servers $\mathcal{E}$.

   Otherwise, assume this request is a retransmission of c's most recent request and forward the request certificate to the execution cluster, which will trigger a retransmission of the c's most recent reply certificate.

3. Send the request certificate and the agreement certificate to the execution cluster.

Second, when the agreement cluster receives a reply certificate $\langle REPLY, v, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ it relays the certificate to the client. Optionally, it may store the certificate in $Cache_c$.

In addition to these two message-triggered actions, the agreement cluster performs retransmission of requests and agreement certificates if a timeout expires before it receives the corresponding reply certificate. Unlike the traditional architecture in Figure 1(a), communication between the agreement cluster and execution cluster is unreliable. And, although correct clients should repeat requests when they do not hear replies, it would be unwise to depend on (untrusted) clients to trigger the retransmissions needed to fill potential gaps in the sequence number space at execution nodes. For each agreement certificate, the timeout is set to an arbitrary initial value and then doubled after each retransmission.

3.1.3 Execution cluster behavior

The execution cluster implements *application-specific* state machines to process requests in the order determined by the agreement cluster.

To support exactly-once semantics, the execution cluster stores $Reply_c$ where $Reply_c$ stores the timestamp and the reply certificate (if the request has been processed) for the last request received from each client c.

When the execution cluster receives a valid request $\langle REQUEST, o, t, c\rangle_{c,\mathcal{A},1}$ and a valid agreement certificate for that request $\langle COMMIT, v, n, d, \mathcal{A}\rangle_{\mathcal{A},\mathcal{E},2f+1}$ it checks to see if $Reply_c$ already contains the reply. If so, the cluster simply resends the reply to the agreement cluster. $\langle REPLY, v, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ Otherwise, the cluster waits until all requests with sequence numbers smaller than n have executed. It then executes the request, updates $Reply_c$, and sends the reply certificate to the agreement cluster.

Note that in systems not using the privacy firewall architecture described in Section 4, a possible optimization is for the client to send the request certificate directly to both the agreement cluster and the execution cluster and for the execution cluster to send reply certificates directly to both the agreement cluster and the client.

3.1.4 Non-determinism

The state machines replicated by the system must be deterministic to ensure that their replies to a given request match and that their internal states do not diverge. However, many important services include some nondeterminism when executing requests. For example, in the network file system NFS, different replicas may choose different file handles to identify a file [36] or attach different last-access timestamps when a file is read [11]. To address this issue, our prototype employs the standard technique [11, 12, 36] of resolving nondeterminism by having the primary agreement node select any nondeterministic value needed by a request and include that value in agreement certificates; each execution node runs a deterministic algorithm to evaluate whether the proposed value is correct to guard against mischief from a faulty primary and to take some action if it is not.

Note that this approach slightly compromises our goal of separating agreement from execution because agreement nodes may need to have some application-specific knowledge in order to identify requests that require intervention and to select the values they need. Although this pragmatic approach has caused no difficulties with the applications we have examined, relaxing the separation between agreement and execution may not be desirable for some applications. An alternative is to add an additional round trip between the execution cluster and agreement cluster whereby the the agreement cluster collects a list of proposals from the execution cluster and the execution cluster deterministically selects from those proposals.

Note that for systems employing the confidentiality filters described in Section 4, allowing different nondeterministic outputs from the execution nodes to reach the agreement nodes can enable a covert channel that can compromise confidentiality if an agreement node is faulty. Therefore, the first approach above for resolving nondeterminism may be preferred to the second in such environments.

3.2 Internal agreement cluster protocol

Above, we described the high-level behavior of the agreement cluster as it responds to request certificates, reply certificates, and timeouts. Here we describe the internal details of how the nodes in our system behave to meet these requirements.

For simplicity, our implementation uses Rodrigues et al.'s BASE (BFT with Abstract Specification Encapsulation) library [36] as a Byzantine agreement module that handles the details of three-phase commit, view changes, checkpoints, garbage collecting logs, and filtering client requests with old timestamps [11, 12, 36]. In particular, clients send their requests to the BASE library on the agreement cluster to bind requests to sequence numbers. But, when used in our agreement cluster library, the BASE library does not execute requests directly against the *application* state machine, which is in our execution cluster. Instead, we install a message queue (described in more detail below) as the BASE library's *local* state machine, and the BASE library "executes" a request by calling *msgQueue.insert(request certificate, agreement certificate)*. From the point of view of the existing BASE library, when this call completes, the request has been executed. In reality, this call enqueues the request for asynchronous execution by the execution cluster, and the replicated message queues ensure that the request is eventually executed by the execution cluster.[2] Our system makes three simple changes to the existing BASE library to accommodate this asynchronous execution.

1. First, whereas the original library sends the result of the local execution of a request to the client, the modified library does not send replies to clients; instead, it relies on the message queue to do so.
2. Second, when the modified library receives repeated requests from a client, it does not send the locally stored reply since the locally stored reply is the result of the enqueue operation, not the result of the executing the body of the client's request. Instead, it calls *msgQueue.retryHint(request certificate)*, telling the message queue to retry the request or to send the cached reply.
3. Third, in order to achieve a consistent checkpoint at some sequence number n across message queue instances despite their asynchronous internal operation, the modified BASE library calls *msgQueue.sync()* after inserting message n. This call returns after bringing the local message queue state to a consistent state as required by the BASE library's checkpointing and garbage collection algorithms [11, 12, 36].

[2] Note the **local state machine** on which the BASE library operates is simply part of our agreement cluster library; it has nothing to do with the **application state machine** replicated among the execution cluster.

Message queue design. Each node in the agreement cluster has an instance of a message queue as its local state machine. Each message queue instance stores $maxN$, the highest sequence number in any agreement certificate received, and $pendingSends$, a list of request and agreement certificates that have been enqueued but not yet acknowledged as well as retransmission-timeout information for each one. Optionally, each instance may also store $cache_c$, the reply certificate for the highest-numbered request by client c.

