RGSA: A Redundant Global Storage Architecture

Jim Leek, David Schultz, Ben Schwarz
{jrleek,dschultz,bschwarz}@cs.berkeley.edu

December 17, 2003

Abstract

We present RGSA, a novel distributed storage architecture designed for reliability and simplicity. Storage servers export an elementary block-based interface that facilitates efficient implementation of higher level semantics on the client side. We introduce an encrypted filesystem that effectively utilizes this system, providing strong consistency and privacy guarantees even in the face of a compromised server. Finally, we survey security issues in distributed storage systems and analyze the properties of our system in this context.

1 Introduction

Ever since the introduction of RAID ([12]), there have been tremendous advances in storage clustering technology, leading to improvements in scalability and reliability. These technologies have had the most profound effects on high-end systems. Meanwhile, personal computers use the same basic storage technology as they did in the 1980's: a *single* disk pack per machine. As areal densities and rotational speeds continue to increase, single-disk systems become increasingly unreliable, and yet the cost of adding redundant physical devices seems prohibitive. Even small companies find that storage administration overhead and backups to be expensive.

On the other hand, many companies that manage large datasets have become quite adept at storage administration. For instance, [4] describes the storage area network developed at Google to provide fast, reliable access to a massive web index. Since storage seem to be inherently easier to manage on a large scale, we propose that storage should be provided as a service. This service can replace small storage systems with more reliable and cost-effective networked storage. Moreover, because the storage architecture is a simple extension of existing storage area network technologies, the cost of deployment for potential service providers is low.

Unfortunately, any wide area distributed system carries liabilities in the areas of security and privacy. Sensitive data is now transmitted over the network and managed by a third party. Furthermore, since each server on the storage network may hold data for thousands of clients, the servers are more attractive targets for attack. These issues must be addressed if storage services are to be accepted by the mainstream. We have designed our storage network to be simple, resilient in the face of failures, and resistant to denial of service attacks.

We also introduce an encrypted filesystem designed to keep clients' data private if one or more servers is malicious (perhaps because it has been compromised). It also detects attempts to surreptitiously modify data and it provides reasonable consistency guarantees even in the face of potentially adversarial servers. In order to compare our filesystem to previous work qualitatively, we develop a scheme for characterizing systems under several orthogonal threat models.

In Section 2, we discuss our goals in designing this system and the rationale behind our design. In Section 3 and 4, we describe the structure of the storage network and filesystem, respectively. Then in Section **??**, we analyze the security and privacy properties provided by our system and compare it to several others. We detail our prototype implementation in Section 6 and do a comparative performance analysis. Lastly, we summarize related work in Section 7, contrasting it with our own, and conclude in Section 8 with some reflections on our experiences with this system and list contributions of this paper.

2 Design Goals

RGSA is comprised of two parts: a server network and a client-side filesystem. The server network can be decomposed further into chunk servers and masters, as described in 3. Each component has distinct goals, and we have tried to the extent possible to accomplish those goals without too much reliance on the other components. All parts of the system were designed with an adversarial model in mind.

2.1 Goals for the Master and Chunk Servers

The chunk servers and master servers are the most vulnerable because they accept connections over the network (as opposed to clients, which only initiate them). Moreover, they are the most attractive targets for attack because they provide service for many users. We strive to keep the protocol simple in order to reduce the chances that we encounter design errors lead to exploitable vulnerabilities.

Additionally, it is desirable for the chunk servers and the masters to be mutually untrusting. If this is accomplished successfully, the extent of damage due to a compromise is limited to a localized denial-of-service attack.

2.2 Goals for Clients

The main goal for clients is to protect the integrity and privacy of their data. Additionally, updates should satisfy the ACID property. With respect to security, we should be able to answer the following questions:

- How do we know that the machines storing our data belong to parties who we trust to reliably store it?
- If a chunk server changes our data, how do we detect and compensate for the attack?
- To what extent can we protect the privacy of our data if a server is malicious?

3 Storage Network

We have architected RGSA around the central ideas introduced by Ghemawhat *et al.* in their work describing the organization of the filesystem and network used in the Google search engine [4]. The Google architecture is responsible for managing and caching hundreds of terabytes of data, and is extremely tolerant in the face of hardware faults. We found this design amenable to large scale distributed storage architectures as well, with a number of modifications to reflect our differing assumptions.

Unlike the Google filesystem, our system has multiple clients, who purchase disk space and bandwidth from a central authority. *Master* servers are responsible for accounting for these allocations and roughly distributing load across the network. Each master is backed by a large number of chunk servers that store the actual data. Thus, the master also serves as a cache for block location. Clients are able to issue queries to the master to learn which chunk server holds their data. The clients then communicate directly with the chunk servers to issue typical read and write requests, rather than going through the master.

Requests to allocate and free blocks are forwarded to the appropriate chunk servers, selected by the master based on available storage and bandwidth capacity. Such requests are expected to be infrequent as compared to actual I/O requests because they are only made when the total amount allocated changes by a significant amount. Similarly, lookup requests are rare because clients can easily cache the locations of their data, and need only query the master when the data migrates across the network (for example, because one chunk server fails and is replaced by another).

The master is not a bottleneck in this architecture; most of its (infrequent) requests are on the order of the size of a single IP packet, while the chunk servers can process requests contain many megabytes of data. For an extremely large distributed storage network, multiple masters may be used. In this scheme, each master is dynamically delegated a portion of the storage arena by a meta-allocator for the purpose of doling out storage, and any master may cache translation information for lookups.

