Implementing Practical Byzantine Fault Tolerant Consensus Protocol for Blockchain Applications

CS 403 & CS 534 Distributed Systems
Term Project for Spring 2019

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 and zmq sockets. In the implementation, there will be multiple processes that are directly communicating with each other in a peer-to-peer (P2P) network . One of the processes is the *proposer* that 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 two phases. The following sections provide the detailed explanations of the project phases. Note that as the system will be fully implemented at the end of the Phase II, the first phase implements the infrastructure for the consensus protocol.

2 Phase I: Implementing the Infrastructure for the Consensus Algorithm

In this phase, you will develop Python codes to implement two types of peer processes: *proposer* and *validator*. There will be one proposer process and $n - 1$ validator processes.

2.1 Proposer

The proposer process implements a REST API that lets all peers (including itself) to register to the P2P network by uploading their IP addresses (just port number in this assignment) and public keys.

The proposer implements a round, in which it generates a block $\mathcal{B}$ of ℓ random transactions tx_i for $i = 1, \ldots, \ell$. It concatenates the hash of the previous block and the current block and signs the result; i.e. it signs h^r, where $h^r = (h^{r-1}||\mathcal{B}^r)$, r stands for the round number and $h^0 =$ “” is an empty string. See the attached file `ecdsa_sample.py` for sample block generation. Then, the proposer sends the transaction block and its signature to all validators.

Then, 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.

2.2 Validator

A validator process first registers itself by uploading its IP address and public key using the REST API. It can also read the public keys of other peers using the REST API for signature verification operation.

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.

2.3 Testing Your Code

In this phase of the project, assume all peers are honest (no malicious peer). Each peer (including the proposer) appends an accepted block of transactions to a file with name “chain_IPAddress.txt”, where the file name contains the IP address of the peer.

You need to submit a Python code called “tester.py”, which reads the files of all peers and performs two check operations:

1. The hash chain is valid; i.e., check if $h^r = (h^{r-1}||\mathcal{B}^r)$,
2. The final hashes of all peers are identical.

We will check your implementation with the following parameters (*demo*):

- $n = 10$ $(k = 3)$ and $n = 13$ $(k = 4)$
- $\ell = 10$ and $\ell = 20$
- $r = 10$ and $r = 20$

Make sure that your code works with different values of n, k, ℓ, and r.

2.4 What to Submit

You are required to submit the following files:

1. The source code of the proposer peer: “proposer.py”
2. The source code of the validator peer: “validator.py”
3. The source code of the tester: “tester.py”
4. The instructions to run your codes for the demo “README.txt”

3 Phase II: Implementing Practical Byzantine Fault Tolerance (PBFT) for Consensus

In this phase of the project you will implement a simplified version of the practical Byzantine fault tolerance (PBFT) protocol for a consensus on the blocks before they are added to the chain. This phase will be the same as the first phase except that the peers will implement the simplified PBFT protocol. Naturally, we will assume now that some of the peers can be malicious (traitors).

The protocol works in rounds, in each of which there will be one consensus as to which block will be added to the blockchain. The system consists of one *proposer* process and $n-1$ *validator* processes. In every round r, the protocol works in three steps: i) **PRE-PREPARE**, ii) **PREPARE**, and iii) **COMMIT**. The following are the protocol steps:

- In the **PRE-PREPARE** step, the proposer selects a block of (random) transactions $\mathcal{B}$, signs it and sends $(\mathcal{B}, \mathcal{S})$ to all other validators, where $\mathcal{S}$ is the signature for $\mathcal{B}$.
- The **PREPARE** step has two sub-steps:
  - A validator first verifies the signature of the block it recieved from the proposer. If the block verifies, it also signs the block and sends the block and its signature to the other validators.
  - All peers (including the proposer) waits for signed blocks sent by other peers. A peer first verifies the signature of each block it receives from other peers. If it receives at least $2k$ signed blocks that are identical, it proceeds to the COMMIT step. The peers that reached the COMMIT step form the *consensus group*.
- In the **COMMIT** step, a peer in the consensus group will add the block and all the signatures (there are at least $2k+1$ signatures for a block including its own) to the chain.

3.1 Testing Your Code

Each peer (including the proposer) appends an accepted block of transactions and all received signatures for the block to a file with name "chain_IPAddress_II.txt", where the file name contains the IP address of the peer.

You need to submit a Python code called "tester_II.py", which reads the files of all peers and performs three check operations:

1. The hash chain is valid; i.e., check if $h^r = (h^{r-1} || \mathcal{B}^r)$,
2. There are at least $2k+1$ signatures for each block in the chain,
3. The final hashes of at least $k+1$ peers are identical.

We will check your implementation with the following parameters (*demo*):

- $n = 13$, $\ell = 10$ and $r = 10$

Make sure that your code works with different values of n, k, ℓ, and r. As an example, you can consider the following use case scenarios:

1. Consider the case $n = 13$, where P_0 is the proposer annd $P_1, \ldots, P_{12}$ are the validators. Assume the coordinator and three validators are malicious (not honest). They try to form two different consensus groups, which accept two different blocks $\mathcal{B} \neq \tilde{\mathcal{B}}$, respectively. For example, malicious peers $P_0, \ldots, P_3$ send $\mathcal{B}$ to $P_4, \ldots, P_8$ and $\tilde{\mathcal{B}}$ to $P_9, \ldots, P_{12}$. Demonstrate they CANNOT be successful and **five** $(k + 1)$ honest validators accept the same block, the others accept no block.

2. For the case $n = 13$, assume now the coordinator and four validators are malicious. Similarly, they try to form two different consensus groups, which accept two different blocks $\mathcal{B} \neq \tilde{\mathcal{B}}$, respectively. Demonstrate they CAN be successful and two groups of three (k) honest validators accept two different blocks.

3.2 What to Submit

You are required to submit the following files:

1. The source code of the proposer peer: "proposer_II.py"

2. The source code of the validator peer: "validator_II.py"

3. The source code of the tester: "tester_II.py"

4. The instructions to run your codes for the demo "README_II.txt"

4 Bonus

In the COMMIT step, a peer in the consensus group waits for a random amount of time and sends its block and all the signatures for the block to all the other peers. If the receiving peer

- is in the malicious group it ignores the block.

- is honest but not in the consensus group, it adds the block to its chain if all signatures verify.

- is honest and in the consensus group it ignores the block and does not send its own block and the signatures for the block to other peers.

Your tester code "tester_II.py" reads the files of all peers and performs three check operations:

1. The hash chain is valid; i.e., check if $h^r = (h^{r-1}||\mathcal{B}^r)$,

2. There are at least $2k + 1$ signatures for each block in the chain,

3. The final hashes of at least $2k + 1$ peers are identical.

5 Appendix I: Timeline & Deliverables & Weight etc.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement	NA	03/05/2019	NA
Phase I	Source codes	10/05/2019	50%
Phase II	Source codes	19/05/2019	50%
Bonus	Source codes	19/05/2019	20%

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.