When the library calls *msgQueue.insert(request certificate, agreement certificate)*, the message queue instance stores the certificates in *pendingSends*, updates maxN, and multicasts both certificates to all nodes in the execution cluster. It then sets a per-request timer to the current time plus an initial time-out value. As an optimization, when a message is first inserted, only the current primary needs to send it to the execution cluster; in that case, all nodes retransmit if the timeout expires before they receive the reply.

When an instance of the message queue receives a valid reply certified by $g + 1$ nodes of the execution cluster, it deletes from *pendingSends* the request and agreement certificates for that request and for all requests with lower sequence numbers; it also cancels the retransmission timer for those requests. The message queue instance then forwards the reply to the client. Optionally, the instance updates $cache_c$ to store the reply certificate for client c.

When the modified BASE library calls *retryHint(request certificate)* for a request r from client c, the message queue instance first checks to see if $cache_c$ contains the desired reply certificate. If so, it sends the reply certificate to the client. Otherwise, it resends the request certificate to the execution cluster.

When the retransmission timer expires for a message in *pendingSends*, the instance resends the request and agreement certificates and updates the retransmission timer to the current time plus twice the previous timeout interval.

Finally, when the modified BASE library calls *msgQueue.sync()*, the message queue stops accepting *insert()* requests and waits to receive a reply certificate for a request with sequence number $maxN$ or higher. Once it processes such a reply and *pendingSends* is therefore empty, the *sync()* call returns and the message queue begins accepting *insert()* calls again.

3.3 Internal execution cluster protocol

Above, we defined the high-level execution cluster's behavior in response to request and agreement certificates.

Here, we describe execution node behaviors that ensure these requirements are met.

Each node in the execution cluster maintains the application state, a log of received but not yet executed requests, and a table $reply$ where $reply_c$ stores the node's piece of its most recent reply certificate for client c. Each node also stores the most recent *stable checkpoint*, which is a checkpoint across the application state and the *reply* table that is certified by at least $g+1$ execution nodes. Nodes also store zero or more newer checkpoints that have not yet become stable.

When a node receives a valid request certificate certified by a client c or a valid agreement certificate certified by at least $2f+1$ agreement nodes, it stores the certificate in the pending request list if the request is newer than the reply stored in $reply_c$. Conversely, if the request's timestamp is not new, the node resends the stored reply to the agreement cluster.

Once an execution node has received both the request certificate and the corresponding agreement certificate with sequence number n and once all requests with sequence numbers lower than n have been processed, the node executes request n on the application-specific state machine. It then authenticates the reply by generating this node's share of the full reply certificate, stores the partial reply certificate in $reply_c$, and sends the partial reply certificate to all nodes in the agreement cluster.

Liveness and retransmission. To eliminate gaps in the sequence number space caused by the unreliable communication between the agreement and execution clusters, the system uses a two-level retransmission strategy. For a request with sequence number n, retransmissions by the agreement cluster ensure that eventually at least one correct execution node receives and executes request n, and an internal execution cluster retransmission protocol ensures that once that happens, all correct execution nodes eventually learn of request n or of some stable checkpoint newer than n. In particular, if an execution node receives request n but not request $n-1$, it multicasts to other nodes in the execution cluster a retransmission message for the missing sequence number. When a node receives such a message, it replies with the specified request and agreement certificates unless it has a stable checkpoint with a higher sequence number, in which case it sends the proof of stability for that checkpoint (see below.)

Checkpoints and garbage collection. The execution nodes periodically construct checkpoints to allow them to garbage collect their pending request logs. Note that the inter-cluster protocol is designed so that garbage collection in the execution cluster requires no additional coordination with the agreement cluster and vice versa. Execution nodes generate checkpoints at prespecified sequence numbers (e.g., after executing request n where $n \bmod CP\_FREQ = 0$. Nodes checkpoint both the application state and their $reply_c$ list of replies to clients, but they do not include their pending request list in checkpointed state. As in previous systems [11, 12, 36], to reduce the cost of producing checkpoints, nodes can make use of copy on write and incremental cryptography [6].

After generating a checkpoint, execution servers assemble a proof of stability for it. When server i produces checkpoint C for sequence number n, it computes a digest of the checkpoint $d = D(C)$ and authenticates its view of the checkpoint to the rest of the cluster by multicasting $\langle CHECKPOINT, n, d\rangle_{i,\mathcal{E},1}$ to all execution nodes. Once a node receives $g+1$ such messages, it assembles them into a full checkpoint certificate: $\langle CHECKPOINT, n, d\rangle_{\mathcal{E},\mathcal{E},g+1}$

Once a node has a valid and complete checkpoint certificate for sequence number n, it can garbage collect state by discarding older checkpoints, discarding older checkpoint certificates, and discarding older agreement certificates from the pending request log.

3.4 Correctness

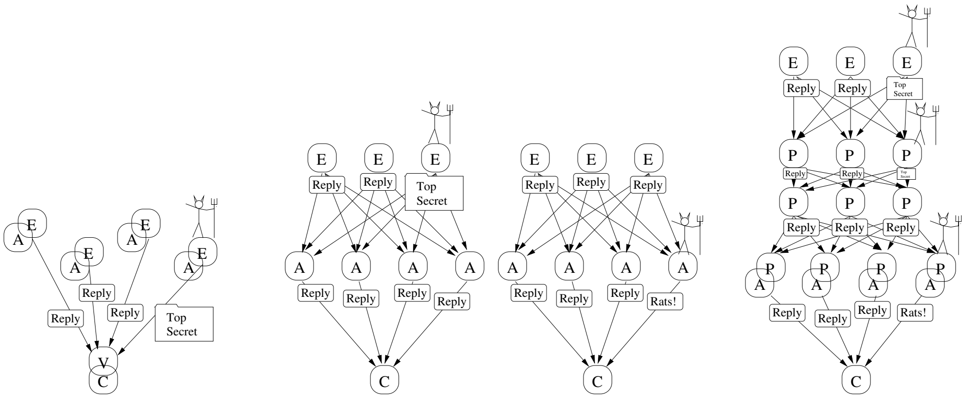
Due to space constraints, we outline our main results here and defer the proof to an extended technical report [4]. We show that the high level protocol provides safety and liveness if the clusters behave as specified. And, we show the node actions specified in our agreement cluster and execution cluster designs ensure that the clusters discharge their requirements.

