Instituto Superior Técnico
SEC16/17
MEIC-A

Group 4:
João Santos 67011
Francisco Martins 76061
Paulo Anjos 87822

Stage 2:

The goal of this part of the project is to extend our implementation from stage 1 to tolerate the possibility that the server side of the infrastructure may suffer from a successful attack. To do so, we implemented a (N,N) Atomic Byzantine register.
It is important to notice that the requirements that weren't met in the previous delivery are now made, which are, freshness by using nounces, server response authentication using signatures, the server supporting concurrent instances of clients and the server now properly saves the client's signature of the password so that the client can be sure the password that the servers returns to him is the one sent by himself. The server also stores all the passwords and respective signatures so that in the future the server can prove that the passwords were stored by the respective clients (non repudiation).
We started the 2nd stage by replicating the server without any type of synchronization and checking if the client could successfully connect to all of them. The library we used (REST) is by default multi-threaded, having a thread per request, so we only needed to control the access to shared variables.
Afterwards we "grabbed" the pseudo-code that is in the book to check the implementation of the (1,N) Regular Byzantine and wrote the code inside the server class. The client needs (N+f)/2 replies from the servers to accept the appropriate values, because if it chose only one reply, that same reply could be from a byzantine server.
In the next step we changed the server to implement an atomic register (1,N) Atomic Byzantine. A write back function was included within each read that was made (get function) this write back assures that after a read, the majority of the servers have that same value (atomic).
To reach the final step, we concluded that the server would have to do "all the heavy work". If the client initiates another session with another device, when he attempts to make a write, the servers will first make a read to all so it updates the timestamp for the given client instance (read before write). Each server knows the other server’s Public Key.
All these steps assure that the password returned to the client is the right one, and the password associated with the username and domain that the client tries to save is exactly the same one.

We have a shell script included within our project folder, it runs 4 server instances to tolerate at least one faulty server and two users, where at least one has two device ids. Instructions are in the Readme.txt file.

Dependability guarantees provided by our system:
Availability-Our system provides availability by having multiple instances of the server running independently from each other. Regarding this, the system needs to have (N+F)/2 servers running.
Reliability-Is provided by the implementation of N-N Atomic byzantine register algorithm.
Safety- Is assured because when we have at least (N+F)/2 servers, the algorithm assures safety, when the system does not have the necessary amount of servers, it just becomes unavailable preventing the users from using it and have inconsistent states in the servers.
Integrity-Is provided not only by the implemented algorithms but also by digital signatures.
Maintainability-Our system is relatively small but due to some simplifications, like hard coded file names, it can be tricky to change and maintain. In despite of it maintainability was not the main goal of the project.

Assumptions:
- Clients are not byzantine, because he can do whatever he wants, it wouldn’t make sense for him to be byzantine
- Byzantine servers do not change the ports, because we use the ports to id files, keystores etc..., because the program was tested with all the servers in the same host.