SILT : S3 Integrity, Locking, and Transactions

Hussam Abu-Libdeh
CS-6464 Class Project
Cornell University

Abstract

Despite their limited put/get interface, simple hosted storage services are becoming very popular. Many companies and individuals are using them to store and backup data. Recent work has demonstrated that it is even possible to build file systems on top of such abstractions. However, the available API lacks some features that could prove beneficiary in building distributed storage systems on top of such services. In this project I investigated extending the interface of Amazon's S3 storage service to allow for more powerful capabilities such as locking, multi-key transactions, and ensuring data integrity.

1 Introduction

Hosted storage services, such as Amazon's S3, provide consumers with a storage service that is highly available, scalable, reliable, fast, and relatively inexpensive. Such services often provide a simple put/get interface that allows users to place and retrieve data ranging from a few bytes to gigabytes in size.

Amazon's S3 service exposes a hashtable-like interface to its users: data is stored in key-value pairs, and pairs are grouped together into named buckets. S3 provides reliability and availability by replicating data to many servers that could be geographically dispersed. Although updates on a single key are atomic, only eventual consistency is guaranteed. Fetching a key's value might return stale data. Additionally, S3 does not support neither locking nor multi-key atomic updates or transactions. These shortcomings could present obstacles for multi-user settings where key-value pairs on S3 might be constantly edited. Furthermore, S3 provides little tangible guarantees that data retrieved on a particular read faithfully corresponds to the last put operation on that key (data integrity).

To exemplify these problems, imagine a scenario where a company uses S3 to store internal files that are shared between multiple employees. Employees may retrieve and edit any number of files, some of which might also be editable by others. In order to maintain a serializable view of the file system, users must ensure that they are dealing with the latest version of a file when modifying it. Serializability also requires the use of locks to prevent simultaneous conflicting updates to a single file. Furthermore, as for many modern file systems, transactions are necessary to group multiple updates and be able to rollback undesired outcomes. Multi-key locks and transactions are also necessary since a single file operation might span multiple key-value pairs for data and meta-data. Finally, file system users need the integrity of their data to be preserved.

In this project I built a system that extends S3's interface and provides these missing capabilities to its clients. Key locking is implemented by the use of a locking service [4,5]. The clients and lock server track the latest key-hash pairs in order to preserve data integrity and avoid staleness. Multi-key transactions came as an immediate consequence of implementing locking and has been integrated into the provided API as well.

2 Related Work

Many network file systems such as NFS [1] have been designed, analyzed, and discussed in literature. These systems rely on a centralized server to manage the updates. The situation is similar to some extent in GFS [2] as well, but both are different than the case here. In this instance S3 manages the data however it has no knowledge of the structure of the data and so can not provide consistent views of the network file system if it were to be built on top of it.

Systems like Bayou [3] manage updates in weakly consistent settings. However, different from our scenario, Bayou clients communicate with servers that themselves can be disconnected. Amazon does indeed manage S3 behind the scenes and ensures the availability of its data. Additionally, the disconnected nature of Bayou meant that a locking service can not be used, and thus update conflicts can arise and in fact users had to be aware of these conflicts when using Bayou. In comparison, SILT uses a locking service so that conflicting updates are avoided when possible by using locks. The end user has to manage conflicts

only if the locking service crashed during operation as will be shown. A locking service ensures that at most one process has access to a particular key at any time and it can be implemented via a centralized lock server that easily manages locks locally, or via a distributed locking service like Google's Chubby [4] or Yahoo's Zookeeper [5]. In this project I chose to implement a centralized locking service that shares its state with the clients of the system so it can reconstruct its state after a crash.

Data integrity and serializability have been explored in systems such as SUNDR [6]. There, a server was assumed to be untrusted and capable of equivocating to clients, and the goal was to provide fork consistency for clients. This was achieved by having the server log the read/write operations it encounters along with the signatures of the users that issued these operations. Clients keep track of their latest interaction with the server and the log as they saw it. Clients reconstruct the status of the file system by redoing the operations from the log. Because clients keep track of the status as of their last interaction with the server, this allows a client to detect if a server has omitted something from its log that the client has already seen in a previous interaction. However, this does not allow clients to detect whether the server is presenting a different, yet consistent, branch of the log to them, and thus only fork consistency is achieved. However, since we can not modify S3's public API, using a technique like SUNDR does not apply here.