The system is safe in that a client receives a reply $\langle REPLY, v, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ only if (a) earlier the client issued a request $\langle REQUEST, o_n, t, c\rangle_{c,\mathcal{A},1}$, (b) the reply value r reflects the output of state machine in state Q_{n-1} executing request operation o_n, (c) there exists some set of operations $\mathcal{O}$ such that state Q_{n-1} is the state reached by starting at initial state Q_0 and sequentially executing each request o_i $(0 \leq i < n)$ in $\mathcal{O}$ as the ith operation on the state machine, and (d) a valid reply certificate for any subsequent request reflects a state in which the execution of o_i is the i'th action by the state machine $(0 \leq i \leq n)$.

This safety follows from the fact that the agreement cluster only generates agreement certificates for valid client requests and never assigns the same sequence number to two different requests. Then, the execution cluster only executes requests in the order specified by the agreement certificates.

The system is live in that if a client c sends a request with timestamp t where timestamp t exceeds any timestamp in any previous request by c and the client repeatedly sends that request and no other request until it receives a reply, then eventually it will receive a valid reply for that request.

This liveness follows from the fact that if a client repeats a new request often enough, the agreement cluster must eventually receive it and assign it a sequence number. Once that happens, the execution cluster must eventually receive it. After that, if the client keeps sending its request, eventually it will receive the reply.



(a) (b) (c)

Fig. 2: Illustration of confidentiality filtering properties of (a) traditional BFT architectures, (b) architectures that separates agreement and execution, and (c) architectures that separate agreement and execution and that add additional privacy firewall nodes.

4 Privacy firewall

In traditional BFT systems, there is a fundamental tradeoff between increasing availability and integrity on the one hand and strengthening confidentiality on the other. Increasing diversity across replicas (e.g., increasing the number of replicas or increasing the heterogeneity across implementations of different replicas [2, 23, 42]) improves integrity and availability because it reduces the chance that too many replicas simultaneously fail. Unfortunately, it also increases the chance that at least one replica contains an exploitable bug. An attacker may gain access to the confidential data by attacking the weakest replica.

Compounding this problem, as Figure 2(a) illustrates, traditional replicated state machine architectures delegate the responsibility of combining the state machines' outputs to a voter at the client. Fate sharing between the client and the voter ensures that the voter does not introduce a new single point of failure; to quote Schneider [39], *"the voter—a part of the client—is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant"* because a faulty voter is then synonymous with a faulty client.

Although such a model is appropriate for services where availability and integrity are the goals, it fails to address confidentiality for access-anywhere services. In particular, if an attacker manages to compromise one replica in such a system, the compromised replica may send confidential data back to the attacker.

Solving this problem seems difficult. If we move the voter away from the client, we lose fate sharing and the voter becomes a single point of failure. And, it is not clear how to replicate the voter to eliminate this single point of failure without miring ourselves in endless recursion ("Who votes on the voters?").

As illustrated by Figure 2(b), the separation of agreement from execution provides the opportunity to reduce a system's vulnerability to compromising confidentiality by having the agreement nodes filter incorrect replies before sending reply certificates to clients. It takes now a failure of both an agreement node and an execution node to compromise privacy if we restrict communications so that (1) clients can communicate with agreement nodes but not execution nodes and (2) request and reply certificates are encrypted so that clients and execution nodes can read them but agreement nodes cannot. In particular, if all agreement nodes are correct, then the agreement nodes filter replies so that only correct replies reach the client. Conversely, if all execution nodes are correct, then faulty agreement nodes can send information to clients, but not information regarding the confidential state of the state machine.

Although this simple design improves confidentiality, it is not entirely satisfying. First, it can not handle multiple faults: a single fault in both the agreement and execution clusters can allow confidential information to leak. Second, it allows an adversary to leak information via steganography, for instance by manipulating membership sets in agreement certificates as in the Introduction's "Greenspan" example.

In the rest of this section, we describe a general redundant confidentiality filter architecture—the *privacy firewall*–that prevents confidential information from leaking to clients in replies.

4.1 Privacy firewall

As indicated in Figure 2(c), our solution inserts privacy firewall *filter nodes* P between execution servers E and agreement nodes A to pass only information sent by correct execution servers. Our protocol makes use of four key ideas: (1) separation of agreement from replication, (2) restriction of communication to ensure filtering, (3) redundant filters to tolerate up to a threshold number of faults, and (4) elimination of non-determinism to allow filters to eliminate unauthorized steganographic communication in the data stream.

If the agreement and execution clusters have a sufficient number of working machines, then a privacy firewall of $h+1$ rows of $h+1$ filters per row can tolerate up to h faults while still providing availability, integrity, and confidentiality. Information flow is controlled by restricting communication; as Figure 2(c) illustrates, each filter node has a physical network connection to all filter nodes in the rows above and below and none in the same row. If the number of agreement nodes is at least $h + 1$, then the bottom row of filters can be merged with the agreement nodes by placing a filter on each server in the agreement cluster.

