Highly Dependable Systems

Sistemas de Elevada Confiabilidade

2024-2025

DepChain

Stage 1

Goals

This project aims to develop a simplified permissioned (closed membership) blockchain system with high dependability guarantees, called Dependable Chain (DepChain). The system will be built iteratively throughout the first and second stages of the project: the first stage focuses on the consensus layer while the second stage will target the transaction processing layer.

As a starting point, we will use the Byzantine Read/Write Epoch Consensus algorithm from the course book [1] (Algorithms 5.17 and 5.18). The goal of this stage is to understand and implement a version of this algorithm with the following simplifying assumptions:

- The system membership is static for the entire system lifetime, including a pre-defined leader process. This membership information (namely the ids and network addresses of the leader and remaining system members that implement the blockchain, which we refer to as the blockchain members) is well-known to all participating processes before the start of the system (e.g., they can fetch this information from a static location).
- There is a Public Key Infrastructure in place. This means that blockchain members should use self-generated public/private keys, which are pre-distributed before the start of the system, and this information is also included in the static system membership.
- The implementation does not have to handle leader changes. For this to work, we assume the leader is always correct, i.e., it does not crash nor presents Byzantine behavior. The rest of the blockchain members might behave arbitrarily (and your implementation must support this). This simplifying assumption means that the algorithmic aspects related to epoch changes do not need to be implemented. That said, it is important to reason about the protocol features that you leave in place so that Byzantine leaders can be implemented in the future (and describe this reasoning in the final report). In other words, the implementation must always ensure safety, and must ensure liveness only if the leader is correct.

Design requirements

The design of the DepChain system consists of two main parts: a library that is linked with the application that users employ to access the system, and the blockchain members that are responsible for keeping the state of the system and collectively implementing the blockchain service. The library is a client of the blockchain service (implemented by the blockchain members, who act as servers), and its main responsibility is to translate application calls into requests to the service. Any client messages that do not obey the expected protocol behavior must be detected by the blockchain service and handled appropriately.

The following assumptions are done on the system's components and environment:

- The application can trust its local instance of the client library. This means that we are not concerned with attacks where a compromised library tries to attack an honest application.
- A subset of the blockchain members may be malicious and behave in an arbitrary manner.
- The network is unreliable: it can drop, delay, duplicate, or corrupt messages, and communication channels are not secured. This means that solutions relying on secure channel technologies such as TLS are not allowed. In fact, we request that the basic communication is done using UDP, with various layers of communication abstraction built on top. Using UDP allows the network to approximate the behavior of Fair Loss Links, thus allowing you to build (a subset of) the abstractions we learned in class.

For this stage, we will use a simplified client that only needs to submit requests to the service (in stage 1, a request is a simple string to be appended to the blockchain) and wait for a reply confirming if and when the request was executed (i.e., the string was in fact appended). The semantics of the interactions between clients and the service will be refined in the second stage of the project.

Note that there is a gap between the interface of the clients (which, recall, can run on different machines than the members of the blockchain) described in the previous paragraph (i.e., an interface accepts a request of the form <append, string>) and the interface of the BFT Consensus algorithm, which has the `Init`, `Propose` and `Decide` primitives. We leave it as an open design decision to devise a mechanism to fill this gap, i.e., translate client requests to invocations of the BFT algorithm, and for the respective response path. There are several approaches to address these issues, so it is up to students to propose a design and justify why it is adequate. Finally, on the blockchain side, there should also exist an upcall that is invoked whenever the DECIDE output is reached, for notifying the upper layer where the blockchain implementation resides. The implementation of the blockchain side (and of the corresponding upcall logic) will be extended in the second phase of the project. For now it can be handled by a simple service implementation that keeps an array with all the appended strings in memory.

You must also include a set of tests for the correctness of the system. For these tests to be thorough, they need to be more intrusive than simple “black box” tests that submit a load consisting of strings to be appended to the blockchain: in particular, there must be a way to change the way that messages are delivered or the behavior of Byzantine nodes to test more challenging scenarios.

Implementation requirements

The project should be implemented in Java using the Java Crypto API for the cryptographic functions.

The implementation should be structured as layered modules, namely by leveraging some of the abstractions that we learned in class. It is important to carefully reason about the appropriate layers and abstractions to leverage. Furthermore, their implementation must not only be semantically correct (as taught in class), but also practical and efficient, which might require design features that were not described in the book nor the slides.

Implementation Steps

To help in the design and implementation task, we suggest that students 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:

- Step 1: As a preliminary step, before starting any implementation effort, make sure you have carefully read and understood the Byzantine Read/Write Epoch Consensus (recall that epoch changes are not required for this stage). Reason about the design and implementation challenges that might arise before moving to step 2.
- Step 2: Implement the authenticated perfect links abstraction.
- Step 3: Implement the conditional collect and test it in a simple scenario with a set of processes.
- Step 4: Simple blockchain member implementation without dependability and security guarantees. Design, implement, and test the Byzantine Read/Write Epoch Consensus algorithm ignoring the cryptographic operations.
- Step 5: Develop the client library and complete the blockchain members – implement the client library and finalize the protocol, namely to support the specified cryptographic operations.

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. The archive shall also include a set of demo applications/tests that demonstrate the mechanisms integrated in the project to tackle security and dependability threats (e.g., detection of Byzantine behavior from blockchain members). A README file explaining how to run the demos/tests is mandatory.
- a concise report of up to 5 pages in LNCS format [2] with:
    - explanation and justification of the design, including an explicit analysis of the possible threats and corresponding protection mechanisms.
    - explanation of the dependability guarantees provided by the system.

The deadline is **March 13 at 23:59**. More instructions on the submission will be posted in the course page.

Ethics

The work is intended to be done exclusively by the students in the group and the submitted work should be new and original. It is fine, and encouraged, to discuss ideas with colleagues – that is how good ideas and science happens - but looking and/or including code or report material from external sources (other groups, past editions of this course, other similar courses, code available on the Internet, ChatGPT, etc) is forbidden and considered fraud. We will strictly follow IST policies and any fraud suspicion will be promptly reported.

References

[1] Introduction to Reliable and Secure Distributed Programming. 2nd Editon.

[2] Information for Authors of Springer Computer Science Proceedings. https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines