COMP90015 Distributed Systems

Project 2 Report

"HACKERS"

Emmanuel Macario, 831659, macarioe@student.unimelb.edu.au
Michelle Keane, 832948, keanem@student.unimelb.edu.au
Maacah Martins, 759486, mmartins@student.unimelb.edu.au
Walid Ayoub, 866377, wayoub@student.unimelb.edu.au

1. Introduction

The aim of this project is to make adjustments to the BitBox implementation that has been completed in Project 1 in order to make it secure and add UDP protocol functionality. This firstly involves building a BitBox Client that can securely communicate with the BitBox peer, using public/private key cryptography along with AES encryption using a Secret Key. Secondly, the BitBox server needs to be modified to have the ability to use UDP instead of TCP between peers, and also be able to switch between these two protocol via the *configuration.properities* file. Furthermore, we are required to propose a way in which encryption techniques could be used to provide a file permission scheme in which read/write permissions are required for users to access and modify files.

This report is in three parts: firstly, in section 2, the design question regarding the file permission scheme will be addressed; secondly, in section 3, the UDP implementation will be explained; thirdly, in section 4, we have listed the steps used to create public and private keys for the BitBox Client implementation.

2. File Permission Scheme

Traditional server-based file systems are designed with the assumption that the central server can be trusted by the other nodes in the network. In a peer-to-peer system, this assumption no longer holds true as some of the participating nodes might be compromised [1]. Creating a secure file system with a file permission scheme on top of untrusted nodes provides new challenges. As there are no trusted servers in a peer-to-peer system, it is not sufficient to encrypt only the client connections since all of the data stored on one peer will be available to any other peer in the system. All data stored on a peer will have to be encrypted [2]. Ideally, a file system would be able to conceal files from every node in the system and give access to only the users and groups that have permission to either read or write.

2.1 Design Overview

The design presented in the section is based off off some of the ideas presented by Adya et al. [3] in implementing *Farsite*, a serverless distributed file system. The design requires some administrative component as described in section 2.2. This component could be created by enhancing the role of the client that has been implemented in this current project. The first few subsections will summarise the basic elements and architecture required for the design to work, which will require some modifications in the current BitBox implementation. Subsequently, the protocol that is followed by clients to read and write to files will be explained.

2.2 The Client

Every peer in the system can have three roles: a client, a member of a directory group (see section 2.5) and a file host (who stores encrypted replicas of files).

2.3 The Administrator

The role of central administration has been modified in this system to involve a single client machine that will authenticate new users and generate certificates (acting as the Certificate Authority) as they join the system. The administrator also creates the namespace roots and assigns peers to different directory groups upon joining.

2.4 Certificates

Trust will be managed using public-key-cryptographic certificates. There will be two types of certificates to be administered by the administrator. A *namespace certificate* associates the directory namespace to the directory group members, as mentioned above. A *user certificate* associates a user with her personal public key.

2.5 Directory Group

A directory group is a set of peers that have been designated to manage file information. These groups maintain an access control list of public keys of all authorized writers. Upon joining the system, BitBox peers may be assigned to one or more of these groups. Each peer in the group is responsible for managing the namespace root. These machines should form a Byzantine-fault-tolerant group [4] so that specific (protected) machines will not have to be chosen to manage each directory.

Due to the current structure of the network, when a client is looking to locate a particular file the client must successively contact directory groups until the group responsible for managing that file is found. This may pose scalability issues given there is no guarantee the file will be located.

2.6 File and directory privacy

Whenever a client creates a new file it will randomly generate a symmetric *file key.* The client then computes a one-way hash of all of the blocks of the file. This hash is used as a key to encrypt the block. The file key is then used to encrypt the hashes of the blocks rather than the file blocks directly (in order to be able to identify and coalesce identical files). The file key is then encrypted using the public keys of all authorized readers of the file. The file key encryptions are stored with the file. A user with a corresponding private key will be able to decrypt the file key and thus the file.

In order to prevent directory group members from viewing the contents of the files or directories that they are managing, the encryption takes place before these files are sent to the group and then sent to the directory metadata.

2.7 File and directory integrity

A Merkle hash tree [5] is computed over the file blocks to ensure the integrity of the data. A copy of the tree is stored with the file and another copy of the root hash in the directory group's metadata.

2.8 Leasing

If a client wishes to access a file, the directory group that manages that file will issue a lease which grants them access for a certain amount of time (this leasing characteristic assists with consistency in the case of node failure). Because the files are cached locally, a client can perform the operations locally and then push updates back to the directory group who will then authorise it.

2.9. Enforcing Permissions

Below is a list of steps that will be undertaken when a client is requesting a file from another client that is known to be part of the file's respective directory group, as shown in figure 1.