Communication is also restricted in that the body of requests (e.g., the operation field o in the request certificates) and the body of replies (e.g., the result field r in the reply certificates) are encrypted. This prevents agreement nodes and filter nodes from examining the bodies of requests and replies to accumulate confidential information in locations not protected by the privacy firewall. Clients encrypt requests using the public key or per-node shared secret of the execution nodes (used in MAC authenticators), and execution nodes encrypt replies using the public key or per-node shared secret of the client. Request and agreement certificates flowing "up" the firewall can use any form of certificate they choose (including MAC-based authenticators). However, reply certificates must use threshold cryptography [14] if the system is to prevent steganographic information leakage via certificate membership sets.

Filter nodes follow a simple protocol. When they receive a valid request certificate or agreement certificate from below, they send it to the node directly above them; the top row multicasts the certificate to all execution nodes (or, as an optimization, the filters in the topmost row can forward the certificates to specific execution nodes on the first transmission and can multicast to all on retransmissions.)

Then, when a filter node first receives a reply certificate with sequence number n and certified by $g + 1$ execution nodes, the filter takes three steps. First, it verifies the certificate. For the top row of filters, this means verifying the partial signatures comprising the certificate and combining them into a single signature by the execution nodes' split group key. All other rows just verify the group signature. Second, the filter node waits to receive all replies with sequence numbers smaller than n. Third, after sending all replies with sequence numbers smaller than n, the filter sends the reply certificate for request n to all nodes in the row below; it retransmits this certificate to each node below it until it receives an acknowledgment or until it receives a reply certificate with sequence number $n + L$ where L is the garbage collection interval in the agreement cluster; such a number allows a filter node to stop retransmitting and garbage collect reply n because it knows the agreement cluster has received and stored that reply.

Since agreement nodes cannot read the requests they forward to the layers above, it is more difficult to have the primary agreement node resolve non-determinism in replies and state of execution nodes. We tackle non-determinism by having the primary perform oblivious resolution of non-determisnistic responses. For instance, the primary may attach a timestamp to each agreement certificate without looking at the contents of the request. We take this approach for our NFS prototype in Section 5.4. However, this approach may not be feasible for applications that depend on significant amounts of application state, confidential information, or sophisticated application-specific logic to resolve non-determinism. And, as noted in Section 3.1.4, adding an extra round of communication between the agreement and execution cluster provides an opportunity to leak information. Balancing confidentiality and non-determinism in its full generality appears hard, as malicious nodes can encode confidential information by probabilistically influencing selection of specific replies from a non-deterministic set of responses. Fortunately, in the applications that we have examined, it has proved straightforward for the agreement machines to resolve non-deterministic choices obliviously.

4.2 Filter properties and limitations

The following definitions and lemmas express the confidentiality guarantees provided by our privacy firewall. We define *confidentiality* as follows. Let C be an abstract state machine specification of a single correct server and S a candidate implementation of C. Let $M[s,(q_1,\ldots,q_n)]$ denote the sequence of symbols output by state machine M in state s given the sequence of input symbols $(q_1,\ldots,q_n)$.

Definition 1 *An implementation S is said to be* confidential *with respect to C iff for any state s, and any sequence of requests $(q_1, q_2, \ldots, q_n)$ of arbitrary length n, $C[s,(q_1,\ldots,q_n)] = (r_1,\ldots,r_n)$ implies that there exists state s' such that $S[s',(q_1,\ldots,q_n)] = (r_1,\ldots,r_n)$, and vice-versa.*

Lemma 1 *The Privacy Firewall architecture, with $2g+1$ execution nodes, $(h+1)^2$ firewall nodes, and $3f+1$ agreement nodes is as confidential as a single correct server in the face of at most g failures of execution nodes, h failures of firewall nodes, and f failures of agreement nodes.*

Note that we define confidentiality with respect to the specification of a single correct server. Our goal is to preserve in a replicated system, whatever level of confidentiality a correct unreplicated server provides. Note that this definition is more flexible and less strict than the notion of *non-interference* [38] which is sometimes equated with confidentiality in the literature and which informally states that the observable output of the system has no correlation to the confidential information stored in the system.

Lemma 2 *A network of firewall nodes requires a minimum of $(h + 1)^2$ nodes to provide confidentiality.*

The proofs of these lemmas are given in [3]. Here we outline several key points for understanding the architecture and the algorithm.

The architecture ensures two properties: i) at least one of these paths consisting only of correct filters and ii) there is at least one correct filter node on all paths from the execution cluster to the agreement cluster.

If we view the privacy firewall as a graph of filter nodes, property (i) ensures availability by guaranteeing that there is always at least one path consisting of only correct filters through which correct replies can reach clients. Observe that availability is necessary for preserving confidentiality as well, because a strategically placed rejected request could be used to communicate confidential information.

Property (ii) ensures that there exists a cut consisting only of correct filter nodes. Faulty filter nodes above this cut might have access to confidential data, but the filter nodes in the correct cut ensure that only replies that would be returned by a correct server are forwarded. And, although faulty nodes below the correct cut might be able to communicate any information they have, they do not have access to confidential information.

The architecture prevents nodes that can access confidential data from exploiting nondeterminism to introduce covert leaks of information. The separation of agreement from execution allows the agreement nodes to impose an order on requests without access to confidential information. Then, filters only transmit replies in that order. And because each filter node multicasts replies to all filter nodes in the row below it, no incorrect filter node can prevent a correct filter downstream from receiving the data it needs.

Though our protocol ensures that the sequence of transmitted data contains only the information transmitted by a correct server, at least two sources of information leaks are not prevented. First, the protocol does not completely eliminate timing channels: it may be possible for a malicious execution server or filter node to transmit information by accelerating or delaying the transmission of certain replies. Eliminating timing channels is difficult; in future work we will consider ways to introduce some synchrony to our system to detect or limit these types of attack. Second, although agreement nodes do not have access to the body of requests, they do need access to the identity of the client (in order to buffer information about each client's last request.) Faulty agreement or filter nodes in our system could leak information regarding traffic patterns. For example, a malicious agreement node could leak the frequency that a particular client sends requests to the system.