Perhaps a less obvious task of the master is to decide where to replicate the data. There are a number of policies which one could choose to employ for a replication level of k:

1. [*Random Distribution*] The master server chooses a random subset of k chunk servers from the set of all active chunk servers. It randomly selects one of these as the primary chunk servers to which the client should issue requests to. As a large number of requests are processed, the Law of Large Numbers dictates that random distribution should favor all chunk servers equally with regards to disk, CPU, and bandwidth load. We later show, in Section 5, that a random distribution also is good for preventing information leaks. In our prototype of RGSA we have chosen this distribution scheme, because it is easy to implement and has the aforementioned advantage.

2. [*Load-Based Distribution*] The master server selects k chunk servers that have the most resources or lowest load in the pool of all chunk servers. The motivation for using this approach is that the resources are almost always nonuniformly used. One may choose to allocate based on CPU usage, memory usage, network bandwidth or disk space, or perhaps a combination of such factors. The disadvantage is that a load-based distribution scheme requires extra bookkeeping by the master server, as well as extra net-

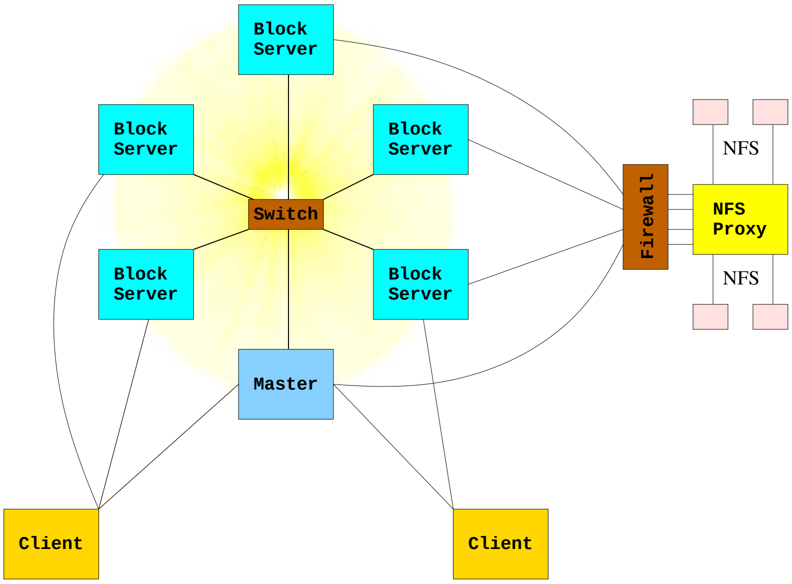

Figure 1: Network Storage Architecture

work traffic as it may need to occasionally query the chunk servers to determine their status.

3. [*Geographic Distribution*] It may make sense to spread out all replicas of data over a large geographic region to avoid natural disasters such as earthquake or large power outages. Conversely, it may be desirable to group replicas of data close together to reduce network latency times. A geographic distribution policy could be employed to handle such cases. As with a load-based distribution, this requires extra bookkeeping on part of the master.

The chunk servers hold large blocks of data[1] that are encrypted for privacy. They are responsible for replicating the data based on the level of replication desired by the client. A chunk server handles typical read and write requests, and they have a facility for scheduling writes using *I/O groups*. An I/O group servers two functions: it enables clients the specify dependencies that are necessary for the implementation of a file system on top of RGSA, but without forcing unnecessary ordering constraints on the server. Secondly, I/O groups allow a server to re-order writes within an I/O group to achieve a higher level of throughput to the disk. Each write request is associated with a particular I/O group, and I/O groups have dependency relationships which form a directed acyclic graph. Since dependencies are specified when I/O groups are created, and all dependencies must refer to I/O groups that have already been instantiated, it is impossible for a client to construct a dependency graph with cycles.

Both the master and chunk servers tolerate failures in the network. If a chunk server fails, replicated data is copied from other chunk servers to new locations to maintain the desired level of replication. The master server maintains a location cache of all the other chunk servers on which the data is replicated, so the new chunk server is quickly able to reconstruct all the data. In the case of the master failing, another machine can be promoted to the master position. The cache that the master maintains is easily reconstructible by querying the chunk servers to find out what data they hold. Accounting data is stored in an ordinary filesystem, so the master automatically gets all the consistency and durability guarantees that the filesystem provides. It should be noted that if a chunk server is compromised, the same protocol can be used for bringing up another chunk server.

4 Filesystem Design

4.1 File system Introduction

While the storage network provides redundancy, the filesystem layer provides privacy for user data and protection against various attacks that malicious servers might

[1] We found the block size of 64 megabytes to be acceptable in our prototype.

perform. Also, given that most data loss problems are actually due to user error ([11]), we provide snapshots and attempt to make the filesystem as self-managing as possible, including automatically expanding and shrinking allocated space as necessary.

4.2 Filesystem Design

Filesystem data is stored in the leaves of a Merkle tree. The data is held in the leaves of the tree and all nodes except for the root are copy-on-write: whenever a block is written or changed, a free block is found and the data is written there. The old file structure is maintained as the current one until the data is fully committed by writing the root, or super block, atomically. When the super block is written, the data atomically becomes committed and available. Some disks do not provide atomicity guarantees for writes larger than a certain size (e.g. 512 bytes). If we cannot make the root block as small as this size, then it suffices to keep two copies of the root block and update them sequentially. A hash of the block determines whether the block was completely written, and the hash should match for at least one of the redundant copies. A timestamp identifies the more recent block. On top of this system is a block cleaner, and a snapshot mechanism for rolling back user errors.

