Dependable Public Announcement Server

Highly Dependable Systems

Project - Stage 2 - Group 25

84708 - David Coimbra
84774 - Tomás Carrasco
87563 - Ricardo Bandeira

We use the Java RMI framework. We develop a client, a server, and an API for communication.

Server

`ForumServer` binds a `Forum` to an RMI `Registry`. This represents the state of a replica, housing `Account` instances identified by a public key. Each contains an announcements `Board`. There is an additional instance of `Board`, the general board. Each `Announcement` has a unique ID. Server responses are encapsulated in `Response` objects, with different implementations for each operation. We tolerate f = 1 byzantine faults at most per interaction, so we use 3f + 1 = 4 server replicas.

Interface

`ForumInterface` provides an API defining client commands: `getNonce`, `getTs`, `register`, `post`, `postGeneral`, `read`, `readGeneral` and `readComplete`.
`ForumReliableBroadcastInterface` and the `EchoMessage` family are used for inter-replica communication.
`ClientCallbackInterface` defines the `writeBack` operation used in a (1, N) byzantine atomic register.

Client

`Client` employs a CLI to interact with the API and methods to analyze server responses to tolerate a potential byzantine fault and check for integrity errors. Requests are sent in parallel using threads through the `Request` family.

Security requirements and solutions

- **Resistance to message drops:** an attacker can drop messages. The system must detect this and resend the message.
- RMI uses TCP, which retransmits packets until an ACK packet is received.
- **Resistance to replay attacks:** an attacker can replay valid messages. The replicas must be able to detect replayed messages and reject them.

- We implement a challenge-response protocol using sequential nonces associated with each `Account`, included in all message signatures. For every request except `register`, the client invokes the `getNonce` remote method before it interacts with a replica.
- `register` is called without a nonce. The server returns 0 as the response's nonce if the key is already registered, giving an attacker the opportunity to replay this response, opening the possibility for a denial of service attack on the client.
- **Resistance to server failure:** a replica can crash, losing the existing data. It must be able to recover in a valid state.
- We employ timing redundancy through serialization as a backup of the current state of the replicas. For every state change in a replica, 2 files are sequentially created. A replica only starts working after the 4 servers are registered, but can still communicate if one of them crashes.
- **Message integrity**: an attacker can alter messages in transit. The replicas must be able to verify whether a message was altered.
- A SHA256 hash of the message parameters is sent with every message, which the receiver rejects if there is a mismatch. Integrity is guaranteed for all communications, as an attacker cannot recompute the hash without invalidating the signature.
- **Message authenticity:** attackers can pretend to be legitimate nodes. The replicas must be able to ensure that the sender is who they claim to be.
- The hash included with all messages is signed with the sender's private key. Private keys are unique for a node, guaranteeing authenticity.
- **Non-repudiation:** It must be possible to verify that an announcement was posted only by the associated author.
- Non-repudiation of messages is guaranteed by signatures. We guarantee non-repudiation for posted announcements by including the signature of the message that created the announcement, as well as the parameters of such message, in the announcement itself. **Byzantine servers:** Byzantine faults are tolerated by carrying out a consensus protocol between the nodes in the system:
- The replication of memory is seen as a shared memory problem, where replicated data represents a shared register. The `getTs` method is used on the client's first post operation to get the current register timestamp, which is incremented thereon. The consensus protocol is done in the client, which chooses a correct response from a Byzantine quorum. The signatures fulfill the requirement of authenticated point-to-point perfect links used in Byzantine registers.
- For the personal boards, we implement a (1, N) Byzantine atomic register using the Byzantine Quorum with Listeners algorithm. Newly read values are never older than previously read values. Each account keeps a set of listeners, to which a client is added to when reading. If the owner concurrently posts, `writeBack` is invoked on all listeners, fulfilling their read requests with the newly written value. Once the client has finished reading, the `readComplete` method is invoked, removing it from the set of listeners. A correct client will then choose the response with the highest timestamp from a quorum.
- For the general board, we implement a (N, N) Byzantine regular register by modifying the Authenticated-Data Byzantine Quorum algorithm to handle N writers. Either the last written value or the one being concurrently written is returned. To this end, we added, on the client side, a *read* phase to the `postGeneral` operation in order to get the current timestamp first. If two clients try to post concurrently, the request of the client with the highest *rank*

(ID) is considered when writing. When reading, the client chooses the response with the highest timestamp and rank pair, from a quorum. We did not implement an (N, N) Byzantine atomic register for the general board as we considered the validity property of a regular register enough to meet the project requirements.

- **Byzantine clients:** a Byzantine client can cause damage to the system in several ways:
- Write different values associated with the same timestamp or change the timestamp of a register to a value that is very high. We address this by requiring that a new write must be done with a timestamp that is equal to the current timestamp plus one, and break ties with a rank parameter. The Byzantine client can still pretend to have a high rank and prevent legitimate clients from writing to the register, if there is a tie. This is a denial of service attack and we did not consider it in our implementation.
- Send different messages to different replicas, or only send messages to some replicas. We addressed this by implementing Byzantine reliable broadcast using the Authenticated Double-Echo Broadcast algorithm. Whenever a replica receives a message, it forwards it to all other servers. Once a quorum is reached, each replica will forward the result with a "ready" signal. The command is executed only if there is a quorum, guaranteeing that all replicas execute the same command. If a replica hasn't received a request but has received 2 identical ready signals, it executes the command and sends its own ready signal.
- Collude with a Byzantine server, who saves malicious write requests and runs them after the Byzantine client is removed. This can be addressed by using *certificates* (Liskov & Rodrigues, 2006): every write is announced by the client beforehand, for which the replicas create a certificate (a quorum of authenticated messages) after validating the request. Every subsequent write operation must include the certificate of the previous operation to be accepted. We did not implement this.