

## **Project Description**

(Version 0.2)

### **1. Objective**

The objective of the project is to facilitate student contact with the practice of fault-tolerant distributed systems programming, with a particular focus on the required protocols and algorithms for implementing the needed interactions and agreements among the processes of the distributed system, while dealing with failures affecting the normal process execution and communication.

### **2. Project description**

The project consists in the implementation of the Streamlet consensus algorithm that can be used to implement a distributed ledger. Such abstraction is sufficient for implementing blockchain or for replicating services using the state machine approach (that will be discussed in future lectures).

The project requires the implementation of a local transaction generation module and a node library capable of generating and ordering the blocks containing the transactions in a distributed way.

A description of the protocol to be implemented is provided in the following paper:

- Benjamin Y. Chan and Elaine Shi. Diego. **Streamlet: Textbook Streamlined Blockchains**. Proc. of the 2nd ACM Conference on Advances in Financial Technologies (AFT '20), October 21–23, 2020.

The paper is available at the course webpage on Moodle (the target protocol is described in Section 5): [https://moodle.ciencias.ulisboa.pt/pluginfile.php/517869/mod\\_resource/content/2/aft20-streamlet.pdf](https://moodle.ciencias.ulisboa.pt/pluginfile.php/517869/mod_resource/content/2/aft20-streamlet.pdf)

### **3. Tasks**

To complete the project, students are invited to carefully read the proposed paper, in particular the crash fault-tolerant version of streamlet, described in Section 5.

The objective is to implement a module that generate transactions, simulating the workload imposed by clients submitting transactions to one or more nodes. Then, the server node packs these transactions on a block and submit them to others for adding to the replicated blockchain.

#### ***1.1 Streamlet Protocol***

The protocol proceeds in incrementing epochs where each epoch is  $2\Delta$  rounds. The duration of an epoch is defined by the students, as a parameter from the system. An epoch can last 2 seconds, for example.

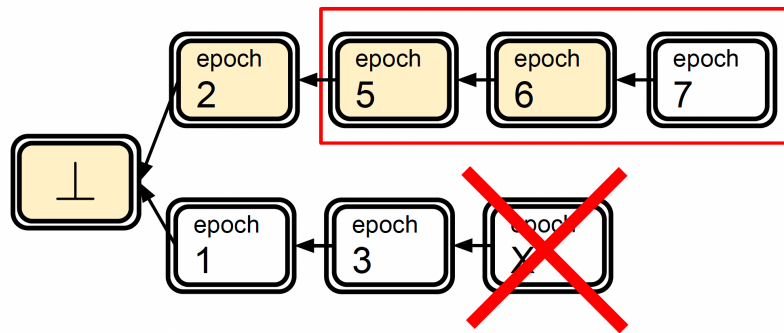
**Notarization.** A collection of at more than  $n/2$  votes from distinct nodes for the same block is called a notarization for the block. A valid blockchain is notarized by a node  $i$  if  $i$  has observed a notarization for every block (except the genesis block).

**Propose-vote protocol.** For each epoch  $e = 1, 2, \dots$

1. **Propose.** At the beginning of epoch  $e$ , epoch  $e$ 's leader  $L$  (which changes on each epoch) sends to everyone a new proposed *Block*, containing a hash of the previous block, epoch number, block length and transactions, where the parent chain is any of the longest notarized chains seen so far by  $L$ , and the transactions is the set of unconfirmed pending transactions.
2. **Vote.** Every node does the following during epoch  $e$ : When a node receives a proposed *Block* from  $L$  for epoch  $e$  for the first time, along with its notarized parent chain, the node 'votes' for the proposed *Block*, by sending a *Vote Message*, i.f.f. it is strictly longer than any notarized chain that the node has seen thus far.

**Finalization.** If at any time, a node sees three consecutive notarized blocks with consecutive epoch numbers, the node finalizes the second of the three blocks, and its entire parent chain. Whenever requested, the node outputs the longest finalized chain it has seen thus far.

**Echoing.** After a node receives a *Message*, it must broadcast it to all other nodes once.



**Figure 1: Streamlet finalization example.** In this example, the prefix of the top chain up to the epoch-6 block is considered final.