3 Design Overview

There are three main aspects of the system, integrity, locking, and transactions. I will discuss them as well as failure recovery in the following sub-sections.

3.1 Data Integrity

In distributed storage systems with multiple readers and writers, it is important for clients to verify that a particular value read from the system corresponds to the latest written version. The ability to read and operate on the latest written values is important to achieve a consistent view of the data across the system. If clients could read and operate on stale versions of the data, then the system can reach an inconsistent state.

Verifying that read values correspond to latest writes is also important if the storage infrastructure is not controlled by the users (owners of the data) but is rather out-sourced to a third-party like Amazon S3. In that case, we can imagine a malicious data store that answers client requests with incorrect data.

Fortunately, verifying data integrity and freshness can both be done with the use of hashes. By maintaining the hash of the latest written value to a particular key, clients can compare a retrieved value against that hash to check if their read corresponds to the most recent version of the key value , and that its integrity has not been compromised. SUNDR uses a similar technique to verify not only the latest version of a file block, but also the entire history of that file. Finally, since the data store can be malicious, value hashes must not be stored in the data store along with the data, but are rather externally on the clients.

In SILT , whenever a client writes a value to a particular key on S3, it computes a hash of the new value. A timestamp for that hash is obtained from a centralized server. The client then gossips this key-hash-timestamp triplet to other clients in anti-entropy[7] style. Clients discard key-hashes if they receive a gossip message with a newer hash for that key. When a client reads a key value from S3, it compares the obtained value with the latest known hash of that value. If the value doesn't match the expected hash, this could either be due to S3's eventual consistency characteristics which could cause stale versions of the data to be returned on a request, or it could be due to a malicious data store. The client tries to retrieve the data a bounded number of times before reporting an error back to the user.

Relying on pure gossip to spread key hashes could still result in a situation where the client has a stale version of the key hash, and it retrieves a stale version from the data store that matches this hash. However, this concern is addressed in the next section.

3.2 Locking

To coordinate simultaneous or overlapping reads and writes, a locking service is used. In SILT , a centralized lock server grants locks on keys for the different clients. This server is also responsible for issuing hash timestamps as mentioned in the previous section. The system issues read or read/write locks. Locks can be issued on a single key or a collection of keys.

When a client wants to interact with S3, it first acquires a lock from the lock server by sending a lock *request* to the server. This request specifies the keys that the client wants to acquire the lock on, and the latest hash and timestamp that the client knows for each of these keys. If the requested keys do not overlap with existing write-locked keys, then the server replies back to the client with a lock *grant* message specifying the latest hashes it knows about for each of these keys. At this point the client now owns the lock on the keys it requested.

Once the client is done with a key lock, it releases it by sending a *release* request message to the server specifying the key it is releasing along with that key's hash value. The server acknowledges the release of the key via a *grant* message as before.

If the client had modified the key's associated value, then

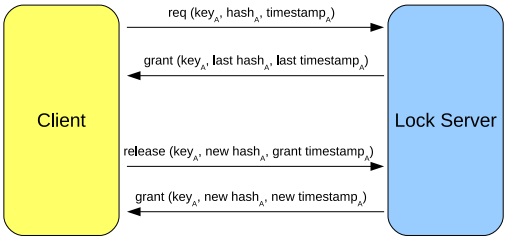

Figure 1. Interaction between a client and the lock server. Hash and timestamp information are piggybacked on lock-related messages.

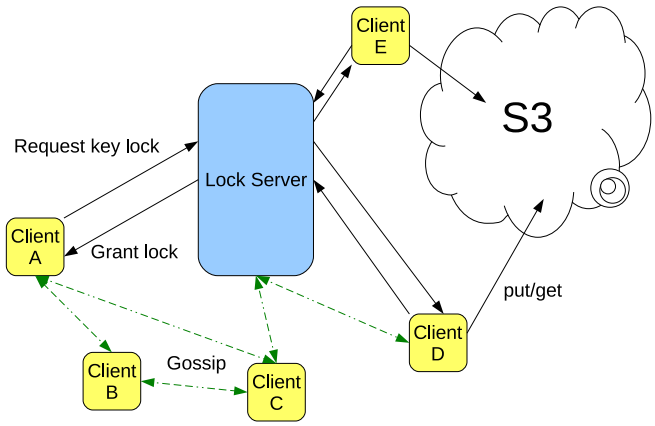

Figure 2. A high-level view of SILT . Here the different clients interact with a central lock server to acquire/release locks before being able to put/get data from S3. Additionally, the clients and the lock server gossip latest key-hash pairs amongst each other.

it sends the new value's hash in its release message along with the timestamp of the grant message that it received from the server when it acquired the lock on that key. Upon receiving the new hash value, the lock server issues a timestamp for that hash value and sends it back to the client piggybacked on the release grant message. The new key-hash-timestamp triplet is then spread to the rest of the clients in the system via anti-entropy gossip by the client and the lock server.