The Merkle tree offers strong consistency guarantees ([16]). Each indirect block contains both pointers to its children and collision-resistant hashes of those children. If any block is found that does not agree with the hash stored in its parent, we know that that block has been corrupted, either by disk error or by malicious activity. Furthermore, since only cipher text is stored, the hash cannot be spoofed by a malicious server.

Other filesystems typically use journalling protocols to guarantee consistency. These approaches cannot easily guarantee fork consistency, and they incur the overhead of writing the data twice: once to the log and once to the physical filesystem. Our system has many of the benefits of a log-structured filesystem, without the extra overhead of duplicating writes ([13]).

Additionally, we can efficiently support efficient snapshots in our system. Our file system allows the user to, on demand, create a snapshot of the current file system as long as space is available. Creating a snapshot is easy: a chunk of space is reserved at the top of the file system that is some multiple of the size of a single super block. In this space, each time a snapshot is taken, a new copy of the super block is written in old or unused space instead of the usual write-in-place. Each of these super blocks contains a version number such that on startup the system may request them all and use the one with the highest version number (or a user requested version number) as its active super block. Therefore, each snapshot only takes as long as a single write to create, but maintains all pertinent data of the filesystem at that time. The user may simply roll-back the entire filesystem, or a single file, if something goes wrong.

The one major disadvantage to the addition of a snapshot system is that a standard free block bit vector system is unable to represent the differences between blocks that have been freed, blocks required in the current system, and blocks that are part of an old snapshot. This problem may be solved by a "block-map" file instead of a "free-map" file. Each block is represented in this file as a set of bits that represents if the block is free, or used by a specific snapshot—including the current one. The algorithm for this system is shown in the [5].

Aside from the advantages and disadvantages of the snapshot system, laying out RGSA as a Merkle tree has one disadvantage: causing a group of updates to become durable is relatively expensive due to the number of indirect blocks that must be rewritten. We can amortize this cost, making it negligible, by grouping together many updates so that indirect blocks are written once for the entire group. Unfortunately, many applications require synchronous semantics. For instance, mail servers require that each message be durable before they acknowledge receipt. To prevent this from defeating performance, we add a local write-ahead intention log to the filesystem. This log is activated when applications make such requests (for example, via `fsync()`, and stores logical redo records that can later be used to replay the action in the case of a failure. Since the log is kept locally, we avoid network delays, providing a very efficient `fsync()`. The fact that all the data is saved means that the user will not lose data even in the case of an untimely crash. When the system claims it has committed the data, that data will be durable under almost any circumstances.

Finally, our file system includes a simple block cleaner—a feature left out of WAFL ([5]). This feature, borrowed from LFS, reduces the effects of fragmentation. While sequential writes are expected to reduce server seeks, the assumption being that reads are cached, fragmentation can carry significant penalties over time. Segment cleaning on a per-chunk basis can reduce this overhead.

4.3 Dealing with Concurrency

Although we discuss privacy in detail, people often wish to share data with particular people. The desire to allow concurrent access and modification of data has historically caused many problems, and RGSA is no exception. To reduce the load and complexity of the server, it is desirable for clients to be able to control concurrency without the explicit intervention of the server. This design also means that a malicious server cannot use the consistency protocol in order to confuse clients regarding the state of the filesystem. Furthermore, because the storage architecture knows nothing about the data it is storing—even the file system meta-data—it would be awkward to deal with consistency on the server side. All locking must be handled by clients in a peer-to-peer fashion.

Thus, clients determine which other clients are active by reading a special block in the filesystem, then communicate with each other to perform updates. xFS is a ([2]) fully decentralized distributed filesystem and uses a similar approach to guarantee consistency. Essentially, every cached block has an associated owner, who coordinates access to the block. The owner tracks the block state, which includes a callback for every client that has a cached copy of the block. The state of the block in each client's cache is either *modified, exclusive, shared, or invalid*. The modified state indicates that the most recent copy of the block is exclusively in a client's cache, and that it is invalid (not present) in any other cache. The shared state allows multiple clients to share the block read-only, and may be upgraded to exclusive or modified as in MESI protocols in shared-memory multiprocessors. The exclusive state is essentially an optimization that permits clients to upgrade to the modified state and write to the block without generating unnecessary bus traffic.

In theory, requesting ownership would require a broadcast among all clients using a particular filesystem. We avoid this by hierarchically partitioning the set of blocks in the system, so that certain clients control the delegation of mastership of an entire set of blocks.

Since the server is not involved in the consistency protocol, it does not need to be involved if we wish to analyze consistency in the context of a malicious chunk server. Clearly a malicious chunk server cannot modify one of our blocks arbitrarily because each block has an associated collision-resistant hash. However, we must consider the possibility that the server ignores some updates but not others or replaces a block with an older version. If the block is not the root block, then its parent will contain its hash and we can verify that the provided version of the block is correct. (The root block contains a hash of itself with the hash field zeroed.) Inductively, we can infer that the root block in some sense defines the state of the entire filesystem, with the exception of recent updates that the clients securely negotiate amongst themselves.

Thus, we can guarantee *fork consistency*, as defined in [7][8]. Informally, fork consistency implies that each client sees a sequentially consistent version of the filesystem *or* the set of clients is partitioned into subsets, each of which has an independent copy of the filesystem. The independent "forks" cannot be recombined without a client noticing the problem. The consistency protocol only strengthens this guarantee, since the clients have an out-of-band method of detecting such forks.

