Dependable Public Announcement Server

Stage 2 Report

Highly Dependable Systems

Group 16 - Alameda

Rodrigo Lima 83559

João Martinho 86454

Henrique Sousa 94126

1. Corrections to the first stage

1.1 - Overall System Architecture

On the previous delivery the client module kept an hardcoded digest for the client's passwords. In this version that is no longer true, as all passwords are protected in a file under the resources folder.
Since this stage required multiple servers, the new key pairs for those were generated the same way as the previous ones.

1.2 - Keystore

Previously we had one keystore that kept all private keys and associated self-signed public key certificates, but the access to those always used the same password. In this version, we have as many keystores as there are key pairs and each keystore is protected by its unique password.

1.3 - Persistence

Regarding persistence everything remains the same except for the obvious, with several servers, there is the need to maintain persistence files for each one.

1.4 - Secure Communication Protocol

The past version did not guarantee integrity, non-repudiation, replay/drop prevention, message stealing and surreptitious forwarding for the read and readGeneral requests (since the client only sent the hash, without signing it, and that same hash did not contain any information regarding sequence numbers nor sender/destination). The new version guarantees all the above requirements, for every request.

1.5 - Client-server synchrony

The register operation, in addition to preparing the server for new clients, now also returns the sequence number and write timestamp (new variable for registers abstraction) for a returning client, to ensure a client can resume its operation with the server.

2. Changes to accommodate multiple servers

2.1 - Sequence numbers

To prevent replay attacks, we keep using sequence numbers, as we did in the 1st stage. However now we need to keep one sequence number per server on the client side, as synchrony between servers may not be guaranteed.

3. Authenticated Perfect links

The register abstractions mentioned below require authentication of every message we send, to prevent impersonation (and no creation), and so we use the abstraction of Authenticated Perfect Links, with the Authenticate and Filter algorithm, described in Section 2.4.6 of the course book.
Note: the transport in our project is done using HTTP, which uses TCP, therefore stubbornness and no duplication is guaranteed by the transport.

4. Byzantine Register abstractions (with correct clients)

4.1 (1,N) Regular Register

To create the (1,N) Regular Register abstraction, we applied the Authenticated Data Byzantine Quorum algorithm (Section 4.7.2 of the course book). Every write and read request, only performable by clients registered in the platform, must wait for a byzantine quorum of valid responses, and only then return to the user.

Note: a valid READ response, in the context of our project, corresponds to all the posts returned being properly signed by the writer of each post.

4.2 (1,N) Atomic Register (Personal Boards)

To create the abstraction of (1,N) Atomic Register, we applied the Read-Impose Write-Majority algorithm (described in section 4.3.4) using authentication. Because this algorithm is for byzantine servers, the quorum we wait for at the client side is a byzantine quorum. Also, the register value in the context of our project is a list. Therefore, to properly and correctly implement the write-back operation, the servers return the full list of announcements it has when a read is performed. This creates a overhead in communication, but security and correctness are not violated, which we deemed to be more important than performance in the context of our project.

4.3 (N,N) Regular Register (General Board)

For this transformation, we used the (1,N) Regular Register abstraction, and perform a "read before write" operation before every postGeneral request. This is done by, before the WRITE request is performed, collecting from a byzantine quorum of servers the latest announcement they know (a read operation) and obtaining the highest wts from those. This allows the writer to know which timestamp it should give to the servers when performing its WRITE operation.

Note: we want to ensure writes are not lost, even if they are delayed. This is important since our register value is a list. As such, the server remembers the highest ts it returns to a client when a read is performed, so it can expect a WRITE with that ts + 1 from that client in the future, even if the write comes much later (due to delays or other issues).

5. Byzantine clients

The algorithms presented above always assume the writers to be correct. However, there may be cases where clients are byzantine. In this section we present some suggestions to mitigate problems related to byzantine clients.

