Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture

Rodrigo Rodrigues and Barbara Liskov
MIT Computer Science and Artificial Intelligence Laboratory

Abstract

This paper presents Rosebud, a new Byzantine fault-tolerant storage architecture designed to be highly scalable and deployable in the wide-area. To support massive amounts of data, we need to partition the data among the nodes. To support long-lived operation, we need to allow the set of nodes in the system to change. To our knowledge, we are the first to present a complete design and a running implementation of Byzantine-fault-tolerant storage algorithms for a large scale, dynamic membership.

We deployed Rosebud in a wide area testbed and ran experiments to evaluate its performance, and our experiments show that it performs well. We show that our storage algorithms perform equivalently to highly optimized replication algorithms in the wide-area. We also show that performance degradation is minor when the system reconfigures.

1 Introduction

There is a growing demand for highly-available systems that provide correct service without interruption. In this context, Byzantine-fault-tolerant systems are becoming more and more important because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior.

This paper presents a novel Byzantine-fault-tolerant storage system called Rosebud. Rosebud is designed to store massive amounts of data. It stores this data at a large number of nodes connected through an unreliable wide-area network, and allows users to read and modify the data simultaneously from diverse geographical locations. It provides a storage utility that can be shared by applications, allowing them to inherit the good properties of Rosebud: security, robustness, extensibility, scaling, and load balancing.

A Rosebud deployment is likely to contain a very large number of storage nodes (e.g., tens of thousands). The nodes can be dedicated servers or a set of reliable nodes, e.g., commodity workstations in a corporation or large university, or federations of these. Rosebud stores each data item at a subset of the storage nodes, chosen so as to spread the load. Each data item is replicated at enough nodes to provide reliability and availability despite Byzantine failures (e.g., when a subset of a corporation's machines are compromised) and replicas are chosen so that they are likely to fail independently (e.g., they are geographically distributed).

A large deployment requires reconfiguration. Machines that break or are decommissioned must be removed from the system and their responsibilities assigned to non-failed nodes, to maintain reliability and availability of the data those nodes used to store. Also new machines must be added to the system as the need for increased storage or throughput dictates. Rosebud reconfigures itself when nodes join and leave the system, while continuing to provide service in the presence of such configuration events. Reconfiguration requires minimal human intervention; thus we avoid the human errors that have been shown to be a major cause of disruption in computer systems [16, 31].

The design of Rosebud posed two hard challenges and our approaches to solving these problems are the main contributions of our research.

First, we introduce a novel membership protocol that allows participating nodes to agree on the system membership despite arbitrary faults. Unlike group communication systems [18, 32], our approach scales to systems containing thousands of nodes, because we run a Byzantine agreement protocol only at a subset of the nodes. Periodically, these nodes output a description of the system membership, authenticated using a proactive threshold signature protocol. This configuration is disseminated to all nodes, which allows them to agree on the set of replicas for each data item.

Second, we developed new Byzantine-fault-tolerant storage algorithms that work in a dynamic setting where the set of replicas for a particular data item changes as the system is running. Getting storage algorithms to work in a dynamic setting is still an active field of research, even for the simpler fail-stop failure model [24, 25]. Our algorithms provide strong storage semantics–they ensure atomic reads and writes [21] in spite of configuration changes, i.e., when the nodes responsible for storing a data item change, and Byzantine-faulty replicas. Yet, these algorithms were designed to be highly efficient, especially in a wide-area network deployment. This is achieved by minimizing the number of round-trips to complete an operation, and by not putting any single replica in the critical path of an operation, since this would cause the performance to degrade when that replica is behind a slow or distant link.

We have implemented our system and deployed a prototype on a wide-area testbed. This paper presents results of experiments showing that our protocols are efficient: the fastest clients read a 1 megabyte file stored on geographically diverse Rosebud servers in under a second. We also compare Rosebud's storage algorithms with a highly optimized Byzantine fault tolerance algorithm [11] and conclude that Rosebud performs equivalently to BFT in the wide-area, and its performance degrades more gracefully when some of the nodes are behind distant or slow links.

To our knowledge, Rosebud is the first system to provide a complete solution and implementation for large-scale, Byzantine-fault-tolerant storage that supports membership changes. Most previous work on Byzantine-fault-tolerant systems assumes that each replica holds an entire copy of the service state (e.g., [11, 27]). These systems provide no automatic way to replace a failed node or to react to overload: they are unable to automatically add more components and offload some of the work to them. Group communication systems (e.g., [32, 18]) automatically adjust to nodes joining and leaving the system, but they do so by executing agreement between all nodes in the system, which is too expensive to be used with a large number of nodes, and they do not provide storage solutions.

Farsite [1] and Oceanstore [19] share our goal of providing shared storage and automatic reconfiguration. In fact, both systems recognize the importance of providing Byzantine-fault-tolerant storage in a dynamic environment. However, neither system presents a complete design for how the Byzantine fault tolerance algorithms need to be modified to work in a dynamic setting. This paper is the first to present a complete solution and a working implementation that solve all these problems.

The remainder of the paper is organized as follows. Section 2 presents our assumptions. Section 3 provides an overview of our design and explains the key decisions that underlie it; Sections 4 and 5 provide the details. Section 6 discusses correctness, and Section 7 presents our performance results. Section 8 discusses related work. We conclude with a discussion of future directions.

2 System Model and Assumptions

This section presents the assumptions that underlie Rosebud's design.

We assume an *asynchronous* distributed system where nodes may operate at different speeds and there are no bounds on message delays. We assume the network may fail to deliver messages, corrupt them, delay them, duplicate them, or deliver them out of order. More specifically, our algorithms do not rely on synchrony to provide correctness. To ensure liveness, we must rely on some weak partial synchrony assumption like eventual time bounds [23].

Rosebud uses a Byzantine failure model — faulty nodes may behave arbitrarily. We assume a powerful adversarial model where an attacker can coordinate faulty nodes in arbitrary ways; yet there are bounds on the number of simultaneous faulty processes, described in Section 6.

To authenticate communication in the presence of Byzantine faults, we rely on cryptographic techniques that are hard to subvert. We assume each node in the system has a public key that speaks for its principal. A non-faulty node can use digital signatures to prevent an adversary (or the network) from undetectably modifying network communication from that node. We also assume the existence of a collision-resistant hash function, and a proactive threshold signature protocol that adversaries cannot subvert.

We assume a very low rate of malicious subversion, and a low to moderate rate of failstop failures. This assumption is justified by real-world observations [1]. The failstop rate will vary depending on the deployment: it will be smaller for dedicated servers than for workstations.

Nodes in Rosebud are geographically distributed, therefore are separated by high latency links. This is not only a likely characteristic of expected deployments, but is also needed for high availability, since geographical diversity is the only way to survive catastrophes.

3 System Architecture Overview

This section describes and motivates our design. A key aspect of our architecture is the separation of configuration management from the storage service. The two modules have different requirements, thus we used different approaches to their design.

3.1 Configuration Management

The configuration management module is responsible for monitoring the arrival and departure of nodes, and propagating information about the current configuration (i.e., a description of the system membership) to the other nodes such that all nodes have a correct view of the current configuration. The hard problem is to do this in a way that scales to thousands of nodes and is secure.

Configuration management is done by a logically-centralized entity called the *configuration service* (CS). The centralized solution offers two main advantages over a fully decentralized one [10]: it is efficient, and it makes it easier to offer strong consistency and prove properties than a decentralized solution, where different participants may have different views of the current configuration at the same instant, possibly leading to incoherent views of the same data.