5 Evaluation

In this section, we experimentally evaluate the latency, overhead, and throughput of our prototype system under microbenchmarks. We also examine the system's performance acting as a network file system (NFS) server.

5.1 Prototype implementation

We have constructed a prototype system that separates agreement and replication and that optionally provides a redundant privacy firewall. As described above, our prototype implementation builds on Rodrigues et al.'s BASE library [36].

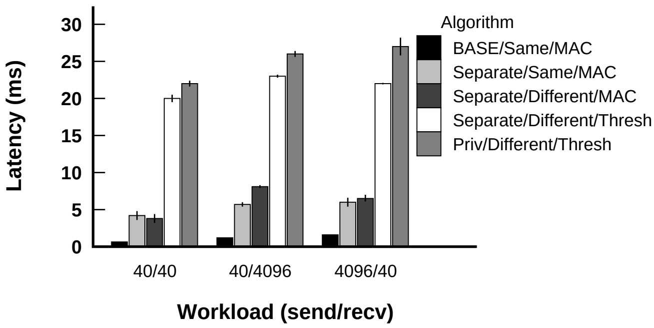


Fig. 3: Latency for null-server benchmark for three request/reply sizes.

Our evaluation cluster comprises nine 933Mhz Pentium-III machines each with 128Kbyte of memory. The machines run Redhat Linux 7.2 and are connected by a 100 Mbit ethernet hub.

Note that two aspects of our configuration would not be appropriate for production use. First, both the underlying BASE library and our system store important persistent data structures in memory and rely on replication across machines to ensure this persistence [5, 11, 13, 30]. Unfortunately, the machines in our evaluation cluster do not have uninterruptable power supplies, so power failures are a potentially significant source of correlated failures across our system that could cause our current configuration to lose data. Second, our privacy firewall architecture assumes a network configuration that physically restricts communication paths between agreement machines, privacy filter machines, and execution machines. Our current configuration uses a single 100 Mbit ethernet hub and does not enforce these restrictions. We would not expect either of these differences to affect the results we report in this section.

5.2 Latency

Past studies have found that Byzantine fault tolerant state machine replication adds modest latency to network applications [11, 12, 36]. Here, we examine the same latency microbenchmark used in these studies. Under this microbenchmark, the application reads a request of a specified size and produces a reply of a specified size with no additional processing. We examine request/reply sizes of 40 bytes/40 bytes, 40 bytes/4 KB, and 4 KB/40 bytes.

Figure 3 shows the average latency and standard deviation of latency for ten runs of 200 requests each. The bars show performance for different system configurations with the *algorithm/machine configuration/authentication algorithm* indicated in the legend. BASE/Same/MAC is the BASE library with 4 machines hosting both the agreement and execution servers and using MAC authenticators; Separate/Same/MAC shows our system that separates agreement and replication with agreement running on 4 machines and with execution running 3 of the same set of machines and using MAC authenticators; Separate/Different/MAC moves the execution servers to 3 machines physically separate from the 4 agreement servers; Separate/Different/Thresh uses the same configuration but uses threshold signatures rather than MAC authenticators for reply certificates; finally, Priv/Different/Thresh adds an array of privacy firewall servers between the agreement and execution cluster with a bottom row of 4 privacy firewall servers sharing the agreement machines and an additional row of 2 firewall servers separate from the agreement and execution machines.

The BASE library imposes little latency on requests, with request latencies of 0.64ms, 1.2ms, and 1.6ms for the three workloads. Our current implementations of the library that separates agreement from replication has higher latencies when running on the same machines—4.2ms, 5.7ms, and 6.0ms. The increase is largely caused by two inefficiencies in our current implementation: (1) rather than using the agreement certificate produced by the BASE library, each of our message queue nodes generates a piece of a new agreement certificate from scratch and (2) in our current prototype, we do a full all-to-all multicast of the agreement certificate and request certificate from the agreement nodes to the execution nodes, of the reply certificate from the execution nodes to the agreement nodes, and of the reply certificate from the agreement nodes to the client; we have not implemented the optimizations of first having one node send and having the other nodes send only if a timeout occurs, and we have not implemented the optimization of clients sending requests directly to the execution nodes and execution nodes sending replies directly to clients. We plan to add these optimizations and measure their effectiveness soon. Separating the agreement machines from the execution machines adds little additional latency. But, switching from MAC authenticator certificates to threshold signature certificates increases latencies to 20ms, 23ms, and 22ms for the three workloads. Adding a two rows of privacy firewall filters (one of which is colocated with the agreement nodes) adds a few additional milliseconds.

As expected, the most significant source of latency in the architecture is public key threshold cryptography. Producing a threshold signature takes 15ms and verifying a signature takes 0.7ms on our machines. Two things should be noted to put these costs in perspective. First, the latency for these operations is comparable to I/O costs for many services of interest; for example, these latency costs are similar to the latency of a small number of disk seeks and are similar to or smaller than wide area network round trip latencies. Second, signature costs are expected to fall quickly as processor speeds increase; the increasing importance of distributed systems security may also lead to widespread deployment of hardware acceleration of encryption primitives. The FARSITE project has also noted that technology trends are making it feasible to include public-key operations as a building block for practical distributed services [1].

5.3 Throughput and cost

Although latency is an important metric, modern services must also support high throughput [43]. Two aspects of the privacy firewall architecture pose challenges to providing high throughput at low cost. First, the privacy firewall architecture requires a larger number of physical machines in order to to physically restrict communication. Second, the privacy firewall architecture relies on relatively high-overhead public key threshold signatures for reply certificates. Two factors mitigate these costs.

