Instituto Superior Técnico

TÉCNICO LISBOA

SEC Report II

2016/2017

Group 11

João Varandas - 77974

Tiago Rosado - 77958

Rui Pacheco - 77989

Introduction

At this stage, we started by solving the issues reported on the first delivery, then we followed the implementation recommendations, that focused on:

1. Starting multiple Replicas without synchronization mechanisms and building a loop on the client side scanning all online servers.

2. Implementation of Byzantine Regular Register that is referred on the section 4.7 of the course book considering the communication of one client to multiple servers.

3. Transform Byzantine Regular Register (1xN) to Byzantine Atomic Register (1xN) model that is presented on section 4.8 of the course book.

4. Transform the 1xN Atomic Register to NxN Atomic Register

5. Implement tests to verify the correctness of project execution.

Improvements from previous delivery

1. Added salt to the hash content that is stored on the server, which is a secret only known by the client (in our case we add the client keystore password) to prevent guessing attacks with rainbow tables.

2. Added security to server-client direction of communication by adding a verification on TransactionID to prevent replay-attacks from a man-in-the-middle attacker.

3. Added signature verification to server-client direction of communication to guarantee authentication and integrity of the data that is sent.

4. Added a log register on server operations to allow the monitoring of operations already done.

Implementation of fault tolerance

Byzantine Regular Register implementation (1xN):

To implement Byzantine Regular Register we started by making a model based on multiple requests from one client to multiple servers. To implement this model we've implemented a middleware that manages the communication between the interface made available to the client and the servers; this middleware was implemented as a multi-threaded class responsible for all the communications to the server, assigning one thread to each server, in order to prevent blocking among the multiple requests that are sent to each server.

Byzantine Atomic Register implementation (1xN):

To implement the atomicity on the regular register, we changed the behaviour of read operations on the client side; this means that we've built some mechanisms over the algorithm that is described above allowing all the servers to maintain correct and updated information.

These mechanisms are based on requesting for each read operation (in our case the read operation means executing the *retrieve_password* in order to read a certain password from server(s)) an update on all the servers with the *chosen correct response* (the most updated password).

This means that we receive a response from multiple servers and choose the response with the highest value for logical timestamp, then we broadcast this value to every server that is online and communicating with client.

Byzantine Atomic Register implementation (NxN):

To implement NxN capabilities on the server, meaning N clients concurrently requesting services from N servers, we've had to solve concurrency issues generated by multiple simultaneous accesses of the same client and of different clients.

As such, three main issues arose:

1. Client A has two instances that concurrently update the same password.
2. Client A and Client B make a simultaneous request to the servers.
3. Client A has two instances A1 and A2, A1 updates a password using *save_password* as A2 requests a *retrieve_password* from the server.
4. Server #i has a byzantine fault.

To solve issue #1, we've included an aforementioned Logical Timestamp (LogTs) that must be previously obtained by the client using *getLatestTimestamp* method representing the latest (the most updated) Timestamp associated with the latest write of all servers. The *save_password* request that has the most recent LogTs will be chosen and the password will be updated; in the case of a tie, a Physical Timestamp (PhyTs) also included in each message will be used as a tiebreaker, in that case, the password with the latest PhyTs will be chosen and updated.

To solve issue #2, the hashmaps used to store the tuple containing the login information now support concurrency, this is, we change the use of Java HashMap[1] to Java ConcurrentHashMap[2] allowing multiple inserts and reads at the same time.

To solve issue #3, we've considered different case scenarios:

- ➔ if the write operation of A1 has already succeeded in at least on server, then A2 will get the latest value written, meaning it will get what A1 wrote;
- ➔ if the write operation of A1 has not yet reached any server, then A2 will get the latest value written, meaning it won't get what A1 wrote but what was previously written.

To solve issue #4, we've had to make a resilient system that maintains correctability in the face of byzantine faults. To attain it, whenever a client makes a request, the middleware responsible for the server communication only has to wait for 2f+1 answers where f is the number of faults the system is designed to support. This quorum can be used to infer the correct answer while maintaining fault tolerance.

Automatization of System Deployment

In order to hasten the tedious process of designing a system capable of sustaining f byzantine faults, we built a bash script that allows us to receive the number of faults as input and to compute the necessary number of servers that support these faults, automatically manage the keystores and certificates of the generated servers and automatically run the necessary tests and launching the servers.

This script "*./run.sh f* " takes an argument f which is the number of faults that the system has to support. The number of replicas to be launched is calculated by the script using the ratio $N = 3f+1$, where N is the number of replicas and then the script runs the process that is described on the upper paragraph.

[1] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
[2] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html