Of course, the CS must be fault-tolerant, even to malicious attacks: We cannot trust an individual node to be correct and make the right decision about adding or removing nodes from the current configuration. Therefore the CS is implemented as a group of replicas that carry out the BFT agreement protocol [11]. To increase the degree of fault-tolerance of the CS we set the parameter that defines the maximum number of faulty replicas in the group, f_{cs}, to a high value, e.g., 4 or 5. Running this protocol on a small subset of the nodes provides the required scalability, unlike membership protocols in which all nodes try to agree among themselves [32, 18].

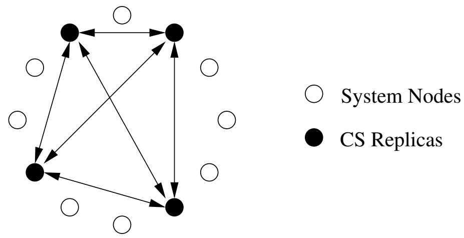

Figure 1: System Architecture. At each moment, a small subset of the nodes in the system are also CS replicas. These nodes execute an agreement protocol to determine the system membership.

The CS runs on a subset of the nodes as illustrated in Figure 1. It runs on different subsets at different times, making it difficult for an adversary to disrupt the system by targeting its replicas. (The system could be trivially modified to run the CS on a separate set of nodes, e.g., nodes with hardened security.)

The CS has to keep track of the current configuration, and propagate it to the other nodes in the system. It must handle requests to add and remove nodes; in addition it monitors node reachability and automatically removes unreachable nodes.

Nodes can only be added by a trusted entity (e.g., a principal) that vouches for the new node. This is necessary to avoid a Sybil attack [13] where an adversary floods the system with malicious nodes. The CS knows the public key of this principal; the associated private key, which we assume is hard to compromise, is used to authenticate a request to add a node to the system. The system can easily be extended so that this principal can delegate trust to others.

The CS monitors nodes and expels those that do not respond to probes. This removes unreachable nodes, but is insufficient for Byzantine-faulty nodes for a simple reason: it is impossible to recognize such nodes. A Byzantine-faulty node can appear to behave properly, e.g., answer all client requests correctly, even if it has been compromised. Therefore obtaining a proper reply from a node does not imply it is correct. This means an adversary can launch a *lying-in-wait attack*: it compromises more and more nodes, making them behave properly until enough have been compromised so that the correctness conditions of the system are broken. Then the nodes launch an attack simultaneously, causing the system to malfunction.

Therefore, we rely on an external mechanism to detect Byzantine-faulty nodes and we require revocation. Byzantine fault detection is still an open research problem, and for the sake of brevity we will not address this issue.

The system moves in a succession of time intervals called *epochs*, and we batch all configuration changes at the end of the epoch. This provides several advantages: it simplifies the task of propagating configuration changes, and it avoids the problem of different nodes having different views of the system configuration. The only potential problem is that the system takes longer to react to failures. However, a *delayed response to failures* is advantageous: it avoids spurious data transfers (e.g., removing nodes that are just temporarily disconnected); avoids avenues for attacks (e.g., a denial of service attack that slows down a large number of good nodes and removes them from the system); and avoids thrashing, where in trying to recover from a host failure the system overstresses the network, which itself may be mistaken for other host failures, causing a positive feedback cycle.

Setting the duration of an epoch, t_{epoch} is a complex choice since we want to obtain the advantages of delayed response to failures without causing data unavailability. This choice depends on the desired availability, amount and type of data redundancy, and distribution of node lifetimes; these issues are discussed in [6]. We expect t_{epoch} to be on the order of a few hours.

At the end of each epoch, the CS outputs a description of the system membership, authenticated using a proactive threshold signature protocol [15]. This allows nodes to validate the current configuration using a fixed, well-known public key. This configuration is disseminated to all nodes, which allows them to agree on the set of replicas for each data item.

3.2 Storage Service

The storage service module provides applications with read and write operations on objects in a flat namespace. Object IDs are chosen in a way that allows the data to be self-verifying. Rosebud provides access to two types of objects:

Content-hash objects are immutable: once created, a content-hash object cannot change. A content-hash object is identified by a hash of its contents.

Public-key objects are mutable and contain a version number that is used to determine data freshness. The ID is a hash of the public key used to sign the object. A public-key object includes a header with a signature that is used to verify the integrity of the data, and that covers the version number. Public-key objects provide access control (for writing); the scheme could be extended to allow principals with different public keys to write the same object, by identifying the object by a hash of the public keys of all possible writers, and indicating in the object which key was used in the signature.

Public key objects are atomic [21]. This means that all operations on an object appear to execute in some sequential order that is consistent with the real-time order in which operations actually execute. Content-hash objects have weaker semantics: they provide sequential consistency [20], ensuring that once the operation that created the object completes, that object is returned by subsequent reads. Having weaker semantics for these objects is a conscious decision, given how we expect them to be used: typically a user publishes their IDs (e.g., using a public key object) only after their write completes. Some applications may require only sequential consistency for mutable objects; we could provide such a category with a simple modification to the algorithms described in Section 5.

Public-key objects can be deleted (we perform this by overwriting objects with a special *null* value of negligible size), but content-hash objects cannot (since they are immutable). Content-hash objects should be garbage-collected after they are no longer useful. We are working on such a scheme, but for the sake of brevity we do not discuss it here. (A preliminary design appears in [5].)

3.3 Object Placement

A key issue is how to partition the objects among the storage nodes. To solve this problem we use consistent hashing [17]. Each node is assigned a *node ID* of size m, and these identifiers are ordered in an identifier circle modulo 2^m. Rosebud stores objects with ID i at the first n nodes whose identifiers are equal to or follow i in the identifier space (called the *successors* of ID i); this is how nodes are selected in Chord [36] but any other deterministic selection technique would also be acceptable.

Using consistent hashing has several advantages. It has good load-balancing properties (all nodes are responsible for approximately the same number of objects); it is easy for a node to verify whether it is responsible for a data item; and, most importantly, the work needed to accommodate membership changes is small: only a small number of nodes is involved in redistributing the data.

4 Configuration Management

The CS is implemented as a BFT [11] replica group. Its two main tasks are described in Section 4.1 (computing configurations) and Section 4.2 (propagating this information to the others nodes). Section 4.3 explains how we pick nodes to be CS replicas, and Section 4.4 explains how storage nodes and clients join the system.

4.1 Computing Configurations

The CS is the sole entity responsible for computing configurations. This section describes how this is done despite faulty CS replicas. Recall that our system allows nodes to be explicitly added and removed. In addition, the CS monitors nodes and evicts ones that are unresponsive.

The CS must prevent an adversary from flooding the system with malicious nodes. It does this by requiring a trusted party to use its private key to vouch for new nodes [13]. It also allows membership to be revoked, again by a trusted party. The format of these operations is as follows:

⟨ADD, *IP address, port number, public key*⟩

⟨REVOKE, *IP address, port number*⟩

where *IP address, port number* is the network address of the new node, and *public key* is the public key of the node, assigned by the trusted party. In our current implementation we support only explicit revocation, but it would be easy to add implicit revocation (where node memberships time out) if desired.

Nodes cannot choose their own IDs, since otherwise an attacker could compromise the system by controlling a few nodes [10]. Therefore, the CS generates node IDs, using a deterministic computation based on the node's address and public key, and the current configuration.

The CS monitors the nodes to identify failed or unreachable nodes. Each CS replica does its own monitoring so that it can form its own view of which nodes are faulty; then it will be able to decide whether a configuration change proposed by another replica is reasonable.

The CS monitors nodes by sending them pings. At present it sends pings at a constant rate; we plan to increase the frequency of pings to unresponsive nodes. Infrequently, pings contain nonces that must be signed in the reply, to avoid an attack that spoofs ping replies to maintain unavailable nodes in the system. Signed pings are used sparingly since they require additional processing on the CS nodes to verify signatures. However, once a node fails to reply to a signed ping, all subsequent pings to that node must request signatures (until a correctly signed response arrives).

