Dependable Public Announcement Server

Joana Coutinho 87666 Rafael Soares 87675 Pedro Rodrigues 87696

MEIC-T T5, May 13th 2020

1 System Description

1.1 System Architecture

Our system was designed using:

- Programming Language - Java 13;
- Build Automation - Maven;
- Security Library - JavaCrypto;
- Distributed Remote Call System - gRPC;

We decided to implement the project using gRPC due to its simplicity and built-in multi-threading for handling multiple clients.

We have implemented three modules: Server, Library and Client.

1.1.1 Server

The Server keeps all the logic and persistence of information, being able to handle multiple clients at the same time;

1.1.2 Library

The Library holds all the client logic for communication with the server and handles all the security and dependability aspects for the Client.

1.1.3 Client

The Client implements upon the Library to choose how to make an interface for clients to use.

For the scope of this project, we are only interested in the aspects of communication and security between the Server and Library, which we will be focusing on the following sections.

1.2 Assumptions

For this project, we have the following assumptions:

- Assumption 1: A Client only makes one request at a time;
- Assumption 2: A Network Attacker can Drop, Reject, Manipulate and Duplicate messages
- Assumption 3: A correct Client never makes mistakes (i.e posts with non-existing announcements, Posting without being registered, etc)
- Assumption 4: A correct Client will have a correct Library

1.3 Library Implementation Decisions

In order to give high availability and security against Byzantine attackers/entities, we have implemented various forms of shared memory algorithms, specifically for the Private Board we implemented (1,N) Atomic Register and for the General Board we implemented (N, N) Regular Register. We implemented these algorithms based on the required abstraction layers shown on the following picture:

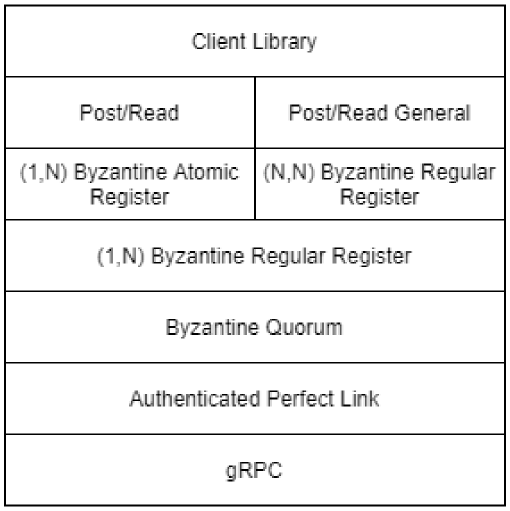
Client Library
Post/Read
Post/Read General
(1,N) Byzantine Atomic Register
(N,N) Byzantine Regular Register
(1,N) Byzantine Regular Register
Byzantine Quorum
Authenticated Perfect Link
gRPC

Figure 1: Abstraction layers for Library side

We will go up the abstraction stack in order to explain each security and availability guarantees that it gives us for each side.

1.3.1 gRPC

Using the FutureStub implementation available from gRPC we are able to use fully asynchronous communication between Client and Server while also giving us the option to add callbacks to the method calls, which will prove us useful for implementing Authenticated Perfect Links.

1.3.2 Authenticated Perfect Links (APL)

Building upon gRPC, we implemented APL to assure Authenticity, Integrity and Freshness of communication between Client and Server. The Links receives the type of request and calls the corresponding method over the gRPC stub.
For each of the abilities we assumed an attacker had, we have the following security counter-measures:

- **Dropping/Rejecting of messages**:
  Since gRPC uses TCP, we do not need to worry about attackers dropping packets, as the transport protocol will re-send them. By definition, APLs are built upon Stubborn Links, which will send infinite messages, assuring this way that eventually the message arrives to the server. We relax this condition by putting a timeout of 40 seconds on the Links in order to save computational power.

- **Manipulation of messages**:
  To prevent message manipulation by an attacker, we have decided to use digital signatures signed with the Client private key using SHA256 with RSA. This way we assure:
  - Non-repudiation of the message;
  - Integrity of the message.

  We also use this digital signature to make the Clients accountable for their posts, as all arguments of an announcement are signed together in the message. This way, all Announcements and Acknowledges sent from the Server can be verified if they have their correct Signature and Freshness. In a real case we would use a protocol for symmetric key generation, like Diffie-Hellman, but for this project we relaxed this and use asymmetric encryption.

- **Duplication of messages**:
  To avoid the duplication of messages, we use logical clocks in the form of sequence numbers to avoid replay attacks. We take advantage of the fact that Regular Register uses logical clocks to use them as freshness.

  For the Client Register operation, we do not implement freshness, as it is an operation only done once and the overhead of checking if a client is already registered is very low.

Besides all this, the APL also verifies for Read requests if the announcements are from the requested type of board (Private or General board) and from the requested Client in the case of reads from Private Boards.

Since we have Assumption 4, we know that any exception thrown by the Server or by failed security checks on the Library side (i.e Integrity/Freshness) are either due to:

- Byzantine Server
- Network Attacker

Since we cannot distinguish between the two of them, we will always resend the request in case of exception, as it could have been a momentary attack on the network.

