Chained Distributed File Storage

Rohin Patel - h2b9

Sergey Skovorodnikov- s8z8

Jonathan McGrandle - y4n8

Abstract

Considering file storage systems such as Dropbox, our goal was to design a similar system that had less of a reliance on a centralized server instance. To that end, we designed a system that is a primarily write once, read multiple (infrequent) times using a network of clients that must each allocate a fixed amount of space for its peers to store files on. In return, each client is allowed to store data on other clients equal to the amount of space allocated. In essence, clients are trading storage space between one another. Storage space and writing transactions are presided over by a centralized server instance.

Introduction and Motivation

The motivation for the development of this system came from exploring file storage systems such as Dropbox or Google Drive while attempting to rely less on a centralized server. It has been developed to behave as a distributed backup system intended to be used in an environment where each client can be trusted not to tamper with the files that have been stored on their computer.

The system adopts a peer-to-peer approach where each client offers a fixed amount of free space to the network and can, in turn, take advantage of the space available on other clients. We use a command line interface, allowing the user to store a file to other peers in the network, and then retrieve or delete a file from a peer that a file was previously stored on. In addition, utility functions allow the end user to determine the locations of a stored file, as well as determine the files and locations that other peers have stored locally.

Design

The system is comprised of two types of actors. First, there is the command and control center (C&C) which is responsible for keeping track of the client nodes and providing client details when appropriate. Second, there are a number of client nodes who offer free space to other clients on the network. Specifics for these two key parts can be found below.

Command and Control Center (C&C)

The C&C specifies a number of constraints that client nodes must abide by, such as storage space, the maximum file size supported, and the maximum number of replications (**max replications**) allowed in a single transaction. The C&C is responsible for tracking the latest storage space information for each of its clients (both storage available on, and storage used by each of the clients) and keeping track of a heartbeat from each of the clients. The C&C is authoritative when it comes to tracking storage and transaction information.

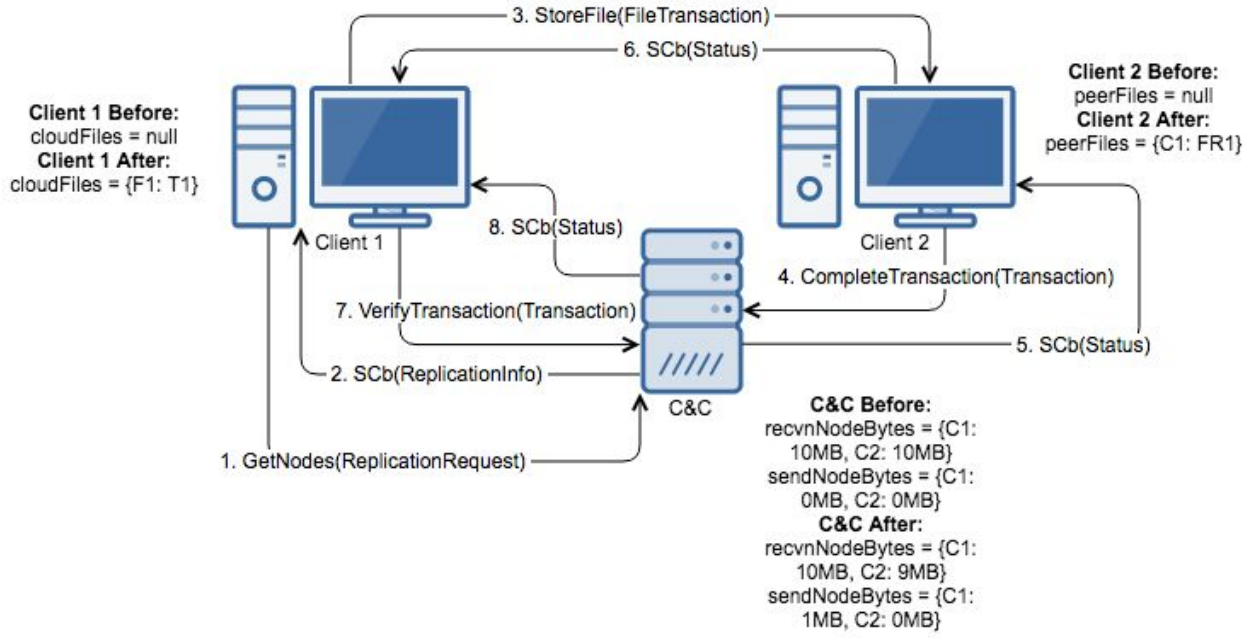
Client Nodes

Client nodes register their system with the C&C to inform other clients that it has storage available for data replication.
These are the commands that a client can issue to interact with the network of other clients:

