Byzantine Federated Storage

David Vorick
Nebulous Labs

September 12, 2014

Abstract

We explore a method of randomly distributing files to hosts in a dynamic system, in a way that prevents malicious hosts from manipulating the distribution of files on the network under certain constraints.

1 Introduction

We wish to build a network of hosts and files, where every host stores a set of files, and every file is stored by exactly one host. This network is dynamic, meaning that files can be added at any time, and hosts can be added at any time. We wish to use this network to store arbitrary data, however we also wish to use this network as a basis for consensus by proof of storage (a host is allowed to participante in consensus if they are completing a sufficient volume of proof of storage). In protecting consensus, we only wish to make sure that no malicious party can gain control of a majority of the voting power.

Assumptions:

- No party will ever control greater than 50% of the raw storage on the network.
- All non-malicious files are fully compressed and non-redundant.
- There is a mechanism of proof of storage that requires a host to demonstrate that they can summon all of the data they are required to store.
- No party can afford more than 50% of the networks sum resources. This means that a party cannot do a 50% attack by buying enough storage hardware, and cannot do a 50% attack by renting 50% of the space available on the network, and cannot have some combination of each that exceeds 50% of the networks total resources.
- Renting a volume of storage on the network for a given period of time is at least as economically expensive as owning or renting the same volume of raw storage for the same period of time off of the network.

In a practical environment, the third requirement may be difficult to achieve, however for simplicity we maintain it as a feature of the network.

From these assumptions alone, we wish to build a network that satisfies the following properties:

- Every file is stored on a host
- The number of files each host stores follows a normal distirbution.
- A set of hosts controlling less than 50% of the raw storage on the network cannot appear to control greater than 50% of the raw storage on the network without spending more money than it would cost to merely own 50% of the raw storage on the network.

In proving the third item, I establish that following these two conditions is sufficient:

- A set of hosts less than 50% of the network cannot probabilistically have a greater than 50% of the files they have uploaded under control of their machines at any time.

- A set of hosts controlling less than 50% of the files on the network cannot perform any action which results in the expected percentage of files across all machines under their control to exceed 50% of the files under their control.

Because we assume that the mechanism behind proof of storage does its job properly, I hold that the only way for a party to appear to control greater storage than it actually controls is to upload some file to the network which can be calculated, but which only the party knows how to calculate. In this manner, the malicious party can upload 'fake files' to the network, and if these fake files land on the hosts in the party's control, the host can pretend to store the file (thus appearing to have that raw storage) while actually calculating the data to do the proof, instead of storing the data. We do allow artificial inflation of storage under one condition: the cost of artificially inflating apparent storage is more expensive than the cost of genuinely inflating storage.

If the only way to inflate the apparent storage is by getting fake files on machines under malicious control, we must prove that under the replacement conditions, a host containing little enough economic power to control 50% of the files or storage (exclusive or) on the network will not be able to inflate their apparent storage above 50%. If a host controlling less than 50% of the network cannot probabilistically have 50% of files landing on machines in their control, they can only fake storage on less than half of the files that they are uploading. Because files cost money equivalent or greater than the cost of raw storage, by being unable to probabilistically get fake files on to their machines at a rate greater than 50%, they cannot save money by using the fake files technique when they control a great number of hosts. So while the malicious party is able to appear that they have more raw storage than they actually have, their expenses are too high and thus we can rely on our assumption about the attackers limited economic power. Therefore, if an attacker cannot probabilistically control files they upload at greater than 50% when they control less than 50% of the machines, they cannot appear to store greater than 50% of the network by controlling a large volume of hosts.

The only other approach to faking storage is to own a large number of files, and to add machines to the network in a way that allows the machines to consist of more than 50% fake files. At that rate, apparent storage can be added to the network at a cost less than raw storage. By preventing this from happening, the party has no other way to increase their apparent storage beyond 50% of the network that is within their assumed economic power.

The rest of the paper shall then design a system that enforces the following conditions, given the prior assumptions:

- Every file is stored on a host
- The number of files each host stores follows a normal distirbution.
- A set of hosts less than 50% of the network cannot probabilistically have a greater than 50% of the files they have uploaded under control of their machines at any time.
- A set of hosts controlling less than 50% of the files on the network cannot perform any action which results in the expected percentage of files across all machines under their control to exceed 50% of the files under their control.

2 A Simple Solution

A simple system that upholds these properties is a rendezvouz hashing scheme. Every host is added to the network with a trusted random seed, and every file is added to the network also with a trusted random seed. The source of these seeds is outside the scope of this paper (though something like the hash of the next future bitcoin block could be used). To determine which host gets a file, each host gets a number determined by taking the hash of it's trusted random seed + the file's trusted random seed. The host with the largest number (largest hash) wins the file.

This satisfies the condition that the number of files each host stores follows a normal distribution. We can also get the condition that every file is stored on a host by requiring that at least one host be on the network at all times.