First, although the new architecture can increase the total number of machines, it also can reduce the number of application-specific machines required. Application-specific machines may be more expensive than generic machines both in terms of hardware (e.g., they may require more storage, I/O, or processing resources) and in terms of software (e.g., they may require new versions of application-specific software.) Thus, for many systems we would expect the application costs (e.g., the execution servers) dominate. Like router and switch box costs today, agreement node and privacy filter boxes may add a significant but relatively modest amount to overall system cost. Also recall that in configurations without the privacy firewall, the total number of machines is not necessarily increased since the agreement and execution servers can occupy the same physical machines. For example, to tolerate one fault, four machines can act as agreement servers while three of them also act as execution replicas. Even if separate machines are used, the agreement machines execute generic agreement code and application-specific code is restricted to the execution nodes (other than that needed to resolve nondeterminism.) For example, to tolerate one fault, four machines run generic agreement code, and only three application-specific execution machines are needed. Finally, although privacy firewalls must run on $(h+1)^2$ nodes (and this is provably the minimal number to ensure confidentiality), even when the privacy firewall architecture is used, the number of machines is relatively modest when the goal is to tolerate a small number of faults. For example, to tolerate up to one failure among the execution nodes and one among either the agreement or privacy filter servers, the system would have four generic agreement and privacy filter machines, two generic privacy filter machines, and three application-specific execution machines.

Second, a better metric for evaluating the hardware costs of the system than the number of machines is the overhead imposed on each request relative to an unreplicated system. On the one hand, by cleanly separating agreement from execution and thereby reducing the number of execution replicas a system needs, the new architecture often reduces this overhead compared to previous systems. On the other hand, the addition of privacy firewall filters and their attendant public key encryption add significant costs. Fortunately, these costs can be amortized across batches of requests. In particular, when load is high the BASE library on which we build bundles together requests and executes agreement once per bundle rather than once per request. Similarly, by sending bundles of requests and replies through the privacy firewall nodes, we allow the system to execute public key operations on bundles of replies rather than individual replies.

To put these two factors in perspective, we consider a simple model that accounts for the application execution costs and cryptographic processing overheads across the system (but not other overheads like network send/receive.) The *relativeCost* of executing a request is the cost of executing the request on a replicated system divided by the cost of executing the request on an unreplicated system. For our system and the BASE library $relativeCost = \frac{numExec \cdot proc_{app} + overhead_{req} + \frac{overhead_{batch}}{numPerBatch}}{proc_{app}}$
The cryptographic processing overhead has three flavors: MAC-based authenticators, public threshold-key signing, and public threshold-key verifying. To tolerate 1 fault, the BASE library requires 4 execution replicas, and it does 8 MAC operations per request[3] and 36 MAC operations per batch. Our architecture that separates agreement from replication requires 3 execution replicas and does 7 MAC operations per request and 39 MAC operations per batch[4]. Our privacy firewall architecture requires 3 execution replicas and does 7 MAC operations per request and 39/3/6 MAC operations/public key signatures/public key verifications per batch.

Given these costs, the lines in Figure 4 show the relative costs for BASE (dot-dash lines), separate agreement and replication (dotted lines), and privacy firewall (solid lines) for batch sizes of 1, 10, and 100 requests/batch. The (unreplicated) application execution time varies from 1ms per request to 100ms per request on the x axis. We assume that MAC operations cost 0.2ms (based on 50MB/s secure hasing of 1KB packets), public key threshold signatures cost 15ms (as measured on our machines for small messages), and public key verification costs 0.7ms (measured for small messages.)

Without the privacy firewall overhead, our separate architecture has a lower cost than BASE for all request sizes examined. As application processing increase, application processing dominates, and the new architectures gain a 33% advantage over the BASE architecture. With small requests and without batching, the privacy firewall does

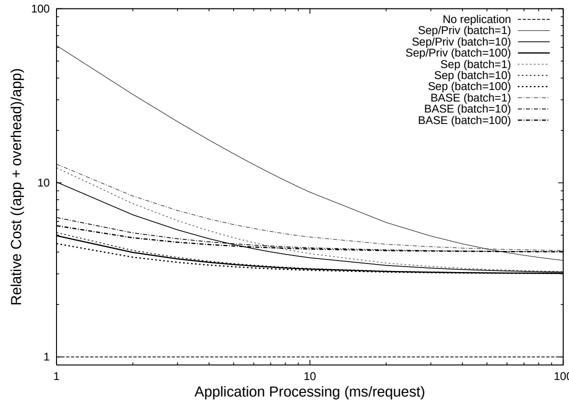


Fig. 4: Estimated relative processing costs including application processing and cryptographic overhead for an unreplicated system, privacy firewall system, separate agreement and replication system, and BASE system for batch sizes of 1, 10, and 100 requests/batch.

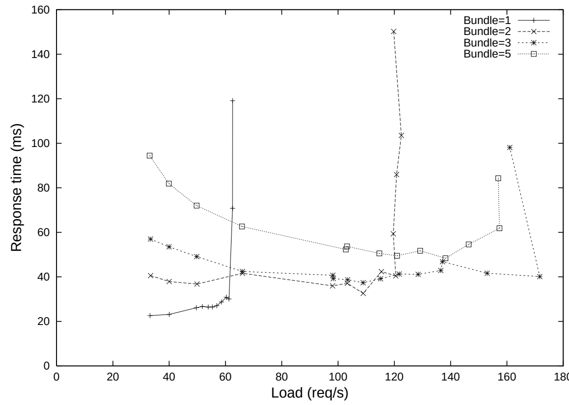


Fig. 5: Microbenchmark response time as offered load and request bundling varies.

greatly increase cost. But with batch sizes of 10 (or 100), processing a request under the privacy firewall architecture costs less than under BASE replication for applications whose requests take more than 5ms (or 0.2ms).

The simple model discussed above considers only encryption operations and application execution and summarizes total overhead. We now experimentally evaluate the peak throughput and load of our system. In order to isolate the overhead of our prototype, we evaluate the performance of the system when executing a simple Null server that receives 1 KB requests and returns 1 KB replies with no additional application processing. We program a set of clients to issue requests at a desired frequency and vary that frequency to vary the load on the system.