The results of the pings are inserted in a *liveness database* local to the CS replica. After pings time out, a CS node queries its liveness databases to decide if the node should be evicted from the system. To evict the node, it must get the other CS nodes to agree, and this is not trivial, since different nodes will have different values for how long each node has been unreachable.

CS nodes initiate the node eviction process if n_{evict} consecutive pings for that node fail. In this case, that CS replica proposes the eviction to other CS replicas. Then, it must collect signed statements from at least $2f_{cs} + 1$ CS replicas (including itself) that agree to evict that node. Other CS replicas accept the eviction (and sign a statement saying so) if the last n_{accept} pings for that node have failed (according to their local database), where $n_{accept} < n_{evict}$. This approach ensures that most eviction operations will succeed.

The eviction threshold n_{accept} is chosen to avoid spurious evictions: the time it takes to send n_{accept} pings should be longer than an appropriate membership timeout to avoid evicting nodes that are temporarily unreachable, as discussed before. However, making this limit too large may result in decreased availability of the data. A possible choice is to cycle through each node in the system every 15 minutes, and to choose $n_{evict} = 5$, and $n_{accept} = 3$.

After collecting the statements, the CS replica that initiated the process invokes the following operation on the CS:

⟨EVICT, *node ID*, $\langle\sigma_1, ..., \sigma_{3f_{cs}+1}\rangle$⟩

where the vector $\langle\sigma_1, ..., \sigma_{3f_{cs}+1}\rangle$ contains at least $2f_{cs}+1$ signatures from current CS replicas agreeing to evict the node. The operation will fail if there are not enough signatures or they do not verify.

4.2 Propagating New Configurations

The CS needs to convey membership changes to other nodes. It does this periodically, at the end of each epoch. The first problem we need to address is how to make CS replicas agree on when to end the current epoch. Later in this section, we address what information is propagated and how it is propagated.

Each configuration lasts for the duration of an epoch, t_{epoch}. Each CS replica keeps track of how long it believes the current configuration is valid, and after t_{epoch} seconds it invokes the $\langle$STOP$\rangle$ operation on the CS. This operation halts monitoring and produces the configuration for the next epoch based on the lists of nodes that have been accepted or evicted so far.

To avoid a faulty CS replica stopping the service too early, we use the non-deterministic choices validation mechanism proposed in [34]. This allows the primary BFT replica to propose a non-deterministic value for each operation. In this case, the primary proposes the value for how long the current epoch has been running. Other replicas verify the choice, in this case by checking whether it is within a small delta of their own values (e.g., a few seconds). The operation will fail if it is invoked earlier than t_{epoch} minus that fixed delta.

Note that we do not assume synchronized clocks, nor synchronized clock rates. However, if there are more than f_{cs} CS replicas with a slow clock rate, this will cause the STOP operation to be executed later, causing an increase in a time window where certain correctness conditions must be met (we present correctness conditions in Section 6).

After the STOP operation is executed, all CS replicas agree on the next configuration: nodes for which EVICT and REVOKE operations have been executed are removed and those for which ADD operations have been executed are added.

This information must then be propagated to the other nodes. We could do this by partitioning the information and sending each node only what it needs to carry out its base algorithm. This approach is frequently used in peer-to-peer overlays; e.g., in a system based on Chord [36], nodes would be given their successors and fingers. But incomplete information is a problem since it leads to multiple hops for routing. Instead we disseminate the entire configuration to all the nodes so that routing decisions can be made locally, avoiding multi-hop latency, and additional complexities (e.g., secure routing [10]).

There are two costs of concern here. The first is the memory usage, but this is small even at large scale. If we assume that the configuration consists, for each node in the system, of a 160-bit node identifier (based on a SHA-1 cryptographic hash function), plus its IPv4 address, port number, and 1024 bit RSA public key, then the entire configuration for a system of $100,000$ nodes will fit in approximately 14.7 megabytes. This information fits in main memory of today's commodity workstations.

The second cost is the communication cost, but this is also small if we use deltas. Thus, the new configuration description is encoded as follows.

$\langle$*epoch number, add list, drop list,* $\sigma_{cs}\rangle$, where
add list = list of $\langle$*node ID,IP address,public key*$\rangle$,
drop list = list of $\langle$*node ID*$\rangle$,
and σ_{cs} is a signature certifying that the CS produced the configuration. The signature is over the epoch number and the entire configuration (not just the delta): nodes produce the new configuration from their old one and the delta and then check the signature.

To produce the signature we use a proactive threshold signature scheme [15]. When epochs change, the set of CS replicas also changes (as described in Section 4.3), and the new replicas obtain new shares from the old replicas. These shares allow the next CS to sign the next configuration, and that signature to be verified by any node in the system with the same, well-known public key. Nonfaulty replicas discard old shares after the epoch transition completes.

To propagate the new configuration efficiently without overloading CS replicas, we use a three-level multicast tree. To build the tree, CS nodes send a level 0 message to the nodes in the next configuration that succeed it by $i \cdot \sqrt[3]{N^2}$ nodes, where N is the number of nodes in that configuration, and $i = 0, ..., \sqrt[3]{N} - 1$. Nodes receiving level 0 messages send level 1 messages to the nodes that succeed it by $i \cdot \sqrt[3]{N}$, again with $i = 0, ..., \sqrt[3]{N} - 1$. Finally, nodes receiving level 1 messages send level 2 messages to their $\sqrt[3]{N}$ immediate successors.

This scheme could work if only one CS replica initiated it, but we force all replicas to perform this for fault tolerance. The scheme could be generalized to more than three levels, but having three levels renders a nice bound on the number of messages each process sends. Note that if some node misses the message, it can obtain the delta from any node that claims to have upgraded to a new epoch.

Clients also maintain a complete configuration description, to determine which nodes to contact to store or retrieve data items. Clients learn the configuration when they join (see Section 4.4); after that they usually obtain configuration deltas from storage nodes they interact with (see Section 5).

4.3 Configuration Service Replicas

Another issue we need to solve is how we decide on what nodes act as the replicas of the configuration service. Recall that the configuration service is superimposed on the system nodes. As discussed earlier, superimposing can provide good security: it allows us to move the CS in each epoch as a way of avoiding an attack directed at a particular set of replicas. We opted not to have any bias when choosing CS replicas (e.g., towards faster or better connected machines) since this might represent an avenue of attack: the attacker might make an effort for his or her machines to be good candidates as a way to gain control of the CS.

We need a way of picking CS replicas that periodically changes these replicas. We achieve this by choosing CS replicas to be the nodes whose IDs follow (or equal) $h(i, e, h(c))$, where $i \in \{1, ..., 3f_{cs}+1\}$ is the replica number, e is the epoch number, and $h(c)$ is a hash of the configuration for epoch e. Adding the configuration for that epoch causes the CS to move in an unpredictable way, thus preventing an attacker from targeting replicas in advance.

There is a further issue concerning moving the CS: how do new replicas learn the state of the old CS? Our solution is to avoid having state transfer from the old CS to the new one. Instead, the new CS starts with a fresh state: it only knows the new configuration, just like every other node in the system. This means that all the ping information gets reset, which is fine given our goal of delayed response to failures; note that unresponsive nodes will be evicted in the new epoch.

4.4 Joining the System

The request to add a storage node to the configuration for the next epoch must be initiated by the trusted authority that executes the respective CS operation. We assume that the node operator contacts this authority via an out-of-band mechanism to request this.