4.4 Dividing the File system: Hierarchical and Flat

Finally, we are concerned with the interface the file system outputs to the user. In our file system, each file as a unique 64-bit ID number, henceforth referred to as the file ID. This file ID is a unique identifier for the file, and is used to reference the file internally. The super block holds a pointer to the "file-map file" that maps each file ID to a pointer to its inode. The file system itself knows nothing about directories or filenames.

However, overlaid on top of this basic file system is a directory structure. Each directory is simply a file that maps file names to file IDs. Decoupling the file system from filenames offers great flexibility in the interface outputted to the user.

It is important to emphasize that the file hierarchy seen by the user is orthogonal to the tree of blocks used to enforce consistency. The former is the abstraction upon which we construct a flat namespace with strong consistency guarantees, and the latter is built on top of it. We implemented such a hierarchical namespace in under 800 lines of Java code.

5 Security and Privacy Issues

We have architected RGSA to minimize information leaks, and be fault tolerant in the face of compromises. This section describes the issues in privacy and security that we have looked at.

5.1 Threat Model

We examine a threat model that includes attacks from both malicious users and also from chunk servers that have been compromised by these adversaries. It is also prudent to analyze the situation where malicious users are working in concert with compromised chunk servers. It is probably impossible in general to completely protect users' privacy, because at some granularity, a compromised chunk server could infer something about the use of the filesystem based on access patterns. For instance, a company might observe when one of its competitors' filesystems has the highest activity in order to predict timing of product launches.

It is reasonable to assume an attacker can listen to the communication going between non-adversarial clients and chunk servers or master servers, as well as the communication between two chunk servers, or a chunk server and the master server. Many applications are available for download on the Internet to aid in such an attack. The Google architecture does not have to consider such a threat, as all the communication between chunk servers is across a private and trusted network. It is imperative that the chunk servers share no trusted computing base, so that if one is put under the control of an adversary it can do no more harm than what a malicious client could do. However, we will discuss why a compromised chunk server may leak information to the adversary that cannot be learned otherwise. Throughout this section *attacker* will be used interchangeably with *adversary*.

Figure 2 illustrates the threat model.

5.2 Attacks Against Privacy

Before we list the ways an attacker may try to learn information, we first present a hierarchical view of privacy for RGSA. It seems unlikely that the problem of privacy will ever be completely solved, so it makes more sense to talk about partial solutions. The four levels of privacy P_0-P_3 in our model are defined as:

1. [P_0*: Complete Visibility*] An adversary can see data or metadata that belongs to another user. For instance, if all data and messages sent between clients and chunk servers are in plain text, the architecture is not P_0-*resilient*.

2. [P_1*: Metadata Visibility*] An adversary can directly observe filesystem metadata but not the actual data. For instance, the attacker may be able to determine the structure of files or objects on disk.

3. [P_2*: Metadata or Data Visibility via Covert Channels*] If an attacker cannot directly observe filesystem metadata, they still may succeed in constructing parts of it by observing covert channels. An example of a P_2-effective attack is a timing attack where an adversary analyzes the time being TCP/IP packets and is able to detect a pattern and infer information about the requests. Another similar attack would be if a malicious chunk server analyzes the size of data items being written and read to deduce what type of data is being stored. For instance, a particular application may have a very distinguishable file structure, that when installed is easy to detect. An architecture must have countermeasures against these type of attacks to be P_2-*resilient*.

4. [P_3*: No Visibility*] An adversary can learn nothing about metadata on the filesystem, and nothing about the data of any other users accessing chunk servers. Ideally, an architecture designed to plug information leaks should be P_3-*resilient*, however we feel this goal is not reachable in practice.

We will call an architecture P_n resilient if it is successful at stopping attacks against privacy at level P_n and all those levels lower than it. An architecture that compromises no privacy is P_4-*resilient*. One which leaks all information is P_0-*resilient*.

5.3 Evaluating Effectiveness

We feel RGSA offers a substantial improvements over other distributed storage architectures in the way of privacy. We will start with level P_0 privacy and work our way up, explaining why.

Prior to storing data on the chunk servers, the clients encrypt the data using a plug-in symmetric key cryptosystem. The choice is entirely up to the client—the point being that the chunk server need not know about the data being stored on its local disk. If a chunk server is malicious or is compromised, it can only see cipher blocks and hence cannot deduce anything about the contents without knowing the key. We also assume the connection between the client and the chunk server is encrypted using a public key cryptosystem. The structure of a message to the chunk server is thus in the form $P_k(\text{Op}, S_K(\text{Payload}))$. An adversary that is able to snoop packets going between the client and chunk server cannot determine what type of operation is being performed. The chunk server must know the operation in order to issue the request to its local disk, so it decrypts the message but still cannot read the payload data. Given a secure cryptosystem, it follows that RGSA is P_0-*resilient* since the wire format and disk format cannot be seen. We believe OceanStore to be P_0-*resilient* since an encryption scheme can also be used to store files or objects. The Google architecture is not P_0-*resilient*—all data stored on their servers is available to the general public to view, so it need not be encrypted.

P_1-*resilience* is also straightforward to show. The clients that communicate with the chunk servers issue requests to store blocks of data. Inside these blocks may be a number of small files, directories, or other meta-data.

