Highly Dependable Systems
Sistemas de Elevada Confiabilidade
2020/21

Project - Stage 1
Highly Dependable Location Tracker

The world pandemic we live in calls for tools for dependable location tracking and contact tracing. The goal of the project is to design and implement a “Highly Dependable Location Tracker” (HDLT) system with the following characteristics:

- Users periodically send their location to a server. To do this they perform the following steps:
  - A user that wants to prove its location - known as the prover - broadcasts a location proof request to other users that might be nearby.
  - A user that receives a location proof request - known as the witness - checks if the prover is close enough and if that is the case sends back a reply. The reply encodes the fact that those two users were nearby at that time.
  - After receiving the proof(s), the user sends it/them to the location server.
- Time is split into epochs. Users prove their location once per epoch. Even though in a realistic scenario users can move during an epoch, for simplicity, we assume that correct users do not change their location in the same epoch.
- The location server stores, for each user, the sequence of location reports, containing, for each epoch, the user’s location and proofs. The server allows users to consult the stored data. There is a special user, representing the Healthcare Authorities (HA) that can obtain the location of all users across any epoch.
- The system should be resilient to attackers that aim to tamper with the integrity of the locations and proofs. Furthermore, attackers should not be able to generate proofs for users other than themselves.

The system must have a client-server architecture where each client and server has a Public key pair. For simplicity, students can assume that there is a Public Key Infrastructure in place, that performs the distribution of keys among all participants before the start of the system. The server should support concurrent clients. Furthermore, in this stage students should assume that there is a single server that

always behaves honestly. This restriction will be lifted in stage 2 by requiring multiple servers.

More specifically, the system should support the following interactions.
Users exchange location proofs with other users as follows:

- requestLocationProof(...)
  Specification: a user asks another user to generate a proof that they are close.

Users are able to perform the following requests to the location server:

- submitLocationReport(userId, ep, report, …)
  Specification: user *userId* submits a location *report.*
- obtainLocationReport(userId, ep, …)
  Specification: returns the location of the *userId* at epoch *ep*. This should only return the information of the user that submitted the request.

The HA client can query the location server as follows:

- obtainLocationReport(userId, ep, …)
  Specification: returns the position of *userId* at the epoch *ep.* The HA can obtain the location information of any user.
- obtainUsersAtLocation(pos, ep, …)
  Specification: returns a list of users that were at position *pos* at epoch *ep*

Design Requirements

The design of the system consists of three main parts: the user nodes which periodically prove their location and witness the location of others around them; the location server, responsible for keeping the proofs of location; and the HA client application that can query the server.

The witnessing mechanism is automatic and does not involve humans. This can be achieved, e.g., by broadcasting via bluetooth a message that will be detected only by nearby nodes. Note that students should only emulate such a mechanism in a simple way, as described in the next section.

Anomalous inputs shall be detected by the server, which should generate appropriate exceptions and return them to the client-side.

In this stage, the following assumptions are done:

- The server is honest.
- An attacker can drop, reject, manipulate and duplicate messages.
- There are no Sybil attacks; public and private keys are assumed to have been securely generated and distributed (e.g. citizen card, trusted CA).

- There can be up to *f* byzantine users in the **entire** system. For simplicity, we assume that there is a limit, f'<f, on the number of byzantine users that can be (or appear to be) nearby a correct user.

The system should guarantee the following properties:

1. A correct user should be able to eventually generate a report of its own location that is deemed valid by the system.
2. A correct user should not be able to repudiate any previously submitted location report.
3. A byzantine user should not be able to create a (false) proof of location on behalf of a correct user.
4. A byzantine user should not be able to create a false proof of its own location.

The students should not only design and implement a solution to ensure the above properties, but also discuss the conditions under which these guarantees hold.

Students will have to analyze the potential threats to the system such as man-in-the-middle or replay attacks, and design application-level protection mechanisms to cope with them. There are several approaches to address these issues, so it is up to students to propose a design and justify why it is adequate.

Note that even though the server is honest, it can crash and later recover. The implementation of the system should guarantee no loss (or corruption) of its internal state in the presence of crash faults. Communication among users should not be confidential, but communication between users and server must ensure confidentiality. Besides the system should operate under the assumption that the communication channels are not secured, in particular solutions relying on secure channel technologies such as TLS are not allowed.

Location Information

To encode the location information, we make the following simplifying assumptions:

- The space is represented by a bi-dimensional grid of a suitable dimension.
- Users are placed, at each epoch, in a given location in the grid. Students are free to place users in the grid at random or using some static assignment.
- Each epoch is represented by one and only one grid. Correct users are in a single location during an epoch and do not move. Byzantine users can behave arbitrarily.
- The usage of a localized broadcast communication medium, such as Bluetooth, will be emulated via the grid. At each epoch, a user consults the grid to determine which other users are nearby.
- Students are free to define the size of the grid, the number of users, and the nearby function as appropriate.

- For simplicity, students can pre-generate a sequence of grid assignments, one per epoch, before starting the system.

As an example, students can encode this information as follows:

```
user1, 0, 10, 20
user2, 0, 30, 40
```

which states that user *user1* in epoch *0* was at position (*10,20*).

Implementation Requirements

The project must be implemented in Java using the Java Crypto API for the cryptographic functions. We do not prescribe any type of communication technology to interface between the client and the server. In particular, students are free to choose between using sockets, a remote object interface, remote procedure calls, or a SOAP-based web service.

Implementation Steps

To facilitate the design and implementation of the system, students are encouraged to break up the project into a series of steps, and thoroughly test each step before moving to the next one. Having an automated build and testing process (e.g.: JUnit) will help students progress faster.

Here is a suggested sequence of steps:

1. Implement the user to user protocol that allows a given user to produce an acceptable location proof. For simplicity, a statically defined grid could be used.
2. Simple server implementation without dependability and security guarantees. Design, implement, and test the server with a trivial test client with the interface above that ignores the crypto parameters (signatures, public keys, etc.)
3. Develop the client library and complete the server – Implement the client library and finalize the server supporting the specified crypto operations.
4. Add the functionality of the HA client.
5. Extend the system to support the dependability and security guarantees specified above, namely tolerate Byzantine users.

Submission

Submission will be done through Fénix. The submission shall include:

- A self-contained zip archive containing:
    - The source code of the project and any additional libraries required for its compilation and execution.
    - A set of demo tests/scripts that demonstrate the mechanisms integrated into the project to tackle security and dependability threats (e.g., detection of attempts to tamper with the data, etc).
    - A mandatory README file explaining how to build the system and run the tests/scripts.
- A concise report of up to 4,000 characters addressing:
    - Explanation and justification of the design, including an explicit analysis of the possible threats and corresponding protection mechanisms.
    - Explanation of the integrity guarantees provided by the system.
    - Explanation of other types of dependability guarantees provided.

The deadline is **16/4/2021 at 19:00**. More instructions on the submission will be posted on the course page.