There are several consequences to these design choices:

- The lock server always has the latest version of key-hash-timestamp triplets. Thus by piggybacking hash and timestamp information on lock requests and grants, a client can be sure that the hash value it is using to verify read data corresponds to the latest hash known by the server.

- Clients can aggregate multiple keys in a single request which will only be granted when all the keys are available to be locked. However, key locks can be released individually even if they had been requested in groups.

- Clients only interact with the server in the context of acquiring/releasing locks and time-stamping key-hash values. All clients and the lock server gossip key-hash information, but that is not in the critical path of accessing data on S3. Additionally, after acquiring a lock on a set of keys, clients interact with S3 directly until that lock is released. After acquiring a lock, clients do not refer back to the server to verify data hash even keys were read/written multiple times. Since hash/timestamp information is piggybacked on top of lock-related messages, this reduces the communication overhead on the server and allows it to scale with the number of clients. The experimental section verifies this claim.

- Gossiping key-hash information allows the server to reconstruct its state in case of a crash. This is discussed in a later section.

Figure 1 shows the messages exchanged between a client and the lock server to acquire/release a lock as well as retrieve/update a key's hash information. Figure 2 shows a high-level view of the system, here multiple clients can be interacting with the server at once to acquire or release locks, additionally clients with locks interact with S3 directly, and the entire system disseminates key information via gossip.

3.3 Transactions

Multi-key locks allow us to implement transactions on top of S3's public interface. In addition to the traditional

`lock(), release_lock(), read(), write()` interface, SILT provides a separate transaction API. A client begins a transaction by specifying a list of keys that are involved in the transaction. The SILT client acquires a read/write lock on these keys and returns to the client a new transaction id. To read/write data in the context of a transaction, the client invokes specialized `transaction_read(), transaction_write()` functions. In the context of a transaction, when a client writes to a key, SILT translates that write to a new intermediate key. This allows a transaction to be safely aborted without affecting the values of the original keys. Naturally, reads in a transaction are routed to the intermediate keys.

A client can decide to either `commit()` or `abort()` a transaction. Committing a transaction to S3 involves writing the latest value of the intermediate keys to their corresponding keys. Since S3 guarantees availability and atomic updates on single keys, a client can repeatedly try to write to all the transaction keys to finish a transaction. This consequently means that a transaction can not be aborted once a commit has been issued even if the commit did not finish. Transaction aborts do not require additional processing besides releasing the acquired locks since all writes were to intermediate keys that do not affect other clients.

This design choice can result in an inconsistent state if a client crashes while committing a transaction. Failure tolerance and crash recovery are discussed in the following section.

3.4 Failure Tolerance

As other distributed systems, SILT must handle the failure of some of its nodes. Client failure can be troublesome because a failed client could have been holding some locks. Even worse, a failed client could have updated a key-value on S3 before releasing its write lock and updating the key's hash on the timeserver. Additionally, the lock server might fail and we need a way to recover the latest per-key hashes as well as handle existing locks when the server comes back up.

Byzantine failure and malicious behavior are not addressed in the scope of this project. Similar to the design of Chubby, the locks in SILT are *advisory* simply because the S3 API is accessible without SILT . So a malicious or Byzantine client can just corrupt the data on S3 directly, and thus such behavior is not discussed any further in this work.

3.4.1 Client Failure

To handle client failures, locks are issued with leases. When a client requests a lock, it can specify a lease period. The lock server enforces a maximum allowed lease period on all locks. If a lock lease expires before the client releases it, the server will send send an *expiry* notice to the client and release the expired locks. If the expired locks were read/write locks or a transaction, it could be the case that the client had modified the data on S3 before the lock expired. For that reason, the lock server tags the keys in expired locks as possibly inconsistent. When a client attempts to lock any of these in subsequent interactions, it will be notified that these keys are possibly inconsistent. It is then left up to the library user to handle possibly inconsistencies.