- A message is sent from a client who wishes to read a file to the directory group that manages that files metadata
- The directory group can prove that they have the authority of this file by providing the corresponding namespace certificates (up to the root namespace certificate that is signed by the administrator).
- The directory group will reply with these namespace certificates, a lease on the file, a one-way hash of the file contents, and a list of the file hosts that have encrypted replicas of the file.
- The client will then retrieve one of the replicas from a file host and validates its contents using the one-way hash.
- If this user has read access to the file, they can use their private key to decrypt the file.
- If the client updates the file, it will then send an updated hash of the file back to the directory group.

- The directory group will then verify whether or not this user has write access to the file.
- If so, the directory group will then instruct the file hosts to retrieve copies of the new data from the client.

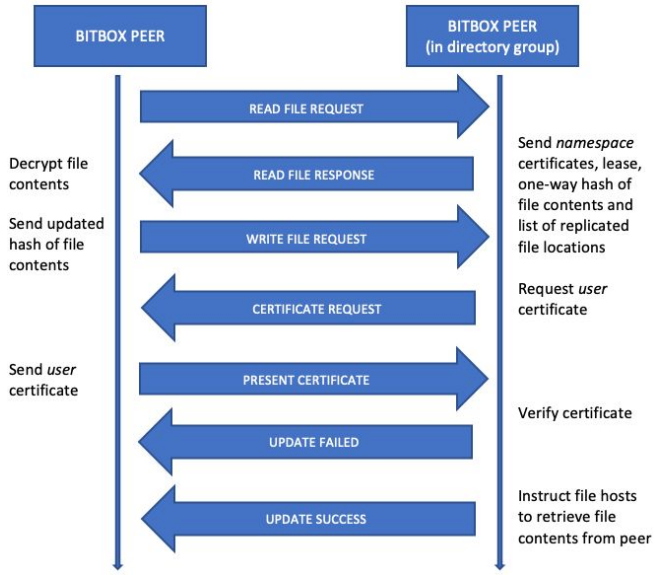

Figure 1: File Request Protocol

2.10 Limitations

This design does require some modifications to the current BitBox implementation. This is a limitation because ideally this type of permission scheme could be implemented using the current architecture and structure. Concurrency and consistency procedures will also have to be amended to work with this scheme.

3. UDP Implementation

The addition of UDP functionality to this system involved implementing conditional statements so that the program could switch between TCP and UDP protocols according to the required settings stated in the configuration.properities file. This required modification to multiple methods in order to carry this additional input parameter of which protocol mode the peer-to-peer system should be implementing at the time.

Having a connectionless protocol removes the need for a handshake, but increases the risk of data being lost due to not authorising peers and checking the connection [6]. Although it is faster than TCP, not scrutinising the peers and the connection leads to riskier file transfer.

However, one of the benefits of UDP in this scenario is that no client data is saved, which prevents identification of the client being available in the future [6]. This increases the security of the system.

As UDP is unreliable, unlike TCP, in order to check whether the packets and commands are reaching their destination, parameters for tracking the number of times the packet has been resent are required, as well as a timeout feature. These are used to check whether the peer in question is still capable of receiving the packets, or if the sending peer should give up and cease all communications with it. As UDP has no form of acknowledgment of received or lost data, these parameters offer some failure handling in regards to irregular peers.

Since UDP does not try to recover lost data, this presents a major issue in the situation of a shared folder system, as synchronicity is at risk and the system has no real way of handling failures as a result of lost data.

UDP should be used in the case where speed is the top concern, whereas TCP is the better-suited protocol when looking to provide a secure and reliable service. In this implementation, all security and peer screening measures are handled in TCP protocol, whereas the communication between the peers is implemented in UDP (although can be switched to TCP if required) This creates a service with all the security of TCP and the speed of UDP, forming a file sharing system capable of high speed and relatively safe data transfer.

4. Crypto Class Explanation

Private Key Creation:

1. Remove headers and new lines
2. Decode base64
3. Decode to PKCS8 spec
4. Modify to PKCS1 spec

5. Create java Private Key Object

Public Key Creation:

1. Remove "ssh-rsa" and identity
2. Decode Base64
3. Get Modulus and Exponent
4. Generate Java Public Key Object

5. References

[1] Amnefelt, Mattias, and Johanna Svenningsson. "Keso-A scalable, reliable and secure read/write peer-to-peer file system." *Degree Project Report* (2004).

[2] Amann, Bernhard, and Thomas Fuhrmann. "Cryptographically Enforced Permissions for Fully Decentralized File Systems." In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, IEEE, 2010, pp. 1-10.

[3] Adya, Atul, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment." *ACM SIGOPS Operating Systems Review* 36, no. SI (2002): 1-14.

[4] Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." In *OSDI*, vol. 99, 1999, pp. 173-186.

[5] Merkle, Ralph C. "A digital signature based on a conventional encryption function." In *Conference on the theory and application of cryptographic techniques*, Springer, Berlin, Heidelberg, 1987, pp. 369-378.

[6] Chaudhari , A. “14 Difference Between TCP and UDP Protocol Explained in Detail” https://www.csestack.org/difference-tcp-udp-protocol/