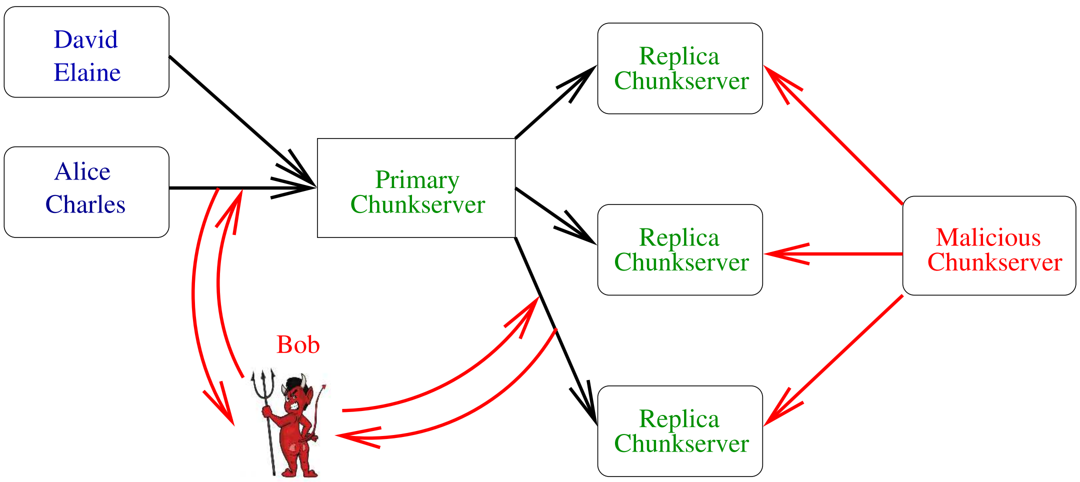

Figure 2: Threat Model for Privacy and Security in RGSA. We consider malicious clients and compromised chunk servers—possibly in cahoots.

It may even be the case that a single block holds only a portion of a file. For this reason, a compromised chunk server knows nothing about the metadata of the filesystem implemented on top of the storage network. More specifically it cannot determine the location of directories, inodes, filemaps, superblocks, etc. It is not obvious that this same extension can also be applied in an architecture like OceanStore, which explicitly stores objects. The natural way to implement a filesystem would involve having objects for all filesystem components. It may be possible, however, to treat a single large object as a virtual disk as we have shown to work in RGSA.

Although a malicious chunk server cannot directly see the data being stored on the local disk it controls, it still may be able to infer something about its contents using timing-based attacks. Because the chunk server knows the primitive operations being performed, it may be able to correlate the operations to well-known behavior for other data sets. For example, a particular application `A` may have a very noticeable installation procedure—for instance, it has cyclic portions of read → compile → write to build a copy. The chunk server is able to see the delays between the read and write requests, and may be able to determine the number of files or objects that the application contains [2]. RGSA makes it more difficult on the attacker by: 1) buffering outgoing requests; 2) inducing random server delays; 3) servicing requests out-of-order; and 4) storing blocks as opposed to files or objects. In particular, a single block may contain files from multiple users of the filesystem, and thus data is pooled and difficult to separate out based on the actual owner. We do not claim that RGSA complete solves the problem of timing attacks, but we strongly believe it mitigates some of the attacks that are easiest to mount. Consequently, we will not claim RGSA is completely P_2-*resilient*, although it is a step in the right direction.

One privacy issue not discussed above or included in the hierarchy is that of data location deduction. An attacker may want to learn where certain data for a specific user is being replicated, so that in order to completely destroy the data only a fraction of the chunk servers would need to be compromised (or destroyed). In order to mount such an attack, the adversary must be able to correlate the outgoing TCP packets from a chunk server to the incoming packets from a specific client. This would enable them to determine where the replicas are being forwarded to. We attempt to thwart such attacks by once again using buffers, out of order request servicing, and random delays into the chunk server. The replication policy, in which chunk servers are chosen at random to contain the replicated data, also ensures that at any given time splice, the outgoing TCP packets from a particular chunk server are likely to be random over the distribution of the other chunk servers. Since we have multiple clients accessing data, the number of outgoing socket connections is likely to be large, and it will be difficult to take a subset of them and claim they originated from a particular client.

[2]This turns out to be surprisingly effective in determining the identity of some applications.

5.4 Attacks Against Security

5.4.1 Malicious Client or Man-in-the-middle

Related to the attacks against privacy are those against security, in which an adversary does not care about reading the data of another person but rather is concerned with destroying it. We claim that man-in-the-middle attacks by any adversary are not possible for three reasons. First, each message sent to the chunk server is encrypted using a public key. If the attacker tries to emulate a request from a client he must know what this key is, or have previously seen the same same request. In the case that the request is being *replayed*—e.g., the adversary remembers the byte stream from a previous communication session that was monitored—the network is still safe. The communication stream between the client and chunk server can use a challenge-response protocol similar to what SSL uses to make ensure the client is really the send of the data. Finally, each message contains a message authentication code (MAC) which effectively prevents any man-in-the-middle from injecting or removing garbage data from the message,

5.4.2 Compromised Chunk server

The storage network consists of nodes of chunk servers that communicate with each other for replication, and with the master for central policy requests. In the case that a chunk server is malicious we must ensure it cannot destroy data on other chunk servers or influence the master to do something bad. We have designed the replication protocol such there is no trusted computing base between chunk servers; that is, the chunk server responsible for replicating to other chunk servers acts as though it was merely a client when sending the requests. For this reason, a compromised chunk server is no more dangerous than a malicious client, although it may leak be able to learn more information than one. In particular, it knows where the replicated data is being stored, and knows about the type of requests it is processing. These are things a malicious client cannot find out.