- **Store a file:** a client queries the C&C with the size of the file, and the number of replicas it would like the file to be stored on (usually a fixed number). The C&C will reply with a list of peers (and their addresses) capable of storing the file. The client will then encrypt the file and send it directly (i.e. peer to peer) to the peers returned by the C&C. The client is responsible for keeping track of files that it stores on other peers. The sender then hashes the file and concatenates it with with the unique client ID and the filename to produce the filename that will be used to store the file. The receiver will verify the transaction with the C&C and the C&C will update storage information for both of the clients involved in the transaction.
- **Retrieve a file:** a client consults its locally stored file map and will contact one of those peers. The peer will respond back with the encrypted file, which the client then decrypts and stores.
- **Delete a file:** a client will query the C&C and request that a file that it has stored on another peer be deleted. The C&C will queue the deletion transaction and expect the client to directly dial the peer client informing that it wishes a file to be deleted. The peering client will dial the C&C to verify that the file is up for delete and the C&C will update its storage information for both of the clients involved in the deletion transaction.

Write Operations

The C&C mediates write operations between peers. Clients that wish to write files to peers are required to query the C&C with the file size and replications desired. The C&C will then initiate at most **max replications** of transactions each with a different receiving client IP and return the transaction information to the sending client. Each transaction is then completed using an RPC chain from the sending client to the receiving client and then from the receiving client to the C&C to fully complete the transaction. Once the sending client receives the initial transaction information from the C&C it will then contact each of the receiving clients and send an encrypted version of its file, passing the relevant transaction information along. The receiving client is required to contact the C&C to verify the file size of the received file and the corresponding transaction. Transactions issued by the C&C timeout if the entire write transaction is not completed within 5 minutes. It is important to note that file transactions are made on a best effort basis and that once a transaction has been fully completed and verified by the C&C all clients are effectively expected to adhere to the C&C contract.



Additional diagrams depicting other scenarios can be found in the Appendix

Consistency

Since transactions are performed using an RPC chain, in order to handle consistency in the event of a failure, transactions are forced through a series of milestones that help determine the appropriate recovery procedures in the event of a failure. A transaction must proceed through **all** of the milestones in order for it to be deemed successful.

Detailed File Sending Procedure (Similar procedure for deletion)

1. Sender dials the C&C with a requested file size and replications requested.
2. The C&C checks to make sure the requesting client has met the network constraints (file size, replications, storage) and finds other peers that can meet the file size request. The C&C will issue transactions each with a local timestamp.
3. **Milestone 1:** The sender places each of the transactions into an embedded key/value database (BoltDB) under the bucket “Unverified Sent”.
4. The Sender dials each of the receiving clients and sends the file.
5. **Milestone 2:** The receiver immediately writes the file to disk (before verifying with the C&C).
6. **Milestone 3:** The receiver then places each received transaction into the key/value database under the bucket “Unverified Received”.
7. The receiver then dials the C&C. If it is unable to reach the C&C, the file is immediately deleted and the sender is notified.
8. If the receiver successfully calls the C&C, the C&C will verify the transaction.

9. **Milestone 4:** If the transaction is verified successfully by the C&C, the receiver will move the verified transaction into a different bucket labelled “Peer Stored Files” indicating that the transaction is completed. Otherwise, the transaction is deleted from the “Unverified Received” bucket.
10. The sender is informed of the status of the transaction. If it is a success the sender will move the verified transaction into a different bucket labelled “Cloud Stored Files” indicating that the transaction is completed. Otherwise the transaction is deleted from the “Unverified Sent” bucket.

A similar process occurs when clients request the deletion of files. It is important to note that the buckets described above allow each client to determine the appropriate steps in order for the system to reacquire consistency in the event of failures. The full importance of the milestones are described in each of the failure cases below. In most cases network failures and client restarts can be treated in the same way.

Failures When Issuing Transactions

The C&C and the sending client are the only parties involved when transactions are being issued. Should the sending client fail or a network partition occur between the C&C and sender, the issued transactions will timeout on the C&C and be rendered invalid. In this scenario the sending client will have to redial the C&C to request new transactions.

Failures While Sending Files

From the sender’s point of view, a network partition or client failure during the course of the RPC to a receiving client will appear the same. Thus, prior to sending, the sending client must place the transaction in the database under the “Unverified Sent” bucket. After a transaction has been placed in this bucket, the sender will be able to verify the state of the transaction with the C&C should any failure (network partition or restart) occur during the course of the RPC on the receiving client.

Failures While Verifying Transactions

Once the receiving client acquires the file, it immediately writes the file to disk, then places a record of the received transaction in the “Unverified Received” bucket in the database. It will then dial the C&C to verify the transaction. In the case where the C&C could not be reached, the receiver discards the file and notifies the sender that the transaction could not be completed. However, a network partition or a C&C failure during the course of the RPC will appear the same to the receiver. The receiver will not know whether the transaction verification step occurred or not. In this case, the receiver is has stored enough information to later verify the transaction with the C&C.