When a client or storage node joins the system, it must obtain a complete and up-to-date configuration description. We assume the existence of an out-of-band mechanism that allows joining nodes to learn the public key of the system and the location of one (or a few) current nodes. The joining node then reads the entire configuration from one (or preferably more) nodes.

There are two techniques to reduce the latency of downloading large configurations. Newly added storage nodes can download the entire configuration when they join; then in the next epoch, when they first become active, they only need to receive the delta. For new clients, we can use Merkle trees [29] to allow the client to control the order in which it receives information, e.g., a client can learn about nodes that store a particular object first. Merkle trees can also be used by a reconnecting client to identify the minimum information that needs to be transmitted.

Each configuration description contains a certificate that vouches for its correctness. But it is still possible that the configuration is old, and if it were old enough, it might contain enough failed nodes that using it would not produce correct results. Therefore the joining node needs to determine whether the configuration is up to date. It does this by contacting the CS replicas, issuing a random nonce as a challenge. The CS replicas respond by signing the random nonce with the threshold signature scheme. But such a response is possible only if the CS is current; a stale CS could not produce such a signature, since nonfaulty replicas discard their shares after the epoch ends. The signature can be verified using the well-known public key that was obtained in advance. If Merkle trees are employed to download less information, then we must extend the configuration certificate to contain the identity of CS replicas, to allow new clients to issue this challenge.

We also require clients to periodically (but infrequently) issue this challenge to the current CS, in order to verify that the configuration they hold is up-to-date. Clients perform this check every $T_{contact}$ units of time (e.g., 1 day). If the CS does not sign the nonce and reply for longer than $\delta_{contact}$ the client halts execution and waits until it hears from the CS. This avoids a situation where a slow client would not receive the most up-to-date configuration and would perform operations on a stale replica group where the fault threshold had been exceeded.

5 Storage Algorithms

This section describes the algorithms used to store and retrieve data. Recall that Rosebud uses consistent hashing [17] to determine which nodes are responsible for storing a data item, and that every node in the system maintains a complete configuration description. Therefore clients can contact the nodes that are storing the data directly.

The algorithms described in the remainder of this section were guided by two design principles: minimizing the number of network round trips; and not putting any single replica in the critical path of an operation. This makes our algorithms efficient in a wide-area deployment where possibly some, but not all, of the replicas are behind slow, congested, or distant links.

We begin this section by describing how the algorithms work assuming a static system. Section 5.2 explains what happens when the configuration changes. Section 5.3 discusses garbage collection of data items that nodes are no longer responsible for, and we conclude in Section 5.4 with a discussion of when to discard old configurations.

Recall that when the system is correct it provides atomicity for public key objects, and sequential consistency for content-hash objects. The conditions required for the system to be correct are detailed in Section 6.

5.1 Static Case

Different algorithms are used to read and write the two types of objects.

Content-hash Objects

Read and write operations for content-hash objects complete in a single network round-trip: clients contact the replicas for the object, and the operation succeeds when they receive a reply from a sufficiently large subset of these replicas, usually called a *quorum*.

Content-hash objects are replicated at $n = 3f + 1$ nodes, where f is the maximum number of (arbitrary) faults we tolerate in each replica group. The write algorithm must ensure that $2f + 1$ replicas claim to have stored the data. The reason why we do not wait for the remaining f replicas is that they may be faulty and so may never respond. Therefore, we need to make progress after hearing from only $n - f = 2f + 1$ replicas.

Thus, the write request must be sent to at least $2f + 1$ replicas (but for efficiency and increased availability it is sent to all $3f + 1$ replicas). When replicas receive the request, they check its integrity by verifying that the ID is a hash of the content, and then send an ack to the client. The ack contains a signature of the ID to avoid spoofing. The client waits for acks from $2f + 1$ replicas with valid signatures; clients know the public keys of all nodes since this is part of the configuration information. If after a timeout the client has not received enough replies, it retransmits the request and sets another timer.

Content-hash data can be read from just one replica, as long as the client verifies that the hash of the contents matches the ID. To select a fast node for downloading the object, we send an initial small request to all nodes, and then request all remaining fragments of the object from the first replier. This approach also provides load balancing for popular content: overloaded servers will be slower to reply and thus less likely to send the entire object. If after a timeout the complete object has not been received, the client repeats the request in the non-optimized mode (i.e., fetching the object from all the replicas).

It may seem that $2f + 1$ replicas would suffice for content-hash objects. We explain why we need $3f+1$ replicas when we address state transfer.

Public key objects

The protocols for public key objects are similar to existing Byzantine quorum protocols [27] (with the exception of several optimizations we propose). The main contribution of our storage algorithms is that we extended these to work in a dynamic setting (Section 5.2).

Public key objects have a header that includes a version number assigned by the writer (in such a way that distinct writers always pick distinct version numbers, e.g., by appending node IDs) and the writer's signature for the object and version number. We use sets of $n = 3f + 1$ replicas and quorums of $2f + 1$.

The *write* protocol has two phases. In the read phase, a quorum is contacted to obtain a set of version numbers for the object. Then the client picks a version number greater than the highest number it read and performs a write phase where it sends the new signed object to all replicas and waits until it hears a reply from a quorum. The client sends a random nonce with both requests, and this nonce is signed together with the current version number in the reply, to prevent replay attacks.

Replicas receiving a write verify that the content of the object, along with the version number, match the signature in the object's header. If this verification succeeds, the replica replies to the client, but only overwrites the object if the new version number is greater than the one currently stored.

The read phase can be omitted in two situations. First, if there is a single writer (as in some existing applications, e.g., Ivy [30]), the writer can increment the last version number it wrote and use it in the write phase. Second, if clients use a clock synchronization protocol where clock skews are smaller than the time to complete an operation (e.g., if the machines have GPS), they can use timestamps for version numbers.

In the *read* protocol the client requests the object from all replicas. Normally there will be $2f + 1$ replies that provide the same version number, and in this case the result is the correct response and the read operation completes. However, if the read is occurring concurrently with a write, the version numbers may not agree. In this case, there is a second phase in which the client picks the object with the highest version number, writes it to all replicas, and waits for a reply from a quorum. (Again, nonces are employed to avoid replays, and the reply in the first phase must sign the nonce and the current version number.)

We use the same optimization as in the content-hash reads, where a small request for the signed version number is sent to all replicas, and the entire object is downloaded from the first replier with the highest version number. (After a timeout expires, the client reverts to fetching from all replicas, and waits until it gets $2f$+1 complete objects.)

Faulty Clients

The semantics provided by the system in the presence of faulty clients depend on whether the clients fail silently (crash failures) or arbitrarily (Byzantine failures).

For crash failures, the problematic situation is when the client crashes after writing to a subset of the replicas. This situation is corrected as soon as a non-faulty client reads from one of the replicas in that subset during the read phase of any operation.[1]

Byzantine-faulty clients can do arbitrarily bad things, except that, of course, they must have the right to modify the object, i.e., they have to know the private key. They can write garbage to the object, select a very large version number and thus exhaust the version number space, or they can send different objects with the same version number

[1]We can claim that the system still provides atomic semantics in this case, provided we change the definition of an operation by a faulty client to conclude only when the first operation that reads the value written by the faulty client concludes.

to different replicas. Our system does not prevent these problems, although in the latter case the problem will go away the next time a non-faulty client overwrites that object. This is a conscious decision, since maintaining atomicity in the presence of Byzantine-faulty clients is expensive, and a faulty client could still write garbage or constantly modify the data to produce confusing results.

5.2 State Transfer

At epoch boundaries, storage nodes receive an authenticated description of the next configuration from the CS or via gossip. At this point the node *enters the new epoch*: it starts to act as determined by the new configuration.

