DepChain - Stage 1

André Gonçalves[1], Miguel Raposo[1], and Vasco Silva[1]

Instituto Superior Técnico

Abstract. The report details the first stage of development of a permissioned blockchain system that is resistant to Byzantine faults. The goal of the first stage was to prioritize the consensus algorithm while creating an interface that allows clients to append strings to a blockchain. This document seeks to explain the design of the system and its dependability guarantees.

Keywords: Blockchain · Byzantine faults · Consensus.

1 Introduction

The project revolves around the development of a permissioned blockchain system called Dependable Chain (DepChain), which has high dependability guarantees and is resistant to Byzantine failures. The project is divided into two stages: The first focuses on the consensus algorithm and the second will focus on the transaction processing algorithm. This report covers the first stage of development, which involves a replicated append-only string to test the Byzantine consensus algorithm. This system allows clients to request strings to be appended to the list of strings and read its current state. When appending a message, they will wait until the process is finalized.

The report is split in two parts: Section 2 details the design of each component of the system and their protection mechanisms against threats, and Section 3 analyzes the dependability guarantees of the system.

2 Design

2.1 Authenticated Perfect Links

Since we are working with a Byzantine model, some processes may forge messages, making them appear as if they were sent by another process. Consequently, the **no creation** property of perfect links cannot be guaranteed in such models. To mitigate this issue, authenticated perfect links can be employed, ensuring the following properties:

- **APL1 - Reliable Delivery**: If processes p_i and p_j are correct, then every message sent by p_i to p_j is eventually delivered by p_j
- **APL2 - No duplication**: No message is delivered (to a correct process) more than once

– **APL3 - Authenticity**: if correct process p_j delivers message m from correct process p_i, then m was previously sent from p_i to p_j

To enforce these properties on our channels, messages must be digitally signed by the corresponding sender, preventing Byzantine processes from jeopardizing or forging them. This approach requires each process to use a key pair: a **private key** to sign outgoing messages and a **public key** to allow other processes to verify these signatures.

The implementation of the authenticated perfect links is based from UDP communication. As such, the creation of a new link involves the creation of a DatagramSocket and the creation of a new thread that will read from the same socket and process incoming messages. Objects that wish to receive messages from a Link are represented as an Observer, that receive them through a given **update** function. This removes the need of polling and limits the amount of threads used.

To guarantee that, eventually, the destination process will receive any given message, the sender process will keep sending the same message, less and less frequently with every attempt through exponential back-off. The receiver of the message will notify the sender that it received the message through an acknowledgment, with the same id as the sent message. When the sender receives the acknowledgment, they will store it in a list of acknowledged messages and stop sending.

An additional improvement that could have been made is the use of HMAC instead of digital signatures in the messages sent by the processes, as HMAC is significantly more efficient in terms of computational cost while still ensuring message integrity and authenticity. One possible solution is to use the Diffie-Hellman key exchange to establish a shared secret key between processes, allowing them to compute the HMAC using that symmetric key.

2.2 Consensus

To achieve consensus among processes on the next string to be appended to the blockchain, the **Byzantine Fault Tolerant Consensus** [1] algorithm was implemented. This consensus mechanism was adapted to guarantee **weak validity** instead of **strong validity**, meaning that if all processes are correct and some process decides on a value v, then v must have been proposed by at least one process.

To manage the consensus rounds and handle the messages that were decided upon for execution, a **Consensus Broker** was implemented. This class is responsible for overseeing the progression of consensus instances and ensuring that the agreed-upon values are properly executed after the consensus process is complete. Incoming message shared via the observer pattern implemented in the link are assessed to determine whether a new consensus instance, identified by a round number, is required to handle this round or whether a previously aborted round needs to be restarted. When these conditions are met, a new thread is scheduled to handle this round and handles incoming consensus messages for

that given round through the collect method, which implements the conditional collect abstraction for each phase of consensus. This method returns the decided value if consensus ended in a final decision or *null* if the consensus was aborted. The consensus algorithm is divided into two distinct phases:

- **Read**: The leader collects the states from a quorum of processes $(2f + 1)$ to decide on the value to be written. These states consist of a **timestamp-value pair**, which represents the last state that was accepted, and a **write-set**, which includes all the writes made by that process during the current consensus round.
- **Write**: The states collected during the read phase are then sent to all other processes so that they can also decide on the value to write, as the leader's decision cannot be blindly trusted if it is Byzantine. All processes decide on the value to write and send these **Writes** until a quorum of them is reached. Once this quorum is achieved, **Accepts** are sent to indicate which value the processes are ready to decide on. The value is only fully decided when a quorum of **Accepts** for the same value is received.