Should the receiving client restart, an index of every file stored is created and any files that are not in either the “Peer Stored Files” bucket or the “Unverified Received” bucket are deleted. Any transactions left in the “Unverified Received” bucket are verified with the C&C. If

there is any sort of failure while dialing the C&C, transactions are left in the “Unverified Received” bucket until they can be verified at a later time.

C&C failure

In the event of a C&C failure, writes are disabled between clients (as they must first receive a series of valid transactions from the C&C before they can write). If the C&C crashes in the middle of transaction verification procedure, receiving clients that managed to begin the verification process (i.e. execute the RPC on the C&C) are expected to keep their files and try to verify at a later date. If a receiving client have tried to dial the C&C and is certain that the RPC call has not begun execution on the C&C, the sender is informed that the transaction has not been completed and is expected to request a new transaction once the C&C has recovered.

Client Failure

In the case of a client failure, all remote files from other clients that it stored locally are unavailable to the rest of the network. The C&C will be responsible for letting the other nodes on the network know that it has found a dead node and that they need to handle it accordingly.

A client, C1, is defined to be dead permanently if it does not send a heartbeat to the C&C after a period of Y time. The C&C will then inform other clients of this failure. These clients will then replicate their files to other clients that are available. If, after Y time, C1 reappears it will be instructed to clear all remote files that it had stored as it is assumed that other clients will have replicated their files to a new client within the system. However, the files of C1 that are stored elsewhere will be not be deleted.

There is a second milestone, that occurs after Z time, where Z time > Y time. After Z time, clients that were storing remote files of the dead client will be instructed to clear any files belonging to that dead node.

File Namespace Handling

In an attempt to ensure that there will be no file naming conflicts when storing a file on another peer, files have been named as a hex encoding of their hash concatenated with the originating client id and the original file name. The pipe (‘|’) character is used to delimit these three fields.

File Versioning

File versioning is not explicitly handled by our system. If the sender creates a new version of a file, the naming convention mentioned above should create an new unique name which can then be stored without conflict on receiving clients. The sender is responsible for ensuring that they retrieve the correct files.

Network

All communication between clients is done using RPC. While there is a central C&C server that keeps track of live client nodes and is required to do write operations, the network topology is more of a chain then a star. This is due to our verification process described previously, which chains confirmation steps instead of broadcasting them in a way two- or three-phase commit does. For example, before sending a confirmation to a sender the receiver has to verify the transaction with the C&C, which means two clients (sender and receiver) are waiting for the reply of the C&C.

Distributed State

Distributed state includes:

- A list of live clients,
- Space available on each client and
- State of transactions.

All state is directly maintained by the C&C and is served to clients on demand. Clients have access to state of the transaction, but do not directly affect it. The C&C is the only component of the system that is able to directly change the storage space and transaction state based on clients' responses to the C&C. Clients also know about some portion of the live clients in the network when they request replication from the C&C. Since the file transaction occurs as an RPC chain, each client really only knows the status of the node that it is dialing (i.e. the connection is one way).

Implementation

All code in this system was written in the Go language. Both the C&C and client nodes make extensive use of an embedded key/value database called BoltDB that provides ACID semantics. BoltDB allows the creation of "buckets" (equivalent to maps) within a memory mapped file and provides functions that perform atomic transactions during any sort of modification to the database. BoltDB is used to store important state information on both the C&C and the client.

System API

Clients in the system are located on user machines and we have complete control over their implementation. Hence, we trust that clients follow the API. Therefore, it is guaranteed that they implement this API. The same is true for the C&C. Clients assume that other clients in the network are adhering to the contract of the network (for example, ensuring files are kept stored after verification). Clients assume presence of the C&C because it is required to do any send operation, however, they are expecting that other peers are able to respond to read operations without C&C's support. Clients also assume that C&C's space information is valid and up to

date. This means that whenever a client requests the C&C for a replication list, clients in that list actually have the requested amount of space. Clients assume that once a transaction has been completed and verified, the receiving node will keep the file stored unless deletion is requested

Clients are sequential in their replication of files: only a file at a time can be replicated per node since it is requested manually by the human user and calling to other clients is done serially. However, each client can receive multiple files at a time.

Client-Client API

- `StoreFile(ft *FileTransaction, reply *shared.Status)`
- `RetrieveFile(fr *FileRequest, reply *FileTransaction)`
- `DeleteFile(txn *shared.Transaction, reply *shared.Status)`

The **FileTransaction** struct contains information such as the actual encrypted file as a byte array, sender address, sender id, filename, transaction id and a up to date GoVector log entry. ***Status*** carries back a message and success code (0 or 1) as well as GoVector log entry. **FileRequest** contains requestor id, requestor address, name of the file and transaction id.

Client-C&C API

The following operation is used by the C&C to notify a given client about other dead clients. **DeadNodeNotification** contains IP and port of the client that died.