5.1 Clients sending different messages to different servers

A byzantine client, with proper registration rights on the server, may send different messages to different servers, but these messages are still properly signed and therefore "valid". This is undesirable, as read operations from then on are never guaranteed to return values that correspond to the specification. To mitigate this, using a Byzantine Reliable Broadcast would prevent such cases. That is, if a message is delivered by a correct process, all other correct processes are sure to deliver it as well. The algorithm proposed in the book is Authenticated Double-Echo Broadcast (section 3.11.2), where the broadcast is done by phases:

- 1st - SEND: client sends the message it wants to deliver to all servers;

- 2nd - ECHO: triggered from receiving a SEND message, each server broadcasts the request it received;
- 3rd - READY: triggered from receiving (N+f)/2 ECHOS for a message or f + 1 READYs for a message, each server broadcasts its commitment to deliver a specific request.
- 4th - Delivery and response: triggered from receiving 2f + 1 READYs for a message, each server delivers the message and responds to the client that issued the request.

In the example execution given in Figure 2, we see that no server meets the requirements for the 3rd phase, therefore no server even sends a READY message and therefore no server delivers a message in that instance of the broadcast.

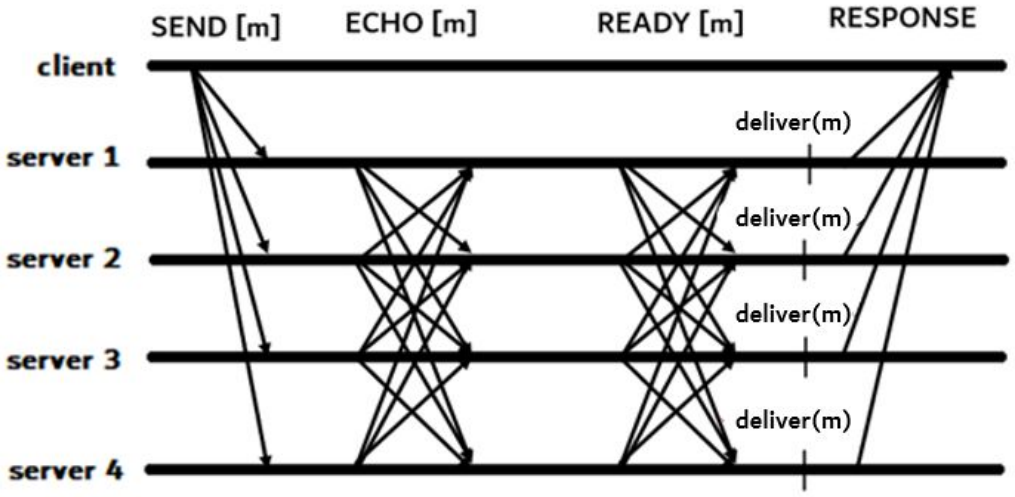

Figure 1: normal execution of the byzantine reliable broadcast (for f = 1)

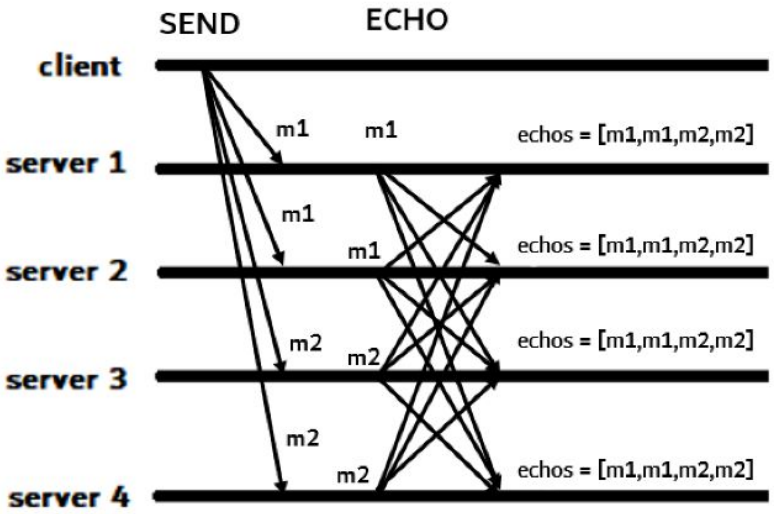

Figure 2: execution of the byzantine reliable broadcast where client is byzantine (for f = 1)

References

1. Java WebServices (JAX-WS)
2. Java Development Kit 8
3. Java Crypto API
4. Apache-Maven 3.6.3