In the new epoch, the set of nodes storing certain objects may change. Each storage node must identify objects that it is no longer responsible for and refuse subsequent requests for them. Each node also identifies objects that it is now responsible for and fetches them from their previous replicas (in the earlier epoch). All nodes have access to both old and new configurations when they change epochs and therefore know which nodes to contact.

The node sends state transfer requests to all the old replicas requesting all objects in a particular interval in the ID space (these replicas enter the new epoch at that point if they have not done so already). In practice, this is done in two steps, with a first message requesting the IDs of all objects in the interval, and a second step to fetch the objects using the read protocols described above, except that, unlike normal reads, state transfer does not need to write back the highest value it read. This is so because the only way that the new set of replicas will read different values from state transfer is if some of them read an incomplete write. However, the client read protocol ensures that the first client to read an incomplete write propagates its value to a large enough quorum to ensure atomicity, so that the property will still hold after state transfer.

State transfer is the reason we require $3f+1$ replicas for content hash objects. For client use, $2f+1$ replicas are sufficient because clients request these objects only when they know they exist. Therefore if a client received $f+1$ replies denying knowledge of the object (these replies would come from a combination of faulty replicas and replicas that didn't hear about the creation of that object), it would know to wait for more replies. However, when state transfer is occurring the new responsible node doesn't know what objects exist; therefore if it received $f+1$ replies that indicated the object did not exist, it would not know what to do. Another point is that if content hashed objects can be garbage collected, the same problem will arise for clients.

Nodes cannot act on client requests that are for the wrong epoch since this could lead to an atomicity violation. Client requests contain an epoch number; the replica rejects the request (and sends the current configuration information) if the epoch is stale. The client will then retry the request in

```
Write (id,o)
  Send ⟨READ, id, epoch_c, nonce⟩ message to all replicas
  repeat Collect replies in a quorum Q until ValidateReplies()
  Choose a version ver_new greater than the highest version it read
  Send ⟨WRITE, id, ver_new, o, nonce⟩_σc to all replicas
  repeat Collect replies in a quorum Q' until ValidateReplies()
  return

Read ()
  Send ⟨READ, id, epoch_c, nonce⟩ message to all replicas
  repeat Collect replies in a quorum Q until ValidateReplies()
  Choose the reply with maximum version, ver, and the corresponding object o
  if (all replies agree on ver)
    return o
  Send ⟨WRITE, id, ver, o, nonce⟩_σc to all replicas
  repeat Collect replies in a quorum Q' until ValidateReplies ()
  return o

ValidateReplies ()
  if (some reply is not signed correctly)
    Remove reply from set
    return false
  if (some reply is in the form ⟨ERR_NEED_CONFIG, nonce⟩_σi
    Send replica i the current configuration
    Remove reply from set
    return false
  if (some reply is in the form ⟨ERR_UPGRADE_CONFIG, next_config⟩)
    verify the authenticity of next_config, upgrade configuration and
    restart current phase
    return false
  return true
```

Figure 2: Client algorithm pseudocode (for PK objects)

the new epoch if necessary. Each individual phase of an operation can complete at the client only when all $2f+1$ replies come from replicas in the same epoch (even though the read and write phases can complete in distinct epochs). This a key aspect of our algorithm. If an individual phase used results from different epochs, we could not ensure the intersection properties that make the operations atomic.

The epoch number sent by the client might be larger than that at the replica. In this case, the replica replies that it has not upgraded yet, and the client can push the configuration to that replica before retrying (if necessary).

The pseudocode for the complete read and write protocols is presented in Figures 2 and 3.

5.3 Garbage Collection of Old Data

Replicas must delete objects they are no longer responsible for to avoid old information accumulating forever. In this section we discuss when it is safe to delete old data.

A new replica sends an ack when it receives a valid response to a state transfer request. An old replica counts these acks and deletes the object once it has $2f+1$ of them. It explicitly requests acks after a timeout to deal with losses of the original acks.

After it deletes an object, the old replica might receive a state transfer request from a new replica (one it had not heard from previously). In this case it sends a special null reply indicating that it no longer has the object in question. If some responses contain the object, it is set as discussed above, just considering the non-null replies. If all (valid) responses are null replies, the replica forgets about a content-

```
Recv (⟨READ, id, epoch, nonce⟩)
  if (ValidateRequest())
    reply ⟨READ-REPLY, val(id), version(id), nonce⟩_σi to client

Recv (⟨WRITE, id, ver, o, nonce⟩)
  if (ValidateRequest())
    reply ⟨WRITE-REPLY, nonce⟩_σi to client
    if ver > version(id) and o.ver = ver and o.signature.verify()
      version(id) = ver
      value(id) = o

ValidateRequest ()
  if (epoch < current_epoch)
    reply ⟨ERR_UPGRADE_CONFIG, next_config⟩ to client
    return false
  if (epoch > current_epoch)
    reply ⟨ERR_NEED_CONFIG, nonce⟩_σi to client
    return false
  return true
```

Figure 3: Server algorithm pseudocode (for PK objects)

hash object, but stores a null value with a special lowest version number for a public-key object. In the latter case, it will obtain the proper value of the object at some later time (which a client using the object writes to it).

The above protocol can be improved in the common case of new and old replica groups having members in common. Such a replica, R, sends replicas in the old group a list of IDs of objects it already stores, plus the version numbers of the public-key objects. The receiver returns the IDs of missing objects, and also of public-key objects whose version numbers are too small; it also counts that replica as knowing all objects that matched. Then R only needs to fetch the objects listed in the reply.

5.4 Removing Old Configurations

The algorithm described above maintains two configurations at each replica: the current one, and the previous configuration, which is used to execute state transfer. This raises the question of how to deal with a slow node that is lagging behind by several epochs, i.e., is it is safe for replicas to discard the configuration for epoch $e-1$ when they receive the configuration for epoch $e+1$?

Our solution is to remove a node from the system if it skips an epoch: if the node has not entered epoch e, it will be removed by the CS at the end of epoch e and therefore will not be a member of epoch $e+1$. This is implemented by having a node reply to a ping only if its came from a CS replica of the most recent configuration it knows, and that node has completed state transfer from the previous epoch. We expect it to be unlikely that in a real deployment nodes will skip entire epochs, given that epochs are reasonably long, and network outages eventually get repaired.

The alternative solution of garbage collecting configurations only when they are not needed [25] is not attractive with malicious nodes. A Byzantine-faulty replica could exploit this by pretending to have not heard about a large number of epochs, which would lead to unbounded storage requirements at nonfaulty replicas.

6 Correctness

This section describes the semantics provided by the system, and states correctness conditions under which the system provides those semantics. Due to space constrains, we defer the safety proof to a separate technical report [4].

6.1 System Semantics

As mentioned, when the system is correct it provides *atomic* semantics [21] for public key objects and sequential consistency [20] for content-hash objects.

We could also provide atomic semantics for content-hash objects; the key is the writing back done by read operations. But we decided not to do this because it may slow down reads and seems unnecessary in light of how content-hash objects are used: typically the client publishes their IDs only after their write completes. Our approach is sufficient to provide atomicity if content-hash objects are used this way.

Our system provides clients with access to data provided they can communicate with a quorum of replicas in the current configuration. This means that clients cannot access data if they are unable to communicate with a sufficient number of replicas. This could happen in the case of a network partition. It is not possible to tolerate partitions and offer strongly consistent and available data at the same time [14]. Since our system is aimed at providing consistency, we cannot also tolerate partitions.

6.2 Correctness Conditions

This section defines conditions for the system to be *safe*. Liveness conditions are outside the scope of this paper. Recall that the algorithm intentionally only provides the desired semantics for non-Byzantine-faulty clients, as mentioned in Section 5. Thus, these conditions only refer to bounds on the number of Byzantine-faulty replicas.