Another attack by a malicious chunk server is that it may ignore certain updates from clients (who think the chunk server is still a legitimate host) in an attempt to make the data in the filesystem appear out-of-date. We solve this problem by using a Merkle tree with SHA-1 hashes at each node. The tree provides *fork consistency*. If the server tries to trick the clients from seeing the updates of each other, the problem can either be detected or else the clients live in parallel universes. The SHA-1 hashes at each pointer in the tree guarantee integrity its integrity. In fact, to check the integrity of the entire tree requires only checking the SHA-1 hash of the root node[3]. Freshness guarantees are made by timestamping each node in tree.

5.4.3 Compromised Master

Finally, we consider the case of a compromised master server. In particular, a compromised master must not be able to arbitrarily issue requests to the chunk servers to free all the data they contain. We propose that all allocation [4] and deallocation requests have a signed certificate from the client issuing the request. In the case of a malicious client which refuses to sign a certificate (perhaps they refused to pay the monthly fees), we propose an offline protocol be used to manually manage blocks. The certificate ensures that a master who has been taken over cannot cause harm to data by deleting it.

6 Implementation and Performance

We have implemented a functional prototype of RGSA using the Java programming language. As proof of concept, we have also implemented a hierarchical filesystem and simple shell to manipulate files and directories. The implementation, which was done in around 5400 lines of code, can be decomposed into three central parts:

- *The storage manager*, which is responsible for managing the storage of the local disk. It also resolves the I/O group dependences by producing a valid topological sort. There is a storage manager for each local disk, although in our implementation each chunk server has only one disk.

- *The network and translation layer*, which processes requests coming from the client and forwards them to the storage manager. This layer handles the coordination of the Master and chunk servers. We adopt a multithreaded approach, where each incoming connection results in a new thread being spawned to handle the requests. In retrospect, an event-based approach might have been easier.

- *The filesystem*. The filesystem is a client object in the RGSA model. It requests chunks and treats them as virtual disks. The filesystem is hierarchical and structured similarly to LFS. A simple shell sits on top

[3]The hash is based on the hash of its children, so the argument works inductively.

[4]For the sake of preventing a malicious master from allocating entire disks to a client and erroneously charging a client for all the disk usage which was not actually requested.

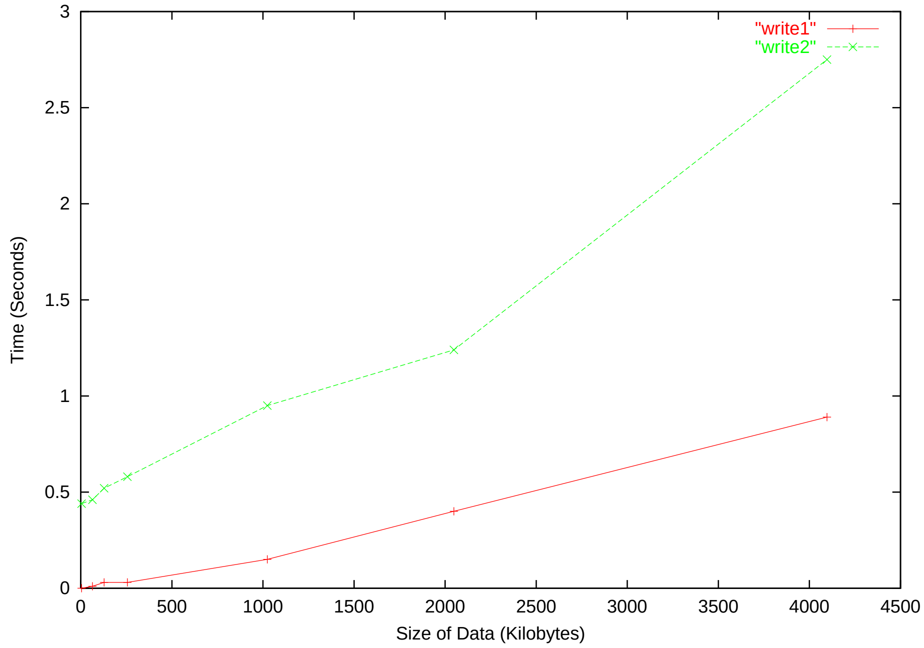

Figure 3: Performance of Writes in RGSA

of the filesystem and allows for interactive user input. Supported commands include: *read*, *write*, *create*, *rm*, *cd*, *mkdir*, *pwd*, *tree* (a command to display the entire filesystem tree). Since the consistency protocol has already been demonstrated in xFS, we did not make its reimplementation a priority.

Our prototype allows for chunk servers to be hot-swapped into the system and instantly utilized by the Master, although we have not implemented any of the fault tolerance design discussed in Section 3. There is no noticeable latency in the interactive filesystem once everything has been initialized, and the servers easily support a large number of clients. The most recent source code can be downloaded from

`http://www.cs.berkeley.edu/~dschultz/fa2003/proj`

along with simple usage instructions.

6.1 Results

We have run performance tests with our filesystem and storage network to evaluate the speed of write and read requests in comparison to that of the local filesystem [5].

