

# Practical Byzantine Fault Tolerance Consensus and A Simple Distributed Ledger Application

Hao Xu  
Muyun Chen  
Xin Li

## Abstract

Along with cryptocurrencies become a great success known to the world, how to deploy a large scale, robust Byzantine Fault Tolerant system turns into an interesting challenge in the technical community. We as a group of practitioners in distributed system are implementing the core consensus used in the distributed ledger – Practical Byzantine Fault Tolerance (known as PBFT, in Liskov), and designing a simple distributed ledger application of simulating the peer-to-peer transactions, in order to have a principle understanding the PBFT protocol, and its powerful strength to survive various software errors and malicious attacks.

## 1. Introduction

The objective of Byzantine fault tolerance is to be able to defend against Byzantine failures, in which components of a system fail with symptoms that prevent some components of the system from reaching agreement among themselves, where such agreement is needed for the correct operation of the system. Correctly functioning components of a Byzantine fault tolerant system will be able to provide the system's service, assuming there are not too many faulty components. The application basically simulates the account transactions (deposit, withdraw, move, etc) of the bank system, which is distributed with data replicated. During the process of the simulation, there might encounter PBF causing some nodes