- `Dead Node Notification(args *shared.Dead Node Notification, reply *string)`

C&C-Client API

Clients are able to contact the C&C via an RPC-based API as well:

- `Heartbeat(args *shared.Heartbeat, reply *shared.Heartbeat)`
- `Getclients(args *shared.ReplicationRequest, reply *shared.ReplicationInfo)`
- `VerifyTransaction(args *shared.Transaction, rep *shared.Status)`
- `CompleteTransaction(args *shared.Transaction, rep *shared.Status)`

ReplicationRequest contains the file size and the desired number of replications. **ReplicationInfo** has the list of addresses of clients available for replication. **CompleteTransaction** is the RPC executed by receiving nodes to complete a file transaction received from a sender and update the storage space of all clients involved in the transaction atomically. The **VerifyTransaction** is the RPC executed by any client to determine the status of a file transaction. Both **CompleteTransaction** and **VerifyTransaction** have analogs that deal with file deletion.

Evaluation

ShiViz Integration

ShiViz was integrated alongside the transactional procedure that occurs during a file transfer. Major events include transaction requesting/issuance, sending a file to the receiving client, verification with the C&C and the reply from the C&C. Minor events include things such as clients dialing one another, milestones along the transactional process such as adding the transaction to “Unverified Sent” or “Unverified Received” and saving/removing files from disk. These minor events help to trace possible failure cases that can occur (for example, what happens if a failure occurs on a receiving client before a disk write or after a disk write).

Tests

Evaluation of this system consisted of manually testing the scenarios listed hereafter.

No Network Failures, No Client Failures

Basic functionality of the system (send and retrieval) was with three clients, C&C and no failures. Using the “SEND” command, files were encrypted and successfully sent from one client to two others. Using the “GET” command, the client was able to retrieve both of the files, decrypt it and store it locally. The C&C updated the storage accordingly in and kept a record of the transaction. The “DELETE” command was successfully used to delete one of the replicas in the network.

Crashing receiver after receiving file and writing file to disk, but before logging of unverified received transaction

This scenario is rather significant as the receiver might crash at this moment, but has no record of the transaction ever being received. This scenario was tested using 3 clients and 1 C&C where one of the clients tries sending a file to the other two (replication factor = 2). To get the two clients to fail at this specific moment, an os.Exit() function call was inserted right after the receiving clients wrote the file to disk.

Result

On restart the receiving client ran its file indexer and removed the correct file. During the transaction, the sender noticed the error with its RPC call to the receiving client(s) and immediately dialed the C&C to see if the transaction had been verified. Since the receiving client crashed before it could verify with the C&C, the C&C replied that the transaction was not verified. No storage updates occurred.

Crashing receiver after receiving file, writing file to disk and logging of unverified received transaction, but before verifying with C&C

In this scenario, the sending client will do the same as in the previous test case. However, the receiving client is expected to dial the C&C as it has a transaction in the unverified portion of its log as well as the file, but is unsure of whether it managed to dial the C&C to verify. Even though the os.Exit() function call was inserted right after the update to the unverified log, this failure case is supposed to handle failures that could occur when the receiving client manages to begin the verification of the transaction on the C&C but is unsure of the status of the verification (either due to the receiving client failing, or a network partition occurring between the the receiver and the C&C).

Result

On restart, the receiving client correctly dialed the C&C to verify if the transaction completed and took the appropriate measures (kept the file if it was verified on the C&C, deleted if not). The sender client also dialled the C&C to verify the same transaction.

Crashing C&C after receiver has received file, wrote to disk, logged of unverified received transaction and has started execution of RPC CompleteTransaction on C&C.

The C&C is crashed somewhere in the middle of its verification process. This occurs after the receiver has received the file and it has begun the execution of the RPC on the C&C. Since the system uses boltDB to ensure atomicity, the transaction should not be in an incomplete state. Either the storage is updated for both nodes, or the storage is not updated at all.

Result

As expected, the transaction was not left in an incomplete state, and the storage for both nodes was not updated. Both the sender and receiver dialed the C&C once it restarted and were updated as to the status of the file.

Limitations

After weighing the advantages and disadvantages with various distributed system designs we were left with the following limitations in our current system.

Unable To Control the Environment of the Receiving Client

When a receiving client receives a file and saves it, there is nothing that can be done to ensure that the receiving client actually keeps that file on its disk and that nothing (maliciously, or otherwise) has removed the file from the filesystem after a transaction has been fully verified by the C&C. The sender has no recourse in that case. One strategy of handling this would be to penalize the receiver if a sender requests a file and the receiver does not have the file available. Some possible options could be disabling writes to and from that client or by removing storage available for that client equal in amount to that of the file size of the file requested by the reader.

In addition, the availability of a client's files depends on the availability of the peers that are connected to the network and the number of replications that are conducted for a particular file.