If the server were able to detect that a client has crashed (fail-stop), then it expires all its locks. If a client crashed but the server was somehow unable to detect its failure (crash failure), the locks held by that client will eventually be expired by the server when their leases run out. This prevents a failed client from halting the progress of the system as a whole.

3.4.2 Lock Server Failure

Handling clients' failures is presumably easier than server failure; after all, the server holds information about which keys are locked by which clients. I assume that a failed server is eventually restarted by a third-party. This could be either by human intervention, or a simple cron job. When a lock server restarts after failure, it has to restore the latest key-hash information, as well as handling existing locks.

A restarted lock server restores its key-hash information via gossip and the key-hash information piggybacked on incoming lock requests. If the server erroneously grants an incoming lock request with an old hash value, it later expires that lock if it learns about a newer hash for that value. If an erroneous lock had been granted and released, the key-hash information for that referred to by this lock are tagged as *possibly inconsistent* in subsequent requests by clients so that the end user can fix any possibly inconsistencies.

Restoring previous locks by a restarted server is done in a similar fashion. A server grants lock requests for available keys. If the server later receives a release message for a lock that it had not granted, and that release request conflicts with an existing granted lock, it checks the timestamp associated with that request. As mentioned previously, the timestamps in release requests refers to the timestamp of when the lock was granted. This timestamp allows the server to distinguish between locks that have been granted before the server crashed from messages that have just been delayed by the network. Using this timestamp, the lock server expires granted locks that conflict with the released keys if the keys in the release request have an earlier lock grant timestamp.

Restoring the state at a server depends on having honest and non-Byzantine clients. As mentioned previously, Byzantine or malicious behaviors are not considered within the scope of this project.

4 Implementation

A prototype of SILT has been implemented in Ruby and evaluated. Besides the lock server and the client library, a simple global node tracker was implemented to introduce nodes to each other when they start. The functionality of the tracker can be replaced by IP Multicast, but a tracker was implemented instead because the evaluation environment (Amazon EC2) does not allow IP Multicast.

Applications written against this library invoke the public SILT interface to acquire/release locks, begin/commit/abort transactions, and do reads and writes.

5 Evaluation

The SILT prototype has been evaluated on a local system and on 15 machines on Amazon EC2. 15 machines were used because that was the maximum allowed number of instances I could create at the time of evaluation. There are two main things I focused on in the evaluation; throughput of the system as the number of nodes increases, and the time it takes a restarted lock server to restore its missing state.

5.1 Throughput

As mentioned in the design section, clients are required to obtain locks on keys before reading or writing to them. This experiment measured the latency experienced by the SILT clients as the total number of clients varied. The experiment had 15 clients in total running on 15 EC2 machines with a new client joining every minute. Each client ran a program that did the following steps in a continuous loop:

1. Pick a random key from a fixed set of 100 keys
2. Acquire a read/write lock on that key
3. Write a random value to that key
4. Read the value of that key on S3
5. Release the lock

The two graphs in figure 3 show the total latency experienced per loop iteration per client. To put the numbers into context, the baseline latency for reading and writing to S3 is also measured. The top graph in figure 3 shows the results of running the experiment on EC2. Accessing S3 from EC2 to do a write then read took about 0.2 seconds on average. The top graph shows that most of the requests incurred little extra overhead by acquiring a lock first, and the addition of extra clients did not hurt the overall performance. A few