For simplification, the epoch change mechanism has not been implemented. However, we still consider cases where processes abort due to the lack of consensus, primarily caused by the presence of a Byzantine process preventing agreement.

Every decided value corresponds to a transaction in the blockchain, allowing the execution of these transactions to be abstracted into a separate module, called the **Execution Module**. In this phase of the project, the Execution Module is solely responsible for appending new strings proposed by clients to the blockchain.

2.3 Application

The application provides two distinct operations for clients: they can either append new strings to the blockchain or retrieve its current state. For both operations, a client request is created and broadcast to all servers to ensure it reaches the leader, which then initiates the consensus process.

The client application will create its own authenticated perfect link to send requests to the servers and receive their responses. Details of the requests are sent in the form of a Client Request, with a unique id, a type of request (which currently are only **READ** and **APPEND**), and a transaction that, in the case of an **APPEND** request, is the string to be appended on each server's list, signed by the client. In the case of a **READ**, this property is *null*. To ensure that the client always sends this request to a leader, the command is broadcast to all servers. To verify the validity of the responses, the client will wait until $f + 1$ equal responses before returning the result (where f is the maximum number of faulty processes). If this state can't be reached, the request will be deemed a failure. This prevents Byzantine processes from sending fake responses to the client and gives the client assurance that the response came from correct nodes.

To receive client requests, each server will open a new authenticated perfect link with a different port. The peers listed in this link will be each client, which is currently hard-coded. When these requests are received, they will be processed by a Client Request Broker, who will perform different actions depending on the type of request. If the request is a **READ**, then it will read its current state and create a response with it. This **READ** is not dependent on consensus, ensuring fast reads.

When handling **APPENDs**, the request is stored in a queue of requests. Before executing the request, consensus is needed to ensure that every correct process performs operations in the same order. As such, the request will be pulled from the queue by the **Consensus Broker**, which is shared to other processes when the leader starts a new round of consensus. Once a decision is made, the request will be processed and the string will be appended. The server's response to the client's request will only be sent when this transaction is executed, which only happens after it has been decided via consensus.

3 Durability Analysis

In the Dependable Chain (DepChain) system, the key dependability guarantees ensure that the blockchain remains both reliable and secure despite potential Byzantine faults.

The system provides safety, ensuring that only one value is agreed upon in each consensus round, even in the presence of faulty or malicious processes, thus maintaining the integrity of the blockchain. The system also guarantees liveness, ensuring that progress is made as long as the leader process behaves correctly, and consensus will eventually be reached. Through its Byzantine fault tolerance [1], DepChain can tolerate up to f Byzantine processes (where f is the maximum number of faulty processes), continuing to operate correctly as long as fewer than $2f + 1$ processes are faulty. Additionally, the system ensures message integrity by using authenticated perfect links, which authenticate and protect messages exchanged during consensus. Consistency is maintained as the blockchain records only the agreed-upon values, and availability is ensured as long as a quorum of correct processes is functioning. Lastly, security is reinforced through cryptographic signatures that authenticate messages, ensuring their integrity and preventing forgery.

However, due to non-flexible conditions in the conditional collect, the system is vulnerable to Denial of Service attacks made by byzantine processes. This prevents the system from ever finishing rounds of consensus as the byzantine processes may send their malicious messages first, filling the threshold buffer which will result in an abort or a rejection of server responses due to mismatched messages.

References

1. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to reliable and secure distributed programming. Springer Science & Business Media (2011)