Performance

Our design uses a library (BoltDB) that provides an embedded key/value database on both the clients and the C&C to ensure ACID semantics. However, since any update to this persistent key/value stores ensures that a write to disk must happen before the state of the database is changed and the extensive use of this library in our system, there is definitely a performance penalty.

All maps are stored in a single boltDB file and locked with a single mutex. Given that some operations are independent of each other (sending and receiving for example) it would have been better to split the database file into two separate files so that there would be less contention for the same lock.

With the current design, each client can only perform replications of a single file at a time. This is a potential performance issue if clients wish to store many files using this system.

Relying on Disk Storage to Maintain State

Both the C&C and clients make extensive use of disk storage to recover state in the event of failure. However, this introduces reliance on the hard disk itself. The reliance on a single hard disk could be alleviated by creating replications of the system state periodically.

File Versioning

The system does not have a concept of file versioning in a typical sense. Clients are able to tell the difference between files using a hash: before sending the file its hash is calculated and appended to the filename. While this allows clients to tell two files apart, it does not provide any information about version of the file and whether the newly received file is newer or older than the one currently stored.

Security Issues

System users are storing their files on peers' computers and security is a concern. Before the file is sent out it is encrypted using AES cipher with Cipher Feedback (CFB) mode, 128 bit blocks and 16 byte Initialization Vector (IV). CFB mode adds security by confusing the structure of the original plaintext (file) over the entire ciphertext. Nonetheless, the user who is storing these files can attempt decryption. While AES cipher with CFB is a very secure encryption method, having several files encrypted the same way and almost unlimited amount of time may provide adversaries with useful information about cipher structure. In addition, the communication between clients is not encrypted or signed. If C&C is spoofed or hijacked by an adversary it may direct writes to malicious clients that don't comply with our protocol. This may have devastating consequences to the system in terms of loss of privacy and consistency of information. We have attempted to implement an authority server that would be able to issue a security certificate for the C&C to clients. Using that certificate the C&C would be able to sign all messages it sends to clients and clients would be able to verify C&C is the sender. However, due to time constraints we were not able to integrate the authority server into the final system.

Discussion

Tradeoff Between Consistency and Performance/Efficiency

One of the most challenging problems we faced when designing this system was ensuring that file transactions were processed consistently across the entire network. There were a number of opportunities for a transaction to fail and we had to ensure that a failed transaction was recognized by the C&C and by each participating client. At the same time we wanted to limit how much extra work a client has to do to ensure that they remain consistent with the rest of the system. To better understand the problem, let’s consider an example. Imagine a client C1, wants to store a file on 2 other clients. First it needs to ask the C&C for 2 clients, and then it has to transfer the file to each of those clients. Now, let’s consider where this fail and possible consequences:

- The C&C may be unreachable when C1 asks for a list of clients. This really isn’t a big issue consistency wise.
- C1 transfers a file to one of the clients, who crashes before storing the file. At this point, C1 doesn’t know if the file was successfully stored or not.
- C1 transfers a file to one of the clients, who stores the file and then crashes. C1 doesn’t know if the file was successfully stored or not.
- C1 transfers a file to one of the clients, and crashes. C1 doesn’t know if the file was stored, and the receiving client isn’t sure whether it should keep the file.

Most of these failures can be handled by adding callbacks. However, adding callbacks has two immediate effects. First, something may happen between the time the callback is sent and the callback is received thereby adding more uncertainty to the system. Second, adding callbacks

increases the traffic in the network. As a result, we had to determine an appropriate trade-off. Ultimately, we elected to add as many callbacks as we could. While this does increase the network traffic, it also creates a consistent system.

Handling Files of Dead Clients

Another issue which we spent considerable time discussing is how to handle clients who are unavailable for a long period of time. We had to consider what to do with a) the files belonging to the crashed client and b) the files stored on the crashed client. Problem A seemed to present two options. First, the C&C could simply tell all of the other clients to delete those files. However, if you delete all of the files and the crashed client comes back online then they have lost all of their backups. Alternatively, you could ignore the crash, but this leads to wasted space if the crashed client never comes back online.

Problem B also had two options. First, you could assume that the crashed client will come back online. However, there is the obvious issue that occurs if this assumption is wrong. In addition, if the client were to come back online, then they may be storing outdated files. Alternatively, you could alert all other clients to this crash and they could re-replicate their files. This makes the assumption that the original file is still available.

In the end we opted to keep the two milestones that were originally proposed. Milestone Y is the first Milestone to occur after a client node failed to issue a heartbeat for a pre-defined period of time. When Milestone Y occurred all other clients in the system were told to re-replicate their files to an additional client. Then Milestone Z occurred, after yet another pre-defined period of time. After Milestone Z, the other clients were told to delete the files they had stored for the crashed client. Should the dead client come back online after Milestone Y, they were asked to delete the files they had stored for others. Should the crashed client come online after Milestone Z, they were told to start fresh as their files had been removed from the network.