The remaining conditions are slightly trickier, as hosts and files can attempt to manipulate the trusted random number by getting re-rolls, whereby a file or host joins the network for a short period of time, and then stays for a longer period of time if the random number is favorable. The solution for files is easy: files must pay for their storage time before the random number is rolled, and once placed on the network, a file cannot add more to the storage balance, nor can the storage balance be recovered. This takes care of the third condition. If a set of hosts less than 50% of the total hosts uploads a file, it will probabilistically end up on a machine that is not controlled by the hosts. The hosts will have to pay more for uploading fake files than they will economically recover, which keeps the network protected.

The fourth condition needs an alternate solution, because hosts can withdraw and rejoin at-will; nothing is forcing a host to stick around and complete storage proofs. In situations where the number of random files on the network is high, a joining host can re-join multiple times to get a favorable set, and then can stick around after getting the favorable set, violating condition 4. This can be stopped by forcing the host to make a down payment securing him to finish the proof of storage. He only receives his down payment back linearly as he completes the storage proofs. The down payment is large enough that the cost of leaving is greater than the probabilistic reward of performing a re-roll, then this condition add safety to the network. The required minimum down payment can be calculated by: Espected return on 1 reroll assuming that 50% of the files are controlled by the party focing the re-roll. Expected return on 1 reroll is dependent on the number of files on the network, and the number of files being sent to the host. When the number of files is large, and the number of files stored on the host is large, the expected return on 1 re-roll will be small, thus security in this manner will be cheap for large networks. After the host has completed the storage proofs, they must make another down payment and roll a new random seed.

One final condition is noted on the storage proofs. A host either has the whole set of files, or it doesn't, and a host cannot report that a single file has corrupted. If the host claims a single file has corrupted, the host is considered to have failed the storage proof and must pay the same penalty as if it was being non-cooperative.

We observe 3 shortcomings with this approach:

- Files need to get new seeds after their original payment expires. This causes turbulence.
- Hosts need to get new seeds after their original payment expires. This causes turbulence.
- The number of hashes needed to determine the state of the network is the number of hosts times the number of files. This will not scale.

A better scheme should be looked for.

3 Addressing Scheme

There is a 80 bit addressing space which helps to determine which hosts will store which files, containing 2^{80} slots for files. Collisions are permissable - multiple files or hosts can be in the same slot. This address space is only partially in use at a time, depending on the number of hosts in the network. For each host, 2^{24} slots are open on the network. Each host has a 'spanning area' of 2^{30} slots, cenetered around the hosts 'home' slot. This means that there is an overlap of 2^6 - each slot on the network will probabilistically be covered by 2^6 hosts. A hosts home slot will never change. This is a feature of the network because if a host's home slot were to change, the files which it stores would also change entirely, resulting in a large bandwidth expense to the network, which is undesirable.

When a file is added to the network, it is assigned a random seed 256 bits in length. The seed is assigend from an external source of entropy - it is not the hash of the file nor has any relationship to the file. This seed is truncated to produce a random slot in the 80 bit address space. If the slot assigned to the file is outside of the active space, the hash of the seed is taken and used to produce a new slot between the 0th slot and the initial slot. This process is repeated until the file ends in a slot within the active address space. **Example:** We have a full address space of 512 slots with an active space of 100 slots. A new file is randomly assigned a seed that is used to produce a random slot between the 0th slot The seed is hashed to prodouce a new value, and this value is used to choose a new slot between index 0 and index 386 (inclusive). The new slot is 105, still outside of the address space. This process is repeated until a slot value appears that is between 0 and 100 inclusive. The expected number of hashes is approximately equivalent to the log of the number of slots that are inactive minus the log of the slots that are active. Because the process itself uses the hashes of a trusted random seed, and stops preciely when a value is first available in active space, which space of the active spaces gets chosen is completely random. Therefore, adding a file to the network will always result in the file being placed in a random slot.

When a host is added to the network, the active addresses are expanded by 2^{24} slots. The host is then given a random seed derived from an external source of entropy. This seed is used to determine a permanent slot for the host to reside.

As a host is added to the network, new address space is opened up. We wish to put files in this new address space at random, taken from the other slots on the network. This is achieved by looking at the iterated hashes used to find a file's current location. If a slot in a previous iteration opens up, that slot is given priority for the file, and so the file is moved from its current address to the now-available space.

When a host leaves the network, every file that the host was storing is given a completely new seed.

4 Byzantine Attacks

We do not care what goals byzantine attackers have, so long as the following properties are maintained:

- Every file is stored on a host.
- The number of files each host is storing follows a normal distribution within a provable bound.
- A host controlling less than 50% of files cannot manipulate the network such that it will probabilistically be able to store greater than 50% of files on itself.
- A malicious body of hosts less than 50% of the network in size cannot manipulate the network such that a file being uploaded will have greater than 50% chance of landing on a member of the malicicous body.