Highly Dependable Location Tracker

Afonso Ribeiro - 86752
Beatriz Martins - 89498
Pedro Lamego - 89526

Stage 2

1 Problem

The aim of this project is to implement a location tracker system that works with time being split into epochs. That system has the users periodically sending their location to a server, accompanied with a number of proofs guaranteed by the users neighbours to validate that the user is in the position it says. In this stage of the project, the system uses multiple servers and it may be subject to Byzantine faults affecting those server processes. Users have some features available: they can consult the server in order to know where they were in each epoch and also, to obtain all the proofs that it provided to other users in a range of epochs. Besides that, there is a special user, the ha client, that can consult the location of all users across any epoch.

2 Proposed Solution

2.1 Design choices

The project was implemented using Rust. For the communication technology we opted to use a crate called tonic which is a gRPC implementation for Rust and for the cryptographic functions we used a crate called sodiumoxide.

The cryptographic part of the system requires the generation of the keys. That is done when the program starts. In contrast to the last stage, we now encrypt the private keys with a password and save them to a file. To simulate the user inputing his password, it is saved in plaintext in a file.

The clients, servers and the ha client have asymmetric signature keys. Besides that the server has an asymmetric key for encryption.

For the signature keys we use Ed25519 (4), which is a public-key signature system that has fast key generation and a high security level.

For symmetric and assymetric encryption we rely on a cryptography library called (5) NaCl. Two concepts were used, the sealed box, which depends on the public key of the receiver, and the secret box, which uses a symmetric key.

The sealed box is a particular combination of Curve25519, Salsa20 and Poly1305 and the secret box is a combination of Salsa20 e Poly1305. For reference, Curve25519 is an elliptic curve algorithm used for elliptic curve Diffie-Hellman, Salsa20 provides encryption and Poly1305 is used for one-time authentication.

As mentioned in the definition of the problem users are required to send their location periodically to the server. They do this by sending a number of proofs, which are part of a report in our design. Besides the proofs the report includes the epoch, the location of the user who is sending it and the id that identifies that user. On the other hand, the proof includes the id of the user who requested it, the id and location of the user that signed it and the epoch it relates to.

The requests are always in the same form: proof of work + info + request. Both the info and the request are signed before they are encrypted. The request is encrypted with a symmetric key generated by the sender (secretbox), the info contains the id of the sender, the generated symmetric key and a nonce which is all encrypted by the server public key(sealedbox).

The response is signed by the server and encrypted with the symmetric key that was in the info.

We assume that the bizantine clients cannot talk to each other unless they are in close proximity.

2.2 Possible threats and protection mechanisms

There can be up to f byzantine users in the entire system and there is a certain limit imposed (f' < f) on the number of byzantine users that can be (or appear to be) nearby a correct user. Then, f' is such that all clients have at least f' + 1. With that in mind, the protection mechanism implemented was: each client tries to obtain 2f' + 1 proofs and sends a report as long as it gets f' + 1 (which by definition it would). The server only needs to verify f' + 1 correct proofs.

In this stage of the project, the servers can now be byzantine. With that in mind, given N servers, we calculate the max number of fs servers possible to respect N > 3fs. So, to reach a quorum we need to obtain $(N + fs)/2$. For that, we implemented the algorithms: Authenticated Double-Echo Broadcast Algorithm(1)[3.18], Authenticated-Data Byzantine Quorum(2)[4.15], Byzantine Quorum with Listeners(3)[4.18].

To protect the register from byzantine clients we used algorithm 3.18 to ensure that we have a quorum of writes.

To ensure non-repudiation all reports are signed by the user who created them, which means that a user is not able to repudiate any previously submitted location reports. The communication among users is not confidential since it does not involve any security mechanism except for the signature mentioned before.

To ensure confidentiality and to protect against man-in-the-middle attacks in the messages exchanged between users and the server the requests are encrypted with a generated symmetric key which is encrypted with the public key of the

server, meaning only he can decode those messages. Then, the server uses that symmetric key on the response. That means that the reports, besides being signed, are also encrypted.

With our implementation a byzantine client can only pretend he is in the neighbourhood where he appears to be. To prevent byzantine clients from submitting too many false reports we created a blacklist. It works in the following way: the server can detect a byzantine client if he submits another report with a different location for the same epoch and if that happens the client is blocked from the system.

To combat Denial of Service attacks we used a mechanism that relies on the Sha256 hash and that is called proof of work. This mechanism aims to reduce the number of requests servers can receive from byzantine clients by imposing a computation cost to trigger the execution of expensive requests. In our case, we opted to use this mechanism for all reads and writes since the benefit that it adds is superior to the cost it inflicts.

2.3 Integrity guarantees

The system should protect against replay attacks. For that, the server has a set of nonces for each of the client requests.

Since it is requested that if the servers crashes data loss or corruption can't happen we use algorithm 3.18, which means that we can guarantee that no data will be loss by a quorum since it keeps that information. Besides that, we use a crate(atomicwrites) which takes advantage of moves being atomic in the same filesystem and that first writes to a temporary file before doing a move.

To ensure synchronization and that the data will not get corrupted we use read write locks. This means that multiple threads can read the data in parallel but an exclusive lock is needed for writing or modifying data.

References

[1] Introduction to Reliable and Secure Distributed Programming, 2nd Edition: C. Cachin, R. Guerraoui, L. Rodrigues 2011 Springer - ISBN: 978-3-642-15259-7, page 118, Algorithm 3.18

[2] Introduction to Reliable and Secure Distributed Programming, 2nd Edition: C. Cachin, R. Guerraoui, L. Rodrigues 2011 Springer - ISBN: 978-3-642-15259-7, page 181, Algorithm 4.15

[3] Introduction to Reliable and Secure Distributed Programming, 2nd Edition: C. Cachin, R. Guerraoui, L. Rodrigues 2011 Springer - ISBN: 978-3-642-15259-7, page 190, Algorithm 4.18

[4] http://ed25519.cr.yp.to/

[5] http://nacl.cr.yp.to/index.html