Figure 5 shows how the latency for a given load varies with bundle size. When bundling is turned off, throughput is limited to 62 requests/second, at which point the execution servers are spending nearly all of their time signing replies. Doubling the bundle size to 2 approximately doubles the throughput. Bundle sizes of 3 or larger give peak throughputs of 160-170 requests/second; beyond this point, the machine on which our clients run is CPU limited and the servers have idle capacity. For example, with

[3] Note that when authenticating the same message to or from a number of nodes the work of computing the digest on the body of a message can be re-used for all communication partners [11, 12]. For the small numbers of nodes involved in our system, we therefore charge 1 MAC operation per message processed by a node regardless of the number of sources it came from or destinations it goes to.

[4] Our unoptimized prototype does 44 MAC operations per batch both with and without the privacy firewall.

Phase	No Replication	BASE	Firewall
1	6	9	19
2	438	1355	1721
3	390	1130	1279
4	752	1744	2347
TOTAL	1586	4238	5366

Fig. 6: Andrew-500 benchmark times in seconds.

a bundle size of 10 and a load of 160 requests/second, the CPU utilization of the most heavily loaded execution machine is 30%. Note that our current prototype uses a static bundle size, so increasing bundle sizes increases latency at low loads. The existing BASE library limits this problem by using small bundles when load is low and increasing bundle sizes as load increases. Our current prototype uses fixed-sized bundles to avoid the need to adaptively agree on bundle size; we plan to augment the interface between the BASE library and our message queues to pass the bundle size used by the BASE agreement cluster to the message queue.

5.4 Network file system

For comparison with previous studies [12, 11, 36], we examine a replicated NFS server under the modified Andrew500 benchmark, which sequentially runs 500 copies of the Andrew benchmark [11, 19]. The Andrew benchmark has 5 phases: (1) recursive subdirectory creation, (2) copy source tree, (3) examine file attributes without reading file contents, (4) reading the files, and (5) compiling and linking the files.

We use Rodrigues et al.'s NFS abstraction layer to resolve nondeterminism by having the primary agreement node supply timestamps for modifications and file handles for newly opened files. We run each benchmark 10 times and report the average for each configuration.

Figure 6 summarizes these results. Performance for the benchmark is largely determined by file system latency, and our firewall system's performance is about 25% slower than BASE. Also note that BASE is more than a factor of two slower than the no replication case; this difference is much higher than the difference reported in [36] where a 31% slowdown was observed. Over the past 5 weeks, we have worked with the authors of [36] to isolate the cause of this difference, but we do not at present have a good explanation.

6 Related work

Byzantine agreement [29] and Byzantine fault-tolerant state machine replication has been studied in both theoretical and practical settings [7, 10, 18, 22, 34, 39]. The recent work of Castro, Liskov, and Rodrigues [11, 12, 36] has brought new impulse to research in this area by showing that BFT systems can be practical. We use their BASE library [36] as the foundation of our agreement protocol, but depart significantly from their design in one key respect: our architecture explicitly separates the responsibility of achieving agreement on the order of requests from the processing the requests once they are ordered. Significantly, this separation allows us to reduce by one third the number of application-specific replicas needed to tolerate f Byzantine failures and to address confidentiality together with integrity and availability.

In a recent paper [28], Lamport deconstructs his Paxos consensus protocol [27] by explicitly identifying the roles of three classes of agents in the protocol: *proposers*, *acceptors*, and *learners*. He goes on to present an implementation of the state machine approach in Paxos in which "each server plays all the roles (proposer, acceptor and learner)". We propose a similar deconstruction of the state machine protocol: in Paxos parlance, our clients, agreement servers, and execution servers are performing the roles played, respectively, by proposers, acceptors, and learners. However, our acceptors and learners are physically, and not just logically, distinct. We show how to apply this principle to BFT systems to reduce replication cost and to provide confidentiality.

In a recent workshop extended abstract [3] we sketched an early design of our privacy firewall. We move beyond that work in two ways. First, we propose a different overall architecture for achieving confidentiality in BFT services. Instead of placing the privacy firewall between the client and the BFT service, we now exploit the separation of agreement from execution and position the firewall between the two clusters. This change is important because it eliminates a potential covert channel by preventing nodes with confidential information from affecting the order in which requests are processed. Second, we report on our experience building a prototype of our system.

Most previous efforts to achieve confidentiality despite server failures restrict the data that servers can access. A number of systems limit servers to basic "store" and "retrieve" operations on encrypted data [1, 24, 31, 32, 37] or on data fragmented among servers [17, 20]. The COCA [44] online certification authority uses replication for availability, and threshold cryptography [14] and proactive secret sharing [21] to digitally sign certificates in a way that tolerates adversaries that compromise some of the servers. In general, preventing servers from accessing confidential state works well when servers can process the fragments independently or when servers do not perform any significant processing on the data. Our architecture provides a much more general solution that can implement arbitrary deterministic state machines.

Secure multi-party computation (SMPC) [9] allows n players to compute an agreed function of their inputs in a secure way even when some players cheat. Although in theory it provides a foundation for achieving Byzantine fault-tolerant confidentiality, SMPC in practice can only be used to compute simple functions such as small-scale voting and bidding because SMPC relies heavily on computationally expensive oblivious transfers [15].

Firewalls that restrict incoming requests are a common pragmatic defence against malicious attackers. Typical firewalls prevent access to particular machines or ports; more generally, firewalls could identify improperly formatted or otherwise illegal requests to an otherwise legal machine and port. In principle, firewalls could protect a server by filtering out all "bad" requests from reaching it. An interesting research question is whether identifying all "bad" requests is significantly easier than building bug-free servers in the first place. The privacy firewall is inspired by the idea of mediating communication between the world and a service, but it uses redundant execution to allow it to filter mismatching (and presumptively wrong) outgoing replies rather than relying on *a priori* identification of bad incoming requests.