Correctness is based on the assumption that no more than f failures occur in the replica group for as long as that group is "needed". Obviously a group is needed while all its members are in the current epoch. But groups from older epochs are also needed until their objects have been copied to the new responsible replicas. Thus we define the following correctness condition.

Correctness condition C1: *For any replica group G_e for epoch e, G_e contains no more that f faulty replicas between the moment when the first non-faulty replica in G_e enters epoch e, until the last non-faulty replica in epoch $e+1$ finishes state transfer for any data that was held by G_e.*

We also need to define similar conditions for the CS.

Correctness condition C2: *The CS replica set must contain no more than f_{cs} faulty replicas between the moment when the first correct CS replica enters the epoch that corresponds to that particular CS, until the moment when last non-faulty CS replica for the next configuration holds that* *configuration description and the new shares for the CS threshold signature.*

This definition is not complete due to the slow client problem that was mentioned in Section 4: A correct, but slow client can have a stale configuration and contact an old replica group, g, long after the group has finished transferring state to the subsequent epoch. In the meanwhile, the number of faults in the group may have exceeded f, which suffices for the group to forge a stale reply to the client.

Recall that we solve this using a periodic check (every $T_{contact}$ units of time, plus an additional $\delta_{contact}$ to wait for a reply) that the CS is still active. After adding this behavior to the algorithm, we obtain the following additional correctness condition.

Correctness condition C3: *Any replica group G_e for epoch e contains at most f faults between the moment when epoch e starts and $T_{contact} + \delta_{contact}$ after epoch e ends (i.e., after CS replicas for epoch e stop being able to produce a shared signature).*

7 Experiments

We implemented a prototype for Rosebud based on the code for the DHash peer-to-peer DHT built on top of Chord [36]. Inter-node communication is done over UDP with a C++ RPC package provided by the SFS toolkit [28]. Our implementation uses the 160-bit SHA-1 cryptographic hash function and the 1024-bit Rabin-Williams public key cryptosystem implemented in the SFS toolkit. Objects are stored in a Berkeley DB3 database. The CS is implemented as a BFT service using the publicly available BFT/BASE code [34]. The implementation is complete with the exception of the proactive threshold signature protocol (temporarily replaced by a vector of signatures from CS replicas). We are studying efficient share refreshment protocols that work in asynchronous systems.

To demonstrate the practicality of the design, we present three sets of experiments. The first is a controlled experiment that uses a small number of machines on a local area network, and focuses on determining the overhead of public key cryptography and quorum replication. The second is a wide-area experiment using 88 nodes in 22 locations distributed over the Internet, and focuses on real-world, client perceived performance. The final set of tests try to assess if the CS is an obstacle to scaling our system.

In all experiments, we set the maximum number of faults per replica group to one (i.e., each data object is replicated in $3f + 1 = 4$ nodes, and quorums of $2f + 1 = 3$ replicas were required for writes and public-key reads to succeed).

7.1 Controlled Experiments

Our first setup consisted of four identical machines connected to a fast local area network. These were Dell Precision 410 workstations with Linux 2.4.7-10. These workstations have a 600 MHz Pentium III processor, 512 MB of memory, and a Quantum Atlas 10K 18WLS disk. All machines were connected by a 100 Mbps switched Ethernet and had 3Com 3C905B Ethernet cards. The switch is an Extreme Networks Summit48 V4.1. The experiments ran on an isolated network.

The goal of this experiment is to determine the performance of the operations in Rosebud, compare it with similar systems, and identify the main sources of overhead. This was done in a local network to avoid the variability of Internet message delays.

We used simple micro-benchmarks to compare Rosebud against our initial code base of Chord running without replication, and with a simple BFT service that efficiently stores and fetches objects from an in-memory hash table. With only four nodes, Chord keeps complete routing tables.

In the first set of tests we consecutively stored and fetched 256 4kB objects in each of these systems. We repeated the experiment for both kinds of objects, except in the BFT service where we produced opaque data. The results are summarized in Table 1. The results show the average of three separate attempts. The standard deviations were always below 1.1% of the reported value.

System/Object Type	Store Time (ms)	Fetch time (ms)
DHash content-hash	2.3	3.2
Rosebud content-hash	23.9	3.3
DHash public key	23.5	2.9
Rosebud public-key (1P)	46.2	24.8
Rosebud public-key (2P)	68.4	49.0
BFT	3.2	2.4

Table 1: Local area network experiments. Average per object store and fetch time for different systems.

For store operations, Rosebud showed significantly worse performance than DHash or BFT. The reason is that Rosebud performs digital signatures in the critical path to ensure that the data has been stored at the correct nodes. DHash does not require this because the client trusts the nodes it talks to, and BFT uses MACs instead of signatures. Therefore, the additional overhead for each store operation is due mainly to signing responses: a digital signature takes on average 21 ms to execute on these machines. The remaining overhead is due to the fact that we replicate data in all four nodes, and therefore transmission costs go up by approximately that factor. Public key object operations are slower than content-hash because the client needs to sign the data before storing it, which accounts for the additional latency.

For two phase writes (i.e., when the client is required to read the version number before writing back the object) the signature overhead appears twice because both phases require signatures from the servers. A possible optimization is to optimistically execute the first phase without requiring signatures, and in the second phase replicas sign both the old version number and a receipt that they stored the current version number.

The second column in Table 1 shows times for fetching the same number of objects. Again, we see the cost of using public key cryptography to ensure freshness for public key fetches in the presence of Byzantine faults: all repliers must send a signed response containing the current version number and a nonce proposed by the client, since otherwise the client could be tricked into accepting stale data. This is not the case when Rosebud is fetching content-hash objects because these are immutable; thus replies do not need to be signed, and the only additional overhead is from fetching from more replicas.

For two phase reads, we artificially forced the client to write back the value it read (even though it read identical values from all replicas). These are relatively slow operations since they require two signatures in the critical path, but we expect them to be rare, since they occur only if the first phase found an incomplete write.

The comparison of Rosebud with other systems is highly conservative. DHash does not use cryptography due to a benign failure model. BFT uses MACs instead of public-keys, which only works because BFT clients had previously established shared keys for MAC authentication with the replicas. If a client interacts with a server group infrequently, BFT has to be changed to use public keys. For frequently accessed objects, Rosebud can easily be optimized to use MACs instead of public key cryptography.

Furthermore, even though the per-store overhead of Rosebud seems significant, it is acceptable when compared to the round-trip latency of Internet communication. For instance, a network round-trip across North America takes more than 70 ms on an uncongested link. This means our is system practical for a wide-area deployment, as we will show next.

7.2 Real World

These experiments use Rosebud servers running on a testbed of 22 machines scattered over the Internet. The machines are part of the RON testbed [3]; most have 733 MHz Celeron processors and 256 MB of memory (or similar), and all machines run FreeBSD 4.7. The experiments were run late at night, when network traffic was low (at least in North America, where most sites are located) and machines were expected to be unloaded.

