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 run 'tester4phase.py" for the log files generated in Step 2.**

3 Phase II: Implementing the Consensus Algorithm for a Block of Transactions

In this phase, you will develop Python codes to implement the consensus protocol, which is a simplified variant of the Practical Byzantium Fault Tolerance (PBFT) algorithm. Once the leader

election protocol is executed (as explained in Section 2), one peer is elected to be the proposer and the rest of the peers are validators. Note that the leader election protocol needs to be modified as follows for the second phase

$$p = \texttt{succ}(\texttt{SHA3\_256}^{(t)}(r_0 \oplus r_1 \oplus \ldots, r_{n-1}) \bmod 2^{24}),$$

where `succ` is the successor function.

In what follows, we explain the steps of the PBFT protocol taken by the proposer and the validators.

3.1 Proposer

The **proposer** implements a round, in which it generates a block $\mathcal{B}$ of ℓ random transactions τ_i for $i = 1, \ldots, \ell$. It concatenates the hash of the previous block and the current block, hashes them and signs the result; i.e. it signs h^r, where $h^r = \texttt{SHA3\_256}(h^{r-1}||\mathcal{B}^r)$, r stands for the round number and $h^0 = \texttt{SHA3\_256}(\text{“”})$ is the hash of empty string. See the attached file “`TransactionBlock_Sample.py`” for sample block generation and “`sample_block_13385391_0.log`”, “`sample_block_13385391_1.log`”, ... for sample blocks. Note that the last line in the file is the proposer's signature for the corresponding block. Then, the proposer sends the transaction block and its signature to all validators.

Later, it waits for validators' messages, each of which contains a block of transactions signed by the corresponding validator. If it receives $2k$ blocks, which are identical to its own block, and the validator's signature verifies, it accepts the block as legitimate and starts the new round.

The proposer adds all the valid signatures for a block at the end of the corresponding file as [{`'pid': pid, 'signature':signature`},...]. The name of the file should be in the form “`block_pid_blockno.log`”. See the sample file “`block_5673504_3.log`” in the assignment package, which is the log file of the peer 5673504 for the block 3.

3.2 Validator

When a **validator** process recevies a signed transaction block from the proposer, if the signature verifies, it also signs it and forwards the block and its own signature to other validators. Then, it waits for meassages from other validators.

When a validator receives at least a total of $2k$ signed blocks, which are identical, it accepts the block as legitimate and starts waiting for the new block from the proposer. Similar to the proposer a validator adds all the valid signatures for a block at the end of the corresponding file as [{`'pid': pid, 'signature':signature`},...]. The name of the file should be in the form “`block_pid_blockno.log`”.

3.3 Testing Your Code

We will use the “`tester4phase2.py`” file for this phase of the project. Note that you must modify the code in “`tester4phase2.py`” in such a way that it will get the public keys of the peers from the index server. See the explanations in “`tester4phase2.py`”.

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 ℓ, r, n and t)**: The peers

(a) register to the index server,

(b) download the peer ids and their public keys (wait for some time after the register so that peers learn about all other peers.)

(c) run the leader election protocol collaboratively and generate the log files; "election_pid[i].log' for $i = 0, 1, \ldots, n-1$.

(d) run the PBFT consensus protocol and generate the log files; "block_pid[i]_j.log' for $i = 0, 1, \ldots, n-1$, $j = 0, 1, \ldots, r-1$.

3. **We will run 'tester4phase2.py" for the log files generated in Step 2.(d).** The tester code must get the peer ids and their public keys from the index server.
See "`phase2_samplelogs_and_pkeys.zip`" for *sample* log files and public keys.

3.4 What to Submit

You are required to submit the following files:

1. The source code of the index server: "index.py"
2. The source code of the validator peer: "peer.py"
3. The modifed version of the source code of the tester: "tester4phase2.py"

4 Phase III: Simulating Malicious Peers

In this phase, we will simulate four scenarios in which there are various numbers of malicious nodes when running the PBFT procotol. The PBFT protocol will be run as in Phase II; the only difference is that there are malicious nodes, which communicate with each other using a secret REST API. You are required to select a certain number of malicious nodes at random every time you run your simulation and you can assume that the malicious nodes know the secret REST API to get to know each other. The malicious nodes must know each other to mount a successful attack. For sake of simplicity, suppose malicious nodes apply the attack always in the last round (i.e., last block).

Each peer creates a directory and writes its received blocks and their associated signatures in log files using the following naming convention:

- **Directory naming convention:** "`ScNo_Peer_PID`", where `No` is the scenario number and `PID` is the peer id. For example, "`Sc1_Peer_441844`" is the directory for Scenario 1 for the peer with ID 441844.
- **Log file naming convention:** "`block_j_i.log`" where `j` is the round number and `i` is the block number a peer receives in one round. For example, "`block_4_0.log`" and"`block_4_1.log`' are the names of the log files for two different blocks a peer receives in round `4`.

We explain the four scenarios in the following.

4.1 Scenario I

In this scenario, there are exactly k malicious peers, where k is the number of malicous nodes that can be tolerated in the PBFT algorithm. Recall that $k = \lfloor (n-1)/3 \rfloor$, where n is the number of all peers. In this scenario, exactly k validators are malicious and the proposer and the other validators are honest. You are required to demonstrate the honest peers can collect sufficient number of signatures for one block and accept it even though it can get different blocks for a round due to the presence of malicious nodes.

In the assignment package, we provided sample log files in "`phase3.zip`" and the tester code "`tester4phase3.py`", where $n = 7$, $r = 5$ and the number of malicious nodes is 2. Note that there are five blocks in the chain (i.e., $r = 5$) and the honest nodes receive two different blocks in the last round. Here, two malicious nodes send a different block than what the proposer sends in the first step of the protocol. Thus, the honest nodes understand that the validator is honest and accepts the block with more signatures.

Note that we did not include the log files of the malicious peers in the samples as we are interested in the honest peers.

4.2 Scenario II

In this scenario, there are exactly k malicious peers, one of which is the proposer. The malicious nodes try to partition the honest peers into two *consensus groups* so that peers in different consensus groups accept two different blocks. But, they cannot succeed to do so as their number is not sufficient.

You are required to demonstrate that honest peers do not accept two different blocks and there are always $k + 1$ peers accept one block.

4.3 Scenario III

This scenario is similar to Scenario I, except that there are $k + 1$ malicious nodes, where k is the number of malicous nodes that can be tolerated in the PBFT algorithm. Here, the proposer is honest.

You are required to demonstrate that none of the honest peers accepts a block and the malicious peers can get any block accepted; i.e. a block which is different than the one sent by the proposer.

4.4 Scenario IV

This scenario is similar to Scenario II, except that there are $k + 1$ malicious nodes, where k is the number of malicous nodes that can be tolerated in the PBFT algorithm. Here, the proposer is malicious and this means that the malicious peers can partition the peers into two consensus groups, each of which accept a different block.

You are required to demonstrate that honest peers are divided against each other, where one consensus group accepts one block while the other accepts a different one. This will result in a *fork* in the chain of blocks.

4.5 What to Submit

You are required to submit the following files:

1. The source code of the index server: "`index.py`". The index server should now additionally implement a secret REST API so that malicious nodes communicate and apply an attack strategy.

2. The source code of the validator peer: "`peer.py`". Now, the peer can be malicious or honest in addition to being proposer and validator.

3. The modifed version of the source code of the tester: "tester4phase3.py"

5 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	Source codes	11/05/2020	30%
Phase III	Source codes	18/05/2020	40%

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.