

Project report: Dependable password manager

Highly Dependable Systems

# Dependable password manager

## Improvements from the 1st phase of the project

The *Diffie-Hellman* algorithm was discarded and instead we opted to share a session key from the server to the client, upon the client connection, ciphered with the client’s public key and signed with the server’s private key.

We manage to achieve password confidentiality by storing the password (ciphered by the client) in a different file, in byte format. Now in the “triplet” storage file, beyond containing the username and domain information, it also contains an index pointing to the position of the password in the byte file.

The nonces solution was also substituted to a simpler and cleaner one. Now timestamps are used to guarantee the freshness of the messages. Basically, the client when sending a message sends its current timestamp and the server checks if it is more recent. The same process occurs in the client side.

In order to support multiple devices per client a unique id was associated. This implementation allows us to treat the clients with the same public keys as different instances and gives us a reference to them.

Amongst all these alterations on the 1st solution, a new file was included in the files’ set. This contains a registration of all the digital signatures from the client to the server, giving a more robust look to the non-repudiation system.

## Broadcast Abstraction

The *Best Effort Broadcast* was implemented using *Perfect Point-to-Point Links* assured by the TCP channel that is used by Java RMI. To broadcast an event, a *read* or a *write*, we do not broadcast the event to ourselves, instead we already account for our response, example, the initial value of *ack* is 1.

## Asynchrony in our project

We have 2 instances of asynchronous calls in our final solution. The first one being when a client registers to a server and the server broadcast asynchronously the register to all other replicas, afterwards collection all the necessary replies. The other instance happens when a server broadcasts a read operation to the register.

## *N:N* atomic register

Since we are working in a *N:N* atomic register environment, multiple writes can be done at the same time, in order to break possible ties we must have an additional conditioner. This comes in the form of a ranking variable that specifies the priority of a given write. In or case this variable is the writer’s port, and we consider that the lowest port always has the highest priority.

## Byzantine tolerance implementation

Regarding byzantine fault tolerance, our implementation achieves a set of warranties that allow the system, consisting of *N* replicas, to keep working under the limit of *f* faults. Such properties are complemented by the implementation of a byzantine toleration algorithm in the *N:N* atomic register, which consists in signing the critical information that is saved in the file and later checking it. Such technique allows the system to be ready to resist attacks regarding the alteration of the passwords/storage file information and failing of some of the replicas (crashing). This behavior is observable in the implemented four tests, where number one, two and four, test the undue alterations of timestamps, ranks and the password itself, respectively. On the other hand, number three tests the functioning of the system after some registered replicas crash, simulating a loss of connection or some other reason that might break the link between replicas.

## Faults of the System

The system presents some faulty features. These are related to possible byzantine faults. The solution doesn’t implement an answer for a byzantine writer, that might contain adjusted (in the syntax level) information, but inconsistent, such as a timestamp that looks regular but it represents a date further away from the current one; or a rank that is far low from the one that server should have.