[5] We are unaware of performance tests done for other architectures organized in a similar structure to RGSA

The clients were running on a 1600+ Athlon XP running FreeBSD 4.8. The master and chunk servers were running on a 2200+ Athlon XP running Redhat Linux 7.2. Figure 3 shows the performance of writes on our storage network compared to the local filesystem. Figure 4 shows the performance of reads. Write results were gathered on the local filesystem by using the `echo` command to redirect input to a file. On RGSA, a similar mechanism using the interactive shell was employed. In the graphs, "write1" and "read1" come from the local filesystem, and "write2" and "read2" are from RGSA.

The results indicate that the performance impact for writes is noticeable, but not significantly larger than that of a local filesystem. The network link between the machines has a bandwidth of 100 Mbps and an approximate round-trip time of 0.7 ms. We expect that over longer links, results for writes would be poorer in proportion to the connection bandwidth. Thus, the local intention log is crucial to ensure that applications requiring synchronous writes do not need to pay the penalty of a round-trip-time for each such operation.

The performance hit for reads was substantial, but we believe that this problem is mainly a property of our implementation rather than our design. In particular, our caching implementation was very poor, flushing cached data for every file when the file is closed. This re-

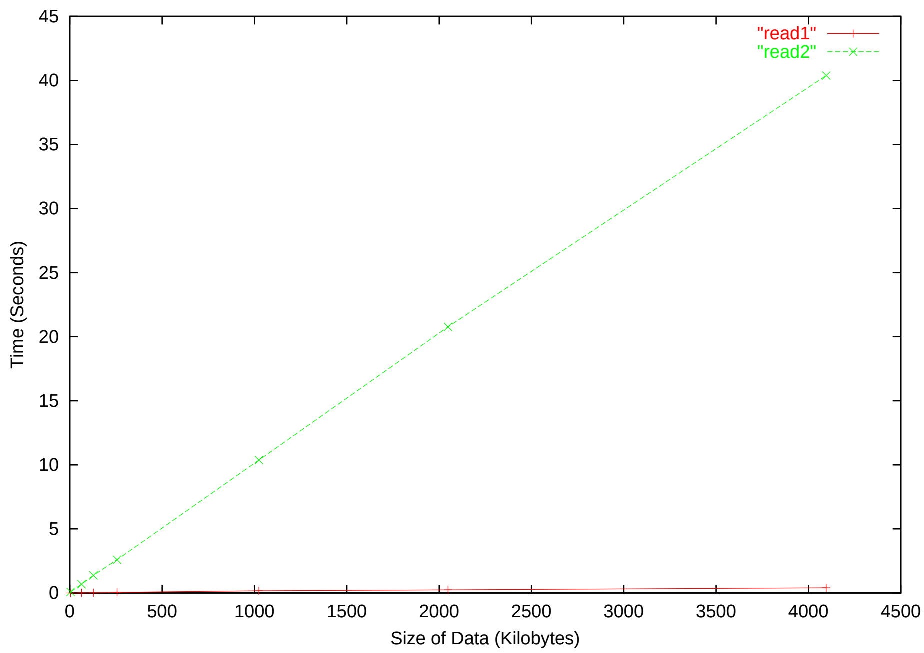

Figure 4: Performance of Reads in RGSA

quires that we pay at least one round-trip time every time we open a file, resulting in a pathname lookup cost of $n \times RTT$ for a path with n components. We conclude that the effectiveness of the caching protocol is the most important element in client-side performance. This is especially true given that network *bandwidth* is increasing much more quickly than *latency*, which is bounded by the speed of light.

We also looked at scalability of the chunk servers with respect the number of clients. For the test, we configured clients to continuously perform the following operations:

- Copy the directory tree into the filesystem for the source to the GNU C compiler version 3.2.2, which contains 1202 files and 32 directories totalling 44 MB.
- Copy this directory tree to a new location in the filesystem
- Delete both copies

A separate process on the local machine measured the response time for a single read request. Three samples were taken for each datapoint to control for variance. The results are summarized in Figure 5. Clearly there is a significant latency in processing each request, but enough parallel requests can occur simultaneously that response time increases nominally with the number of clients, even in our prototype. After about 260 clients, the bandwidth of the machine's network interfaces became the limiting factor and we were unable to test with a larger number of clients without skewing results by running some of the clients locally.

We lacked the hardware resources to obtain any interesting results from the master server. We started three block servers on three separate machines on a 100 Mbps network and had clients continually log into the network with new user ids, creating new filesystems (and thereby allocating new chunks) constantly. However, the master's response time was flat until the machines we were running the clients on began to thrash.

Based on our data for writes, we believe that our design can lead to entirely reasonable performance in a production implementation. Most of the overhead for writes is attributable to the cost of copying data between the kernel and userland, as well as the garbage collection overhead of using Java. We found that Java was a poor choice of language for our distributed filesystem because it provides inadequate control over memory, and because it supports event-based programming poorly, requiring a separate thread for each I/O request we wish to issue to the underlying operating system in parallel and a thread for each network connection.

