Implementing the Practical Byzantine Fault Tolerant Consensus Protocol for Blockchain Applications

CS 403 & CS 534 Distributed Systems
Term Project for Spring 2020

E. Savaş
Computer Science & Engineering
Sabancı University
İstanbul

Abstract

You are required to implement a consensus protocol that can be used for blockchain applications. You will use the Python programming language in your implementation, zmq sockets and REST API. In the implementation, there will be multiple processes that are directly communicating with each other using point-to-point links in a peer-to-peer (P2P) network. Before every consensus protocol run, one of the processes is dynamically elected to be the *proposer* and proproses a block of transactions to the other processes (*validators*), which will first verify the transactions in the block. If the verification is successful, a validator accepts the block and signs it. If at least $2k + 1$ processes sign a block, the block will be appended to the chain as the next legitimate block. Here is k is the maximum number of faulty (malicious) processes that can be tolerated, where there is a total of $n = 3k + 1$ processes.

1 Introduction

The project will be completed in three phases. The following sections provide the detailed explanations of each phase. Note that the system will be fully implemented at the end of the Phase III, and the first phase implements the infrastructure and the leader election protocol.

2 Phase I: Implementing the Infrastructure for the Consensus Algorithm and Leader Election Protocol

In this phase, you will develop Python codes to implement and *the index server* and *peer processes*, and the leader election protocol, which are explained in detail below.

2.1 Index Server

The index server implements a REST API that lets all peers to register to the P2P network by uploading their ID numbers, IP addresses (or just port numbers in this assignment) and public keys.

2.2 Peer

A peer process uses the REST API provided by the index server to register and learn about other active peers. A peer communicates with another peer by establishing a point-to-point communication link (via zmq sockets).

2.3 Leader Election Protocol

Before executing the consensus protocol, peers participate in the leader election protocol and as a result a peer is elected as the proposer of the consensus protocol (see the *Practical Byzantium Fault Tolerance* algorithm). In what follows, we describe the leader election protocol.

Suppose we have n peers $P_0, P_1, \ldots, P_{n-1}$. Each peer P_i generates a unifromly random number $r_i \in [0, 2^{256} - 1]$ and sends it to the other peers. When a peer receives random numbers from all other peers, it calculates

$$p = \texttt{SHA3\_256}^{(t)}(r_0 \oplus r_1 \oplus \ldots, r_{n-1}) \bmod n,$$

where $\texttt{SHA3\_256}^{(t)}$ stands for computation of the hash function t times successively. Here, $p \in [0, n-1]$ is the ID of the new proposer; namely P_p.

The peer P_i writes r_j values for $j = 0, \ldots, n-1$ as well as ID of the elected proposer peer p, it calculated, in a log file whose name is constructed as "election_i.log'. The peer signs them and appends the signature and its public (verification) key in the log file. Sample log files are given in the assignment package: "sample_election_0.log", "sample_election_1.log","sample_election_2.log".

2.4 Testing Your Code

To test your code in this phase of the project, we will use the Python code named "test4election.py", which reads the log files of all peers "election_i.log' for $i = 0, 1, \ldots, n-1$ and performs the following *verification* operation:

$$p = \texttt{SHA3\_256}^{(t)}(r_0 \oplus r_1 \oplus \ldots, r_{n-1}) \bmod n.$$

The test code also verifies the signatures in each log file. If the verification operation holds for each peer and log files have the same ID for the leader, your code passes the test.

We will use the following steps (in this order) when testing your submissions:

1. **We will run the index server first**: The index server is expected to implement the REST API.

2. **We will run peers; (with various values of n and t)**: The peers register, run the leader election protocol collaboratively and generate the log files; "election_i.log' for $i = 0, 1, \ldots, n-1$.

3. **We will modify and run 'test4election.py" for the log files generated in Step 2.**

3 Appendix I: Timeline & Deliverables & Weight etc.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement	NA	27/04/2020	NA
Phase I	Source codes	04/05/2020	30%
Phase II	TBA	TBA	TBA
Phase III	TBA	TBA	TBA

Notes:

1. You may be asked to demonstrate every phase to the TA.
2. Students are required to work in groups of two or alone.