7 Conclusion

The main contribution of this paper is to present the first study to systematically apply the principle of separation of agreement and execution to BFT state machine replication to (1) reduce the replication cost and (2) enhance confidentiality properties for general Byzantine replicated services. Although in retrospect this separation is straightforward, all previous general BFT state machine replication systems have tightly coupled agreement and execution, have paid unneeded replication costs, and have increased system vulnerability to confidentiality breaches.

References

[1] A. Adya, W. Bolosky, et al. FARSITE: Federated available and reliable storage for incompletely trusted environments. In *5th Symp on Operating Systems Design and Impl.* Dec. 2002.

[2] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput.*, 37(4):418–425, 1988.

[3] Anonymous. Citation removed for double-blind review.

[4] —. Extended technical report. We will make an anonymized version of the report available to the program chair upon request., 2003.

[5] M. Baker. *Fast Crash Recovery in Distributed File Systems*. Ph.D. thesis, University of California at Berkeley, 1994.

[6] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementally at reduced cost. In *Eurocrypt97*. 1997.

[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. of the ACM*, 32(4):824–840, October 1985.

[8] M. Burrows, M. Abadi, et al. A Logic of Authentication. In *ACM Trans. on Computer Systems*, pp. 18–36. Feb. 1990.

[9] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. Ph.D. thesis, Weizmann Institute of Science, 1995.

[10] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Tech. Rep. 92-15, Dept. of Computer Science, Hebrew University, 1992.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd Symp. on Operating Systems Design and Impl.* Feb. 1999.

[12] —. Proactive recovery in a Byzantine-Fault-Tolerant system. In *4th Symp. on Operating Systems Design and Impl.*, pp. 273–288. 2000.

[13] P. Chen, W. Ng, et al. The Rio File Cache: Surviving Operating System Crashes. In *7th Internat. Conf. on Arch. Support for Programming Languages and Operating Systems*. Oct. 1996.

[14] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology: Crypto89, the Ninth Annual International Cryptology Conference*, pp. 307–315. 1990.

[15] S. Even, O. Goldreich, et al. A Randomized Protocol for Signing Contracts. *Comm. of the ACM*, 28:637–647, 1985.

[16] M. Fischer, N. Lynch, et al. Impossibility of distributed commit with one faulty process. *J. of the ACM*, 32(2), Apr. 1985.

[17] J. Garay, R. Gennaro, et al. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, Jul. 2000.

[18] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for n¿ 3t Processors in t+1 Rounds. *SIAM Journal of Compouting*, 27(1), 1998.

[19] J. Howard, M. Kazar, et al. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, Feb. 1988.

[20] A. Iyengar, R. Cahn, et al. Design and Implementation of a Secure Distributed Data Repository. In *Proc. of the 14th IFIP Internat. Information Security Conf.*, pp. 123–135. 1998.

[21] R. Karp, M. Luby, et al. Proactive Secret Sharing or How to Cope with Perpetual Leakage. In *Proc. of the 15th Annual Internat. Cryptology Conf.*, pp. 457–469. 1995.

[22] K. Kihlstrom, L. Moser, et al. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*. 1998.

[23] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, 1986.

[24] J. Kubiatowicz, D. Bindel, et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In *9th Internat. Conf. on Arch. Support for Programming Languages and Operating Systems*. Nov. 2000.

[25] S. Kumar and K. Li. Using model checking to debug device firmware. In *5th Symp on Operating Systems Design and Impl.* 2002.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[27] —. Part time parliament. *ACM Trans. on Computer Systems*, 16(2), May 1998.

[28] —. Paxos made simple. *ACM SIGACT News Distributed Computing Column*, 32(4), Dec. 2001.

[29] L. Lamport, R. Shostack, et al. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[30] B. Liskov, S. Ghemawat, et al. Replication in the Harp File System. In *13th ACM Symposium on Operating Systems Principles*. Oct. 1991.

[31] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pp. 203–213, 1998.

[32] D. Mazières, M. Kaminsky, et al. Separating key management from file system security. In *17th ACM Symposium on Operating Systems Principles*. Dec. 1999.

[33] M. Musuvathi, D. Park, et al. CMC: A pragmatic approach to model checking real code. In *5th Symp on Operating Systems Design and Impl.* 2002.

[34] M. Reiter. The Rampart toolkit for building high-integrity services. In *Dagstuhl Seminar on Dist. Sys.*, pp. 99–110. 1994.

[35] R. L. Rivest, A. Shamir, et al. A method for obtaining digital signatures and pulic-key cryptosystems. *Comm. of the ACM*, 21(2):120–126, 1978.

[36] R. Rodrigues, M. Castro, et al. Base: Using abstraction to improve fault tolerance. In *18th ACM Symposium on Operating Systems Principles*. Oct. 2001.

[37] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*. 2001.

[38] A. Sabelfeld and A. C. Myers. Language-based information-flow security.

[39] F. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial. *Computing Surveys*, 22(3):299–319, Sep. 1990.

[40] Secure hash standard. Federal Information Processing Standards Publication (FIPS) 180-1, Apr. 1995.

[41] G. Tsudik. Message authentication with one-way hash functions.

ACM Computer Comm. Review, 22(5), 1992.

[42] U. Voges and L. Gmeiner. Software diversity in reacter protection systems: An experiment. In *IFAC Workshop SAFECOMP79*. May 1979.

[43] M. Welsh, D. Culler, et al. SEDA: An architecture for well-conditioned, scalable internet services. In *18th ACM Symposium on Operating Systems Principles*, pp. 230–243. 2001.

[44] L. Zhou, F. Schneider, et al. COCA : A Secure Distributed Online Certification Authority. *ACM Trans. on Computer Systems*, 20(4):329–368, Nov. 2002.