1.3.3 Byzantine Quorum (BQ)

Built upon the APLs, we made a layer to obtain a Quorum of responses. It was made general to work with any kind of request and to wait for a defined number of responses. Since we wish to obtain a Byzantine Quorum, we wait for $\frac{N+F}{2}$ answers, with N being the total number of Servers and F the number of faults we wish to tolerate.

1.3.4 (1,N) Byzantine Regular Register (BRR)

Built upon BQ, we implement our base algorithm (1,N) Byzantine Regular Register in order to replicate the boards between different replicas. To adapt this protocol to our project, we consider each slot of the Private and General Board as individual registers. Our Write will be the Post/PostGeneral methods and our Read will be the Read/ReadGeneral methods.

Since a Read operation may want to return the latest N messages Written, each Server keeps the list of announcements ordered by Write Timestamp in order to be trivial to return the latest N announcements.

When the BRR receives these values from the BQ, it searches the responses for the M unique messages with highest Write timestamps and returns these to the Client.

Due to the underlying BQ and APLs, we guarantee that a Byzantine server cannot transmit false Announcements to legitimate Clients.

1.3.5 (1,N) Byzantine Atomic Register

On top of the (1,N) BRR, we implement (1,N) Byzantine Atomic Register to make sure that once a Client reads the latest written on a Private Board, every subsequent read will also return the latest value.

1.3.6 (N,N) Byzantine Regular Register

We use this algorithm for the General Board as many Clients may be writing in this board at the same time.

In order to deal with collisions of multiple clients writing in the same register, we simply allow both request and order the list of announcements first by Write Timestamp and then by Public Key, as it can work as a Client identifier.

When a Read General occurs, the (N,N) BRR not only picks the highest Write timestamps as it also orders by the Clients Public Keys.

1.4 Server Side

We will now describe the Server side layer stack.

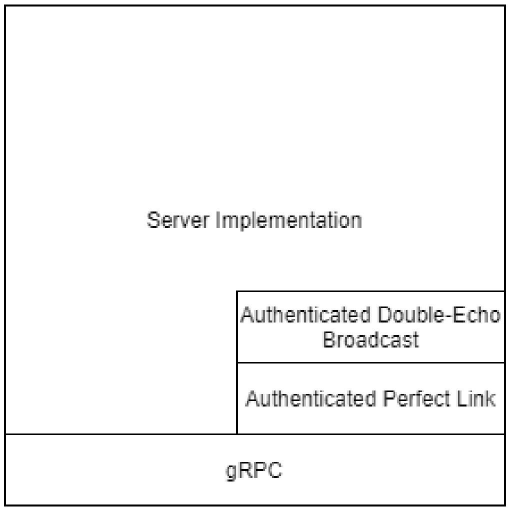
Server Implementation
Authenticated Double-Echo Broadcast
Authenticated Perfect Link
gRPC

Figure 2: Abstraction layers for Library side

Servers must be able to handle two main problem:

- Byzantine Clients, which may send different requests to different Servers
- Byzantine Clients and Servers working together.

To handle both of these cases, we will rely on communication between Servers, which we will discuss in the next sections

1.4.1 gRPC

The same applies from the gRPC section of the Library side.

Apart from it, gRPC has its own built in multi-threaded system for server connections, allowing multiple clients to connect and make server calls at the same time. However, it alone does not handle concurrency issues, so we must include them ourselves. Specifically, we used an Array List with locking mechanisms for the General Board and a Concurrent Hash Map with an Array List for each Client Board. We are only allowed to implement Client Boards this way due to Assumption 1.

If a Read request arrives, the Server does not need to talk with other Servers about their state, since the Byzantine Quorum will handle that in the Client side.

For a Post however, a Server is not sure if the same Post method is being done in all replicas with the same information. To make sure of it, we will use Authenticated Double-Echo Broadcast. We will discuss its purpose in the further sections.

On contrary to the Client side, Integrity and Freshness checks of Client requests are done in the Server Implementation and not in the APL, since we don't need to try and send repeatedly to the Client.

1.4.2 Authenticated Perfect Link (APL)

Much like its client counterpart, the APL will check Integrity, Freshness and Authenticity in communication between Servers, making sure the responses have not been altered and have valid responses.

1.4.3 Authenticated Double-Echo Broadcast (ADEB)

To make sure that a Server is not being tricked by a Byzantine Client by receiving a different request from the other replicas, the Servers makes an ADEB between themselves in order to make sure at least a BQ number of Servers have the same request before delivering it.

In order to avoid coalition between a Byzantine Client and Server, we adapted the $READY$ message from the Broadcast to piggyback a signature of the Announcement being requested. This way, every Announcement will be signed by a BQ of Serves, being a Byzantine Server unable to inject single Announcements from a Byzantine Client.

1.4.4 Persistence of Data

For data persistence, we save each data structure into a file on the server side. In order to avoid corruption from crashes in the middle of writing, we first save the structures in an auxiliary file, making sure the full structure is saved before substituting the old file atomically.

We save both boards, the Server generated IDs of the Announcements and the Read and Write Freshness of the Clients.

1.5 Key Protection

In order to protect the private key of each entity, we keep each key separately in its own password-protected Java Key Store.