We did make note of a potential flaw in this system. Hypothetically, a malicious client could store a group of files on the network and then ‘crash' before storing files itself. It could then send a heartbeat to the C&C just before Milestone Z, which would effectively keep its files in the network. By repeating this heartbeat process, the malicious client is able to store a selection of files on the network without being required to store anything itself. We saw no way of avoiding this while working under our own personal constraints, and thus chose to ignore this issue until a later date.

Allocation of Work

Sergey

Worked on logging, file indexing and node recovery as well as encryption and security component. Also initially we planned to add security aspect in the form of authority server that is issuing certificates to clients to verify the C&C. However, due to limited time that feature was left incomplete. Certificate issuing and validation is implemented without integration and without the authority server.

Jonathan

I helped create many of the methods used to transfer files between clients, and keep them connected to the C&C. More specifically, I initially wrote the client helper methods for requesting a replication from the C&C, and replicating a file to a single client. In addition, I worked to create the client RPC methods to retrieve a file from another client, store a file to a client, and handle a DeadNodeNotificiation from the CNC (much of which was changed when we shifted towards storing maps on the hard disk). On the C&C side of the project, I worked to create methods that would scan for dead clients and handle them appropriately, which included notify other clients that a client had gone offline. I also worked on C&C RPC methods, and initially created the RPC method a client would call to initiate the replication process.

Midway through our project we also chose to change the client heartbeat from UDP to RPC, allowing us to easily identify when the C&C went offline. I was responsible for implementing this change, but again, much of this code changed when we shifted towards storing client state on the hard disk.

Rohin

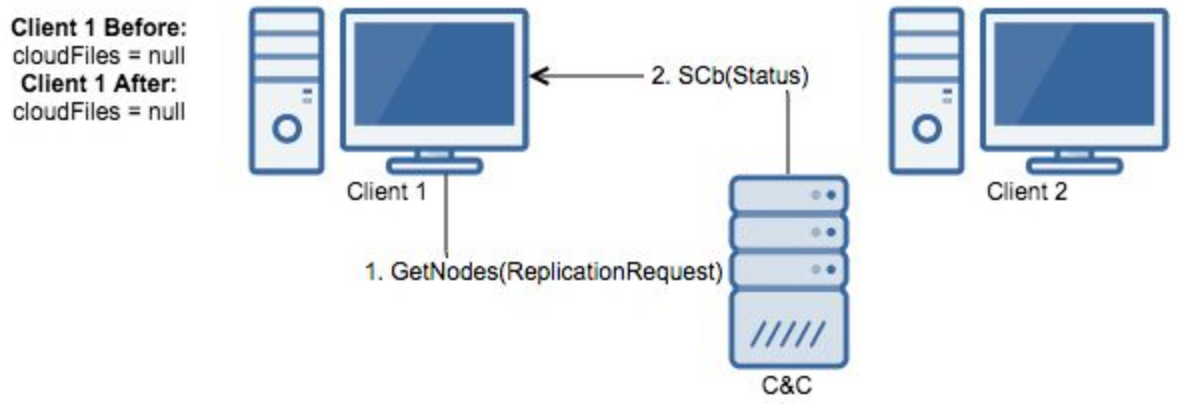
I wrote most of the code dealing with the file transaction system and I wrote the code that dealt with handling user commands from the command line. I integrated both boltDB and GoVector with the file transaction system on both the C&C and the clients. I also performed a lot of bug fixing and testing.

Appendix

The below diagrams illustrate the series of RPC calls made in different scenarios. This is a very high level representation, and many details are omitted for clarity.

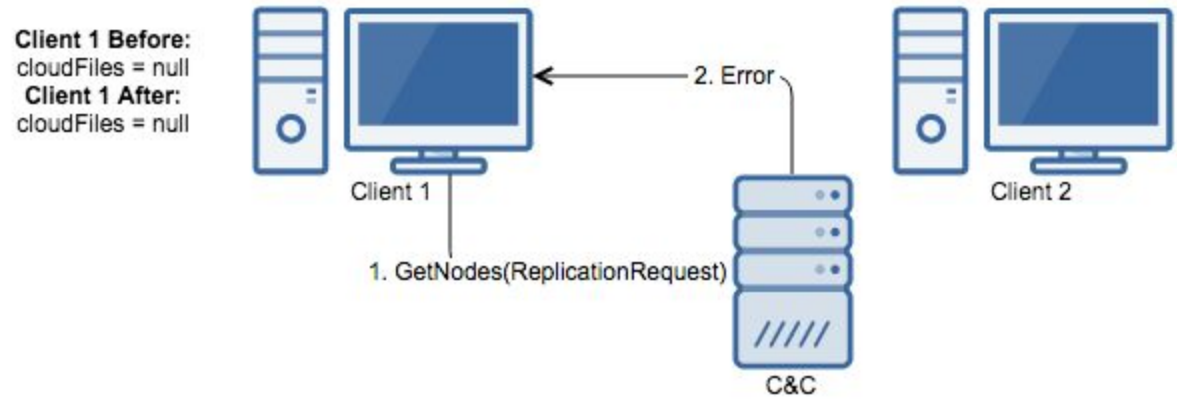
Attempting to Store An Ineligible File

A file may be considered ineligible if it exceeds the max file size, or if there are no clients with enough space to store it. In either event, the Status sent back from the C&C alerts the Client of the issue.