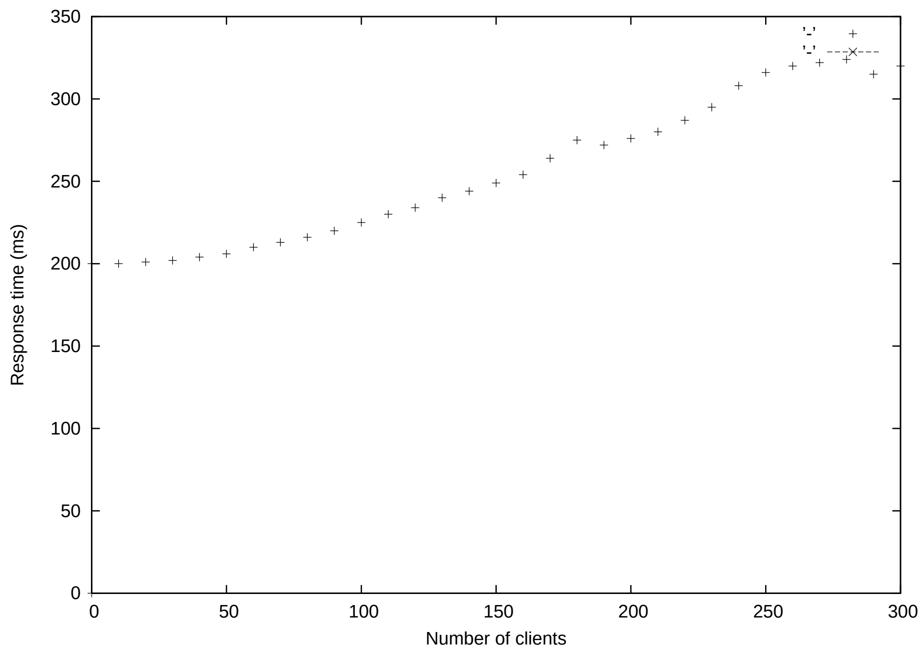

Figure 5: Server response time vs. number of clients

7 Related Work

This paper evaluates a new distributed filesystem that provides strong consistency, reliability, security and privacy guarantees. It incorporates ideas from encrypted filesystems, consistency protocols, storage area networks, and log-structured filesystems. Here we evaluate several other ways in which some of these ideas have been combined.

Ghemawat et al. ([4]) describe the storage architecture used at Google to perform millions of search requests per day efficiently. Their network uses masters for object location, but it is optimized for extremely large datasets streamed over the network and offers no consistency for updates.

The designers of Sprite ([10]) introduced a distributed filesystem in which writes were cached locally to improve performance, although they had an unrefined coherence protocol. xFS ([1][2]) has a more sophisticated consistency and coherence protocol based on directory-based multiprocessor protocols. Our system relies heavily on being able to maintain consistency via client-client communication, with little intervention from servers, and the viability of this scheme is well demonstrated by xFS.

Sprite also introduced the log-structured filesystem ([13]), which showed that being able to perform writes sequentially substantially improves performance. Our filesystem, based on Merkle trees ([9]), takes note of the effectiveness of the LFS allocation policies while providing a cleaner structure for accessing filesystem metadata. The original copy-on-write filesystem design actually dates back to the first System R DBMS prototype ([3]), although it proved to be unsuccessful in that case due primarily to the complicated handling that was required for partial aborts. [5] also used this approach to greater success and showed how an intent log can improve the performance of synchronous writes and amortize the cost of updating the tree of blocks over many writes. However, they were implementing a standards-conformant NFS server, so their intent log was stored on the server side.

GPFS ([14]) is a highly fault tolerant shared-storage filesystem. The designers attempted to handle both node failures, network failures, and disk failures, and recover from each. This is possibly the most difficult part of a distributed filesystem, and an aspect that is often ignored.

OceanStore ([6]) is a highly decentralized, massively distributed storage network that has similar consistency guarantees to our system. Unlike our system, block location is determined via distributed hash tables ([17]). We decided *not* to use this approach because of the security considerations voiced in [15], which include routing attacks, denial of service attacks, and other ways malicious users may introduce inconsistent behavior. We could presumably use DHTs if our masters prove to limit scalability, but preliminary results show that with a sufficient number of masters, this will not be the case.

8 Conclusion

We have introduced several design concepts for global storage architectures. Chief among these are the architectural issues for security. RGSA is highly resilient in the face of compromise: all components are fault tolerant and mutually untrusting. RGSA attempts to protect the integrity and privacy of data at every stage, and the architecture plugs common sources of information leaks with very simple solutions. Additionally, we have introduced a simple and stateless protocol by which clients communicate with chunk servers and Masters. RGSA satisfies the ACID property: it has provisions for atomicity, guarantees consistency, provides isolation, and can be highly durable if desired.

References

[1] M. Dahlin, C. Mather, R. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Measurement and Modeling of Computer Systems*, pages 150–160, 1994.

[2] M. D. Dahlin. Serverless network file systems. Technical Report CSD-96-900, 1996.

[3] D. C. et al. A history and evaluation of system r. pages 632–646, 1981.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, New York, NY, USA, October 2003.

[5] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, November 2000. ACM.

[7] D. Mazières and D. Shasha. Don't trust your file server. In *Eighth Workshop on Hot Topics in Operating System*, pages 113–118, 2001.

[8] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.

[9] R. C. Merkle. A certified digital signature. In *Proceedings of the 1989 Annual Cryptology Conference CRYPTO*, pages 234–246, 1989.

[10] J. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[11] D. A. Patterson. Recovery oriented computing: A new research agenda for a new century. In *HPCA*, page 247, 2002.

[12] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, June 1988.

[13] M. Rosenblum and J. K. Ousterhout. The LFS storage manager. In *Proceedings of the USENIX Summer 1990 Technical Conference*, pages 315–324, Anaheim, CA, USA, 11–15 1990.

[14] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[15] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables.

[16] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying file system protection. pages 79–90, 2001.

[17] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.