Two experiments are described in this section. First, we look at a deployment of 88 Rosebud nodes. This is essentially a proof-of-concept experiment where had a real deployment with a significant number of nodes spanning three continents, and measured how fast clients can download data from the system. The second experiment focuses on algorithms to ensure atomic updates and compares our

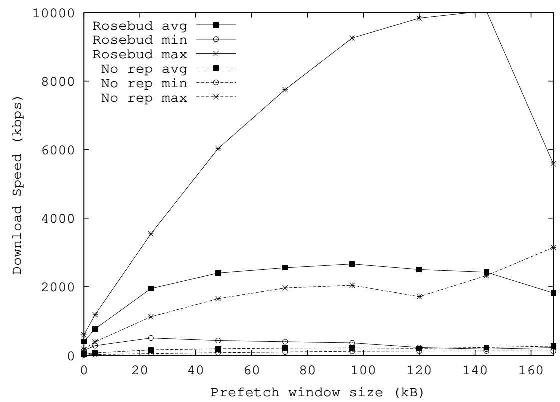

Figure 4: Average download speed for a 1 MB file divided into 4 kB objects for different pre-fetch window sizes. The nodes were part of the RON wide area testbed. We compare Rosebud with and without replication.

update algorithm BFT, which also provides atomic semantics, and is used by other large-scale storage systems [19, 1].

In the first experiment the servers held one megabyte of data that was split into 4kB content-hash objects. This produced a similar load across all servers. To test download speed, the client fetched all the data. The client used prefetch to overlap lookup with fetching: It initially issued a window of some number of parallel object fetches; as each fetch completed, the client started a new one.

We compare Rosebud with the same code without replication, i.e., setting $f = 0$. This will demonstrate the advantage of using quorums.

Figure 4 shows download speeds for the entire data, for a range of pre-fetch window sizes. The results represent download speeds from all nodes in the testbed, with three runs at each node. For each system, we show download speed for the fastest and slowest client, and for the average across all clients.

This experiment shows that Rosebud clients can perceive fast download speeds from the system, despite the fact that the data is replicated in a slow wide-area network. This is especially true if applications take advantage of prefetching, easily obtaining an average download speed above 2 Mbps. Note that the fastest client obtained download speeds over 10 Mbps.

The experiment also shows the advantage of using quorums. The fact that no-rep always downloads the block from the successor of the data makes it vulnerable to slow repliers. Rosebud uses quorums to avoid this: slow replicas can be excluded from read and write quorums.

Our second experiment in the RON testbed tries to assess the cost of providing atomic objects in Rosebud. We simulated a write-intensive application by performing a series of consecutive updates (with sequential version numbers)

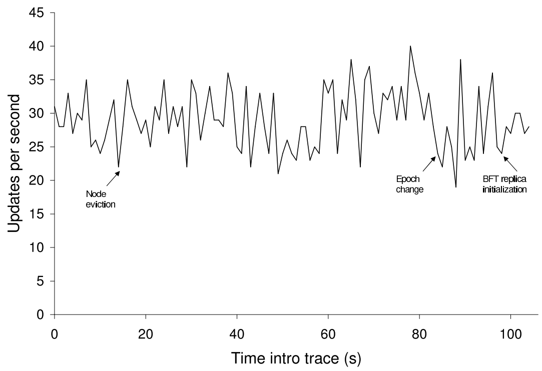

Figure 5: Reconfiguration timeline.

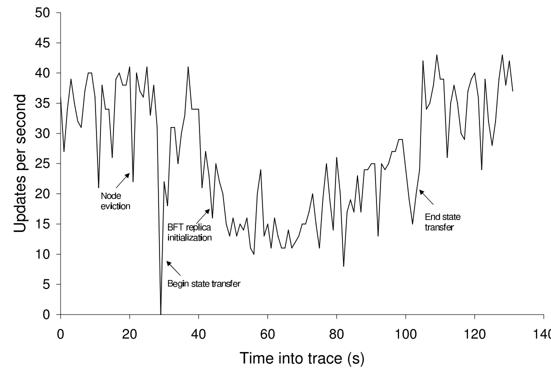

Figure 6: Reconfiguration and state transfer timeline.

to a public key object. Our benchmark performs a total of 256 updates to a 4 kB object. Since all the writes are to a single object, only one replica group will be contacted. Therefore this experiment only used four nodes located at Cornell, CMU, University of Utah and UCSD.

In this experiment, we compare Rosebud with BFT, which will provide the same consistency guarantees. We implemented a very simple and efficient BFT service that stores objects in an in-memory data structure.

The BFT service was running on the same replicas where Rosebud stored the data, and the client was located in Cornell for both systems. For BFT, we measured the performance for all four different choices of primary replicas. BFT was running with all possible optimizations.

System/Primary	Total Time [Estimated using PKs] (s)
Rosebud (1PW)	31.3
Rosebud (2PW)	53.3
BFT / Cornell	32.7 [49.1]
BFT / Utah	36.4 [52.8]
BFT / CMU	36.3 [52.7]
BFT / UCSD	42.6 [59.0]

Table 2: Update-intensive benchmark. Running time in seconds to perform 256 consecutive updates to a 4k object for Rosebud and BFT with different primary replicas.

Table 2 shows the running time for this benchmark. We report the average of the three runs. The standard deviations were always below 2.7% of the reported value. This verifies that our main design choices of minimizing network round-trips and not placing a single replica's work on the critical path were correct. We can see that Rosebud can outperform BFT when it writes in a single round-trip (this can be performed if the clients uses clock values as version numbers, or if there is just one writer, as in [30]), even though Rosebud uses digital signatures, which are more than two orders of magnitude slower than the MAC-based authentication of BFT. If BFT had to use digital signatures for client-server communication (and MACs for inter-server communication) each operation on BFT would take an additional 64 ms (digital signatures take an average of 32 ms on these machines, and both client requests and server replies would be signed), so the figures for BFT would go up by at least 16.4 seconds (we show the total estimated time in brackets).

Note also that the performance of BFT is dependent on a good choice of primary replica. Its performance degrades as the primary moves away from the client, since the first phase of the protocol is a direct communication between the client and the primary. Therefore our choice of only using nodes that were connected by Internet 2 links was beneficial to BFT, since a congested link between the client and the primary would seriously affect BFT's performance. But Rosebud would not be affected provided the other connections were fast. This property makes Rosebud more adequate for deployments in heterogeneous networks (e.g., an Internet environment).

We also expect Rosebud to scale better than BFT with the number of replicas, as BFT's protocols generate a number of messages that is quadratic in the number of replicas, and Rosebud generates a linear number of messages.

7.3 Cost of the Configuration Service

Rosebud's fully decentralized storage is scalable: since the behavior of the nodes in read and write operations does not depend on the number of nodes in the system, we expect these operations to continue to perform well when we scale to thousands of nodes.

However, the CS is the single point that is not fully decentralized. In this section we assess the cost of running a CS replica, and how that cost scales with the system size.

The CS may cause performance problems to the nodes that run it for three possible reasons: (1) the cost of monitoring other nodes in the system (ping protocol), which grows with the number of nodes in the system; (2) the cost of configuration changes; and (3) the cost of running node addition and eviction operations.

To test (1), we placed a node that was chosen as a CS replica under an intensive write load and measured how the write throughput changed as we increased the ping frequency. We ran this with 5% of signed ping requests. The result was that there was no change to the write throughput when we varied the ping frequency up to a maximum frequency of 100 pings per second. Running pings at this frequency allows us to cycle every node in a $10,000$ node system every 2 minutes. This suffices given our goals of delayed response to failure. Thus the ping protocol is adequate for large systems and does not slow down CS replicas.

To see the effect of (2) and (3), we measured the write throughput for a CS replica under a heavy write load, while it executes node eviction operations and epoch changes (Figure 5). The node in question goes through a series of CS operations: It participates in evicting a node from the system, it finishes the current epoch, signs the new configuration and starts propagating it. The node in question will also become a CS replica for the new epoch, so it has to spawn and initialize a BFT process after the epoch change.

This trace shows that throughput drops when running CS operations, but this decrease is small. Furthermore, these occurrences are rare, given that membership changes are rare, and epoch changes only happen every few hours. We conclude that the centralization imposed by the CS is not a significant burden to the nodes that implement it.