## 1.2 Data structures

We suggest the implementation of the following data structures for the project:

### *Transaction (tx):*

- *Sender* (integer): the sender of the transaction.
- *Receiver* (integer): the receiver of the transaction.
- *Transaction id* (integer): a nonce that together with the sender should form a unique id for the tx.
- *Amount* (double): the amount to be transferred.

### *Block:*

- *Hash* (bytes): SHA1 hash of previous block.
- *Epoch* (integer): the epoch number the block was generated.
- *Length* (integer): the number of the block in the proposer blockchain.
- *Transactions* (array of *Transactions*): the list of transactions on the block.

### *Message:*

- *Type* (enumeration of one of three values):
  1. **Propose**: to be used for Propose messages. The *Content* is a *Block*.
  2. **Vote**: to be used for Vote messages. The *Content* is a *Block*., with the *Transactions* field empty.
  3. **Echo**: to be used when echoing a message. The *Content* is a *Message*.
- *Content* (*Message* or *Block*)
- *Sender* (integer)

**Note:** All nodes of the system start with the genesis block, which has *Length*, *Epoch* and *Hash* equals 0.

### 1.3 Additional Features and Clarifications

For the 2<sup>nd</sup> phase of the project, it is expected the students to show their project can deal with forks and later make the blockchain converge back to single chain of blocks. The students are free to define a way to demonstrate this in their implementation, but bellow I describe one method for creating forks during a confusion time.

The strategy assumes each server put all received message in a queue, that is consumed by the protocol thread to process the messages and modify the protocol state. Besides, there is a function to determine the leader of each epoch. This means the implementation follows this structure (in Python-like pseudocode):

```
while True:
    if queue.size() > 0:
        msg = queue.dequeue()
        # process msg

def get_leader(n_servers):
    return random.randint(1, n_servers)
```

Given this structure, to create a period of confusion (i.e., in which forks would appear), take the following steps:

1. Define and initialize two new variables:
  - *confusion\_start*: the epoch after which the confusion starts.
  - *confusion\_duration*: the duration of the confusion in epochs.
2. Instead of using the loop mentioned above, use the following loop:

```
while True:
    if queue.size() > 0:
        if current_epoch < confusion_start or
           current_epoch >= confusion_start + confusion_duration - 1:
            msg = queue.dequeue()
            # process msg
```

3. Instead of using a function like *get\_leader* mentioned above, use the following function to determine the leader of each epoch:

```
def get_leader(n_servers, epoch, confusion_start, confusion_duration):
```

```

if epoch < confusion_start or
    epoch >= confusion_start + confusion_duration - 1:
    return random.randint(1, n_servers)
else:
    return epoch % n_servers

```

To see how this modification creates confusion, assume  $S=\{s_0,s_1,s_2,s_3,s_4\}$ ,  $n=5$ ,  $\text{confusion\_start}=0$ , and  $\text{confusion\_duration}=2$ .

In epoch 0, server  $s_0$  is the leader. It creates and proposes a block  $B_1$ . As no block is proposed before  $B_1$ , the genesis block is the parent of  $B_1$ .

In epoch 1, server  $s_1$  is the leader. It creates and proposes a block  $B_2$ . Because of our modification, no server processed  $B_1$  in epoch 0. Hence, the genesis block is the parent of  $B_2$ .

In epoch 2, server  $s_2$  is the leader. It creates a block  $B_3$  and proposes  $B_3$ . As all servers processed blocks  $B_1$  and  $B_2$  during epoch 1,  $B_1$  and  $B_2$  can be notarized at the beginning of epoch 2. Hence,  $B_3$  can extend either  $B_1$  or  $B_2$ .

#### 4. Delivery

The project is to be submitted through the course pages in Moodle. Only one submission per group is required.

Delivery consists of a zip file containing the developed code, a readme file, and the report.

- **Phase 1:** The first phase consists of the implementation of the basic protocol without fault tolerance or desynchronization of nodes. It is to be delivered until November 20<sup>th</sup>, 16:30hs.
- **Phase 2:** The second phase consists of the fault tolerant implementation of the protocol and its demonstrator with nodes being delayed, losing epochs and with capabilities of crash and recovery. It is to be delivered until December 11<sup>th</sup>, 23:55hs.