Secure Distributed File System

Drew Barlow :: drew.barlow@umbc.edu

Zahid Ahsan :: gz35697@umbc.edu

Kyle Wright :: kwright4@umbc.edu

Nitin Buddhana :: nitinb2@umbc.edu

Deval Rane :: devalr1@umbc.edu

Table of Contents

Problem Statement

It is no secret that socket-to-socket communication was not designed to be secure from the start. As a result, entire fields of cybersecurity, such as network security, are dedicated to it. Designing a distributed file system is no doubt a difficult task, but it is one that is vital to any given network of substance. Furthermore, this file system must be secure in all cases. That is, it must make use of encryption. Depending on the methods chosen, a properly implemented distributed file system can meet most attributes of the CIA-NA acronym with just encryption– those being confidentiality, integrity, non-repudiation, and authenticity.

Requirements

This project requires the implementation of a Peer-to-Peer (P2P) style of network. As a result, one must be able to not only download files to their machine, but also store and serve them to others. The implementation must also be made available on GitHub, and can be found here.

Let us lay out what we already know about this system:

1. The system shall be of a P2P architecture. That is,
    a. Untrusted machines shall have files stored on them, and
    b. The system shall deal with concurrent reads and writes.
2. The system shall be secure. That is,
    a. Key revocation shall be implemented,
    b. A malicious file server shall be detected,
    c. Suspicious behavior shall be logged, and
    d. Data shall be encrypted at rest and in transit.
3. Users shall be able to perform the basic create, read, update, and delete (CRUD) operations. Included in this is permission setting of files and directories.
4. Users shall see the latest version of any given file,

Addressing Requirements

Out of the gate, we know the following: the Transmission Control Protocol (TCP) would be useful for guaranteeing that data reaches its intended destination. Furthermore, because of the fact that multiple clients may connect to any machine at a given time, the server must have some concurrency or scheduling aspect.

Python 3 has a wide range of libraries that are easily accessible and well-documented via both its built-ins and its **`pip`** system. For networking applications, this functionality really shines.

Let us now discuss how we shall address the requirements as mentioned in section II.

Peer-to-Peer (P2P) Architecture

As previously mentioned, concurrency must be used here, and TCP would cut out the hassle of self-managing guaranteed transit. Python 3's **`asyncio`** library provides us with an interface to the low-level socket library. As a result of using this library, any functions that perform read or write actions shall be coroutines, and therefore are scheduled in nature[1]. Furthermore, the use of asynchronous functions means using a locking mechanism is perfectly suited to handling concurrent reads and writes. These locks are provided to us as synchronization primitives.

When machines join and leave the network, other machines must be made aware of this. Furthermore, this notification must be atomic– that is, it reaches all machines in the network or is canceled. Therefore, some multicast algorithm must also be present.

Security Details

This is where the bulk of time shall be spent. As mentioned in the Problem Statement, we shall implement the file system in such a way that confidentiality, integrity, non-repudiation, and authenticity is handled by cryptographic algorithms. The following description is a very high-level overview of the algorithms we shall be using and why we shall be using them. The **`PyCryptodome`** library provides us with many of the algorithms we shall make use of– those being AES-256 (**`Crypto.Cipher.AES`**) and SHA256 (**`Crypto.Hash.SHA256`**). Furthermore, we shall use the **`ECIES`** library for public and private key encryption. The keys used are derived from Elliptic-Curve Cryptography (ECC).

Beginning with ECC, public keys shall be exchanged upon an established connection. It is with these public keys that a freshly generated AES-256 symmetric key is exchanged (confidentiality). After this occurs, an acknowledgment is sent. This acknowledgement is encrypted under the symmetric key established, and is bundled with a signed hash digest of that ciphertext (non-repudiation, integrity). This bundle is then encrypted under the recipient's public key and is sent to the recipient. Upon reception, the recipient can verify that the hash matches the ciphertext. They can then decrypt the ciphertext with the sender's public key (authenticity), thus returning the acknowledgement in plaintext. Thus, data is protected in transit. These same key pairs can be used for assigning permissions to

[1] Writer's note: Writing error handling for the C-style sockets is a punishment worse than death. Handling thread synchronization also gets real messy real fast. God bless the Python developers for doing the hard work for us. -Drew

files and directories (i.e., key revocation). At rest, all data shall be encrypted under the peer's public key. If the peer decides to generate a new key pair, then all data on the file system shall be re-encrypted under this new public key. Thus, data is protected at rest.

Lastly, all interactions shall be logged locally. Python's `logging` library has a variety of tools that make this a simple process. If suspicious behavior, such as a potential replay attack or attempted unauthorized file access, is detected then it shall be flagged and reported in the logs.

CRUD Operations

To handle basic CRUD operations, we intend to implement a simple "shell-like" state with a basic querying language. The peer shall be able to enter a command and have it multicast to all peers in the network, which shall respond accordingly. Each portion of the CRUD acronym shall have an accompanying command. Because of this querying language, users shall be able to also modify not only the contents of files and directories, but also the permissions attached to them.

Furthermore, reading and writing shall be made asynchronous via the usage of the `aiofiles` library.

Versioning

It is vital that all machines have proper clock synchronization. Without this, a peer may have incorrect results when querying for updated files. In order to tackle this problem, we intend to make use of <some-clock-synchronization-algorithm>.

I was initially thinking about using the Berkeley algorithm, but taking the average state of all machine's clocks can be problematic between sessions. Best to make use of some other clock algorithm that is maybe already implemented. -Drew

Project Architecture and Documentation

Documentation here (update as the project progresses).

UML Diagram

Placeholder

Limitations and Future Work

This implementation does not do anything to take availability into account. If a machine stops responding, it is simply dropped from the network.