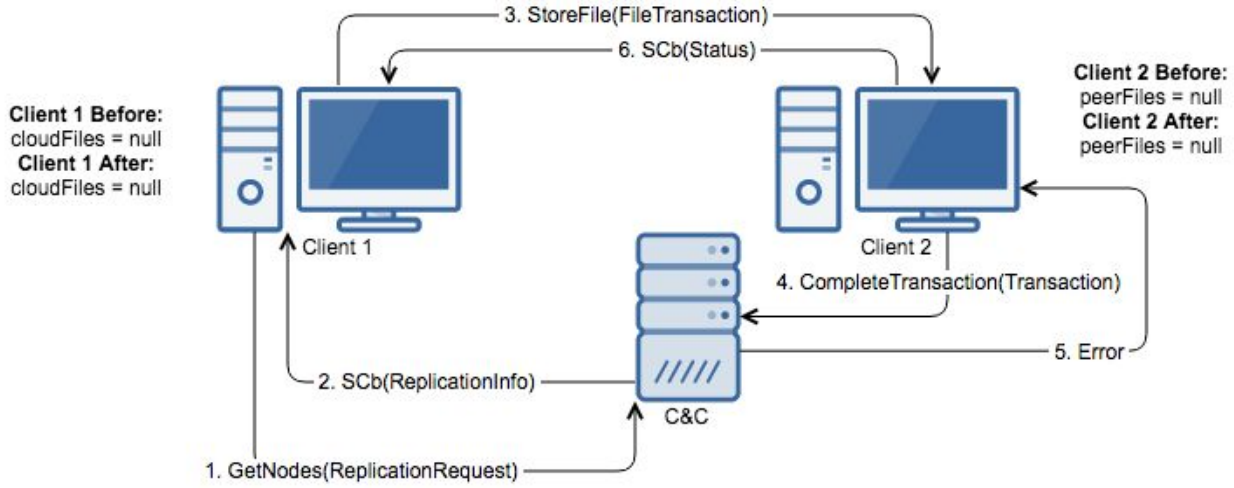
Attempting to Store A File With An Unreachable C&C

A C&C could be unreachable for a variety of issues. For example, there could be a network error on the client's side, or a network error on the C&C side. Additionally, the C&C may have crashed.



Unreachable C&C Partway Through A File Store

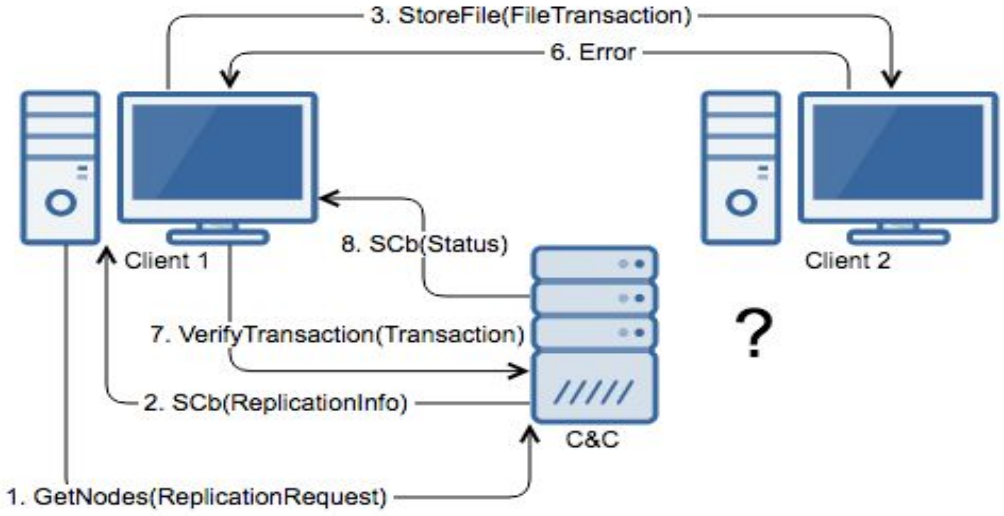
If Client 1 attempts to store a file, and the C&C crashes midway through this process, then ultimately Client 1 will be alerted by the Status coming back from Client 2. The file transfer will then be considered to have failed.