Even though this experiment ran in a small system (around 20 nodes), the cost of having more nodes is negligible, because the cost of most individual operations shown in the timeline does not increase with the system size. This is true for member evictions, epoch changes, and BFT initialization. Furthermore, we designed the operations that have increased costs to scale well: We only transmit deltas during epoch changes, and we expect these messages to be small, given our moderate membership dynamics assumption. Also, our multicast tree propagation scheme implies that the maximum number of per-node transmissions to propagate a configuration is $O(\sqrt[3]{N})$, which is small.

We also measured the effect of state transfer in a system under a heavy write load, by measuring the update rate at a node under constant write requests while that node changes epochs. We injected one fault and one node addition near the neighborhood of the node we analyze, which forced the node to transfer 40 MB from other nodes.

This trace is shown in Figure 6. We can see in this trace one large throughput drop immediately after the epoch change. This is cause by an inefficiency in our state transfer protocol (soon to be fixed) where we request neighboring nodes to send all object IDs in the new interval in a single message. After this, we see a decrease of throughput while the node is performing state transfer. Note that in this execution we allowed the node to transfer three objects at a time. Changing this parameter will allow us to trade impact on troughput for time to complete state transfer.

8 Related Work

Byzantine fault tolerance has been widely studied both in theoretical and practical settings. Different building blocks to build Byzantine-fault-tolerant systems have been studied, e.g., Byzantine agreement [22, 7, 9, 8], state machine replication [35, 32, 18, 11], or quorum systems for read/write variables [26, 27].

Our storage algorithms are most closely related to Byzantine quorum systems [26, 27] (in the static case they resemble Phalanx [27]). However, we differ from all of the above in two important aspects. First, previous work assumes that each replica maintains a copy of the entire service state. Therefore they do not need to address issues like partitioning the service state and dealing with a large scale membership that needs to be agreed upon. Second, these algorithms assume a static membership of the system, whereas we address the complicated issue of providing strong semantics despite changes in replica sets.

Some state machine replication protocols (notably Rampart [32] and SecureRing [18]) are built upon group communication protocols, which allow membership changes. Adding and removing processes in these systems is a heavyweight operation, where all nodes in the system execute a three-phase Byzantine agreement protocol [33]. Our configuration module can be seen as a group membership algorithm, but it is designed to work with thousands of nodes. Group communication systems do not address some of the issues that Rosebud solves such as storage algorithms and state transfer.

Alvisi et. al. [2] looked at dynamic Byzantine quorum systems. However, the dynamics of their system are limited, since they assume a fixed replica set, and only allow the failure threshold to change throughout the execution. We focus on the more difficult problem of allowing a dynamic replica set.

In fact, providing an atomic memory in the presence of a dynamic membership is still an active field of research for the simpler fail-stop or crash failure model (e.g., [24, 25]). Our work advances on this by focusing on the harder Byzantine failure model.

Recently, some reconfigurable large-scale systems, notably Oceanstore [19] and Farsite [1], were designed to provide strong semantics despite Byzantine failures. The main difference from our work is that these systems do not provide neither a complete design nor an implementation of dynamic membership for their Byzantine-fault-tolerant components (the primary tier and the directory groups, respectively). We detail more differences to these systems below.

Oceanstore [19] stores data with various consistency guarantees. Their solution for the problem we focus on (strong consistency for mutable data) is based on a primary tier of replicas that run BFT [11] and thus serialize updates. We differ in that all our replicas are responsible for serializing updates and we do not rely on state machine replication but quorum operations that, due to their symmetric design, work better in heterogeneous environments like the Internet. Oceanstore does not address the issue of discovering and changing the set of primary tier replicas, and how to modify the storage algorithm (in this case, BFT) to work with a dynamic replica set. We could combine Rosebud with Oceanstore, using our system for mutable data with strong consistency (i.e., as a primary tier) and relying on Oceanstore's techniques for disseminating static content to billions of nodes.

Farsite [1] is a Byzantine fault-tolerant file system designed that uses spare resources from desktop PCs to logically function as a centralized file system. This system uses BFT [11] serialize updates. We differ from Farsite in two main ways. Farsite is a file system and most of its data structures and techniques are specific and optimized to a file system service. Second, like Oceanstore, they fail to explain how to change BFT to support dynamic replica sets.

Castro et al. have proposed extensions to the Pastry peer-to-peer lookup protocol to make it robust against malicious attacks [10]. This work is mainly focused on the fault tolerance of the lookup protocol. The improved protocol provides probabilistic guarantees that it finds the correct nodes, and the better the probabilities the worse the lookup performs. We assume moderate membership dynamics, which allows us to employ more efficient and deterministically secure protocols in determining the location of the data.

9 Conclusions

This paper has described a novel Byzantine fault-tolerant storage architecture named Rosebud. Rosebud is the first Byzantine-fault-tolerant system that is highly scalable and provides a complete solution for maintaining strong consistency when replica sets change.

We deployed Rosebud in a wide area testbed and our experiments confirmed the efficiency of our algorithms: clients download data from our system at fast speeds despite the data being located at distant nodes. Our algorithms are fully symmetric, and thus there is no performance penalty if some replicas are behind slow links. We also show that performance degradation is minor when the system reconfigures.

We are investigating interesting applications that use Rosebud, but we are also leveraging existing applications that are built for systems with a similar, well-specified interface [12] that we are compliant with. We are also looking at the possibility of extending the operation set to provide richer semantics, and of implementing state machine replication with a partitioned state.

References

[1] A. Adya et al. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, Dec. 2002.

[2] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. Wright. Dynamic Byzantine Quorum Systems. In *Proc. DSN*, June 2000.

[3] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. SOSP 01*.

[4] Anonymous. Citation omitted for blind reviewing. we will make this TR available to any of the program chairs at the reviewer's request.

[5] Anonymous. Distributed storage quota enforcement. Tech Report.

[6] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. 9th HotOS*, 2003.

[7] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–240, 1985.

[8] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th PODC*, July 2000.

[9] R. Canneti and T. Rabin. Optimal Asynchronous Byzantine Agreement. TR #92-15, CS Dept., Hebrew University, 1992.

[10] M. Castro, P. Druschell, A. Ganesh, A. Rowstron, and D. Wallach. Security for structured peer-to-peer overlay networks. In *OSDI'02*.

[11] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd OSDI*, Feb. 1999.

[12] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. 2nd IPTPS*, 2003.

[13] J. Douceur. The sybil attack. In *Proc. 1st IPTPS*, 2002.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Sigact News*, June 2002.

[15] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proc. CCCS*, 1997.

[16] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *Proc. SIGCOMM IMW*, 2001.

[17] D. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the WWW. In *STC'97*.

[18] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. of the Hawaii International Conference on System Sciences*, Hawaii, Jan. 1998.

[19] J. Kubiatowicz et al. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th ASPLOS*, Nov. 2000.

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

[21] L. Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.

[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[23] N. Lynch. *Distributed Algorithms*. 1996.

[24] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. IPTPS'02*.

[25] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service. In *Proc. DISC*, Oct. 2002.

[26] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Journal of Distributed Computing*, 11(4):203–213, 1998.

[27] D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. of the 17th SRDS*, Oct. 1998.

[28] D. Mazières. A toolkit for user-level file systems. In *Proc. Usenix Technical Conference*, pages 261–274, June 2001.

[29] R. C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology*. 1987.

[30] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. 5th OSDI*, Dec. 2002.

[31] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proc. 4th USITS*, Mar. 2003.

[32] M. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, pages 99–110, 1995.

[33] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, Jan. 1996.

[34] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. SOSP 01*.

[35] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, Dec. 1990.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM 2001*.