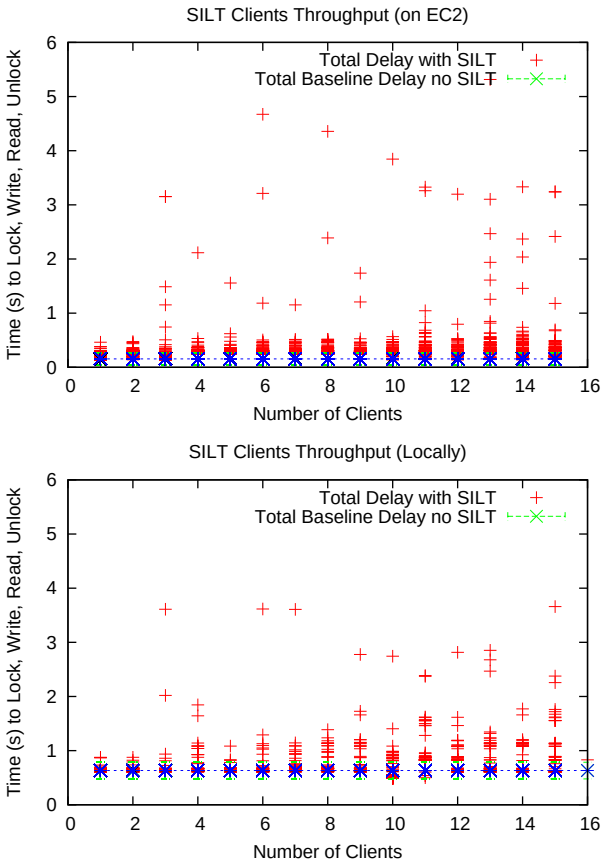

Figure 3. Total time required to acquire a lock, write data to a key, read that data back, then release the key evaluated as the number of clients varies.

requests incurred extra delays, these correspond to clients blocking on a lock that is held by other clients. As one would expect, the number of lock blocks increases as the number of clients increases since in each iteration clients choose a key to lock at random.

For comparison, the experiment was repeated on my local machine, there it took on 0.6 seconds on average to write then read from S3. The behavior experienced locally mimicked that on EC2 with the latencies shifted up as expected.

5.2 Restoring Server State

The following experiment tested how long does it take a server to rebuild its knowledge of the key-hash information and existing locks. This experiment highlights the advantages gained by piggybacking the key-hash information on lock messages. To start this experiment, a set of clients initialized a fixed set of 100 keys with random values such that each key was written to by all the clients, and each client

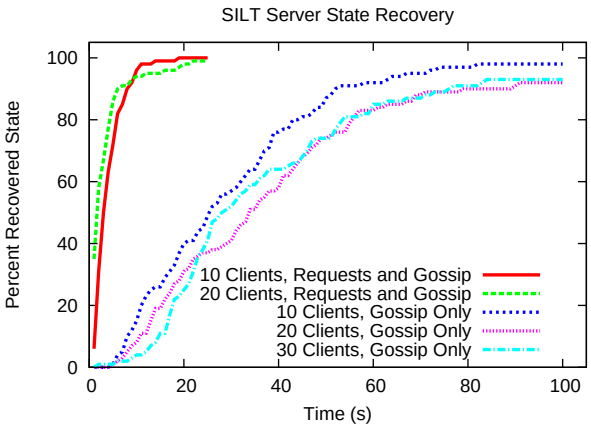

Figure 4. Time required to rebuild a restarted lock server's key/lock state.

holds the latest writes for a random subset of the keys. After initialization, the lock server is killed and restarted.

The graph in figure 4 shows the time it took the new server to restore the old state with and without piggybacking hash info on lock requests, and with a varied number of nodes. The gossip epoch in this experiment was 0.5 seconds (nodes picked a random peer and exchanged key information with it every epoch), and the client lock requests were issued in a loop program similar to that used in the previous experiment. As the graph shows, using gossip only, the server restores its state in longer time than with piggybacked hash info. Also, as expected, we get the typical gossip "S graph" for the propagation of new hash information through the system.

When key-hash information were propagated via lock messages, the server was able to restore its pre-crash state noticeably faster. This highlights the benefit of piggybacking key-hash info on lock messages.

6 Conclusion

In this project I investigated building a service to wrap Amazon S3's API and provide data integrity, locking, and transactional features on top of S3. By using a simple lock server and disseminating key-hash information amongst the clients, multiple end users can coordinate using shared S3 keys and detect when the integrity of their data has been compromised. A prototype of my system has been implemented and evaluated. The evaluation showed that requiring clients to check with the server only when acquiring and releasing locks reduces the overhead of the system and allows it to scale with the number of clients. Additionally, piggybacking key-hash information on lock-related messages resulted in speedy server recovery after failure.

7 References

1. B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The nfs version 4 protocol," 2000.

2. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 29-43.

3. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Copper Mountain Resort, Colorado, 1995.

4. M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 335-350.

5. Yahoo! Zookeeper: http://research.yahoo.com/node/1849

6. J. Li, M. Krohn, D. Mazires, and D. Shasha, "Secure untrusted data repository (sundr)", in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, p. 9

7. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*, 1987