Storing A File When The Receiving Client Crashes

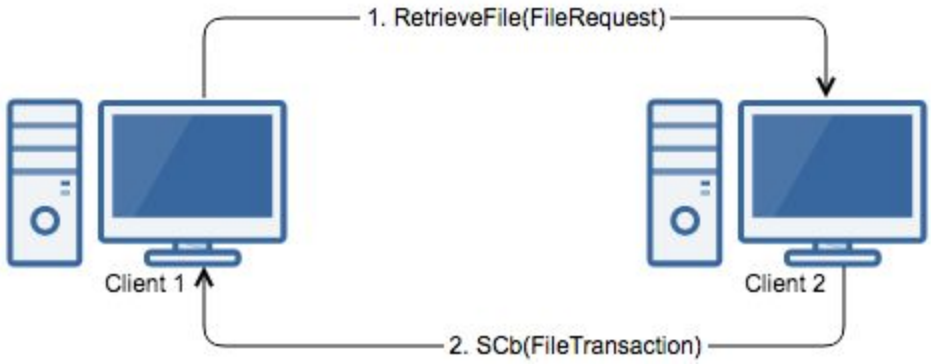
Should Client 2 crash after receiving the StoreFile RPC request from Client 1 then there is a lot of uncertainty. It is unknown if Client 2 stored the file and then crashed before informing the C&C, if Client 2 stored the file and crashed before informing Client 1, or if Client 2 never even stored the file.

To resolve this issue, Client 1 will ask the C&C if the transaction was ever completed. Client 1 will then take the appropriate steps.



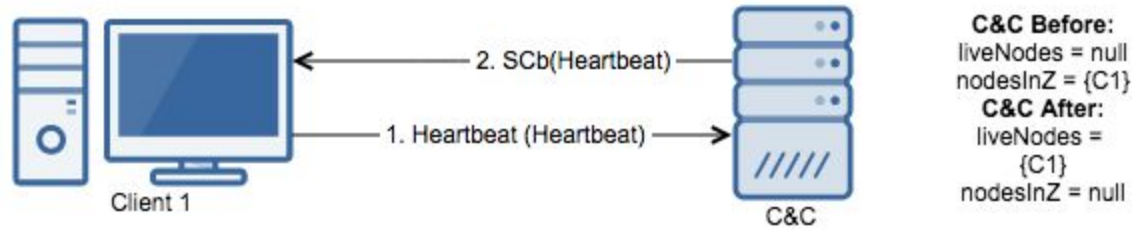
Retrieving A File

The retrieved file's content is encrypted, stored within a **FileTransaction** struct, and sent back from Client 2. Client 1 will need to decrypt the contents.



Connecting to The C&C After Crashing

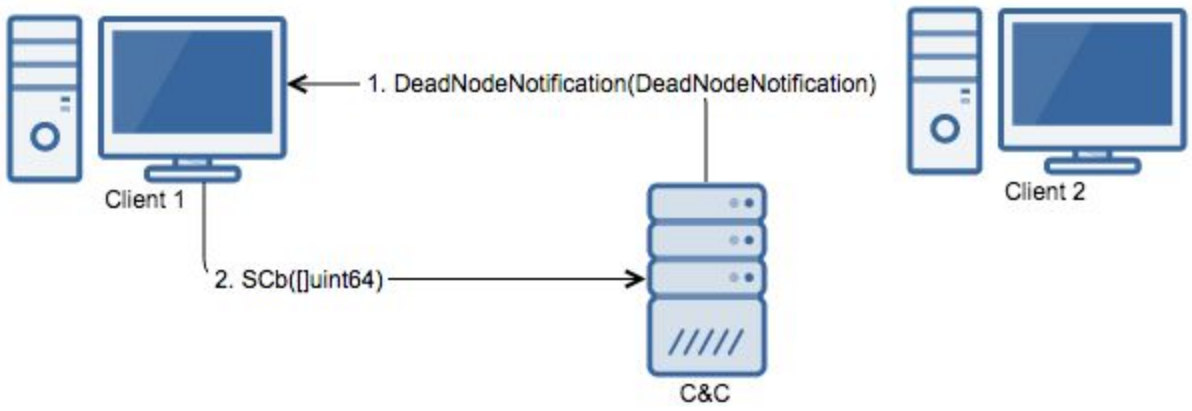
A client who crashes is able to reconnect by sending a heartbeat back to the C&C. The C&C will then take the appropriate actions to reinstate the client.



C&C Alert of A Dead Client

Should Client 2 stop sending a heartbeat to the C&C, the C&C will send a **DeadNodeNotification** to the other clients informing them of a) which client has died and b) which milestone that client has past (Y or Z). Client 1 will then take the necessary steps.

If Client 2 has passed Milestone Z then Client 1 may respond with a list of Transaction IDs. These ID's correspond to the files from Client 2 which have now been removed. The C&C will need to verify these Transaction IDs and take note of the newly acquired space on Client 1.


1. DeadNodeNotification(DeadNodeNotification)
Client 1
Client 2
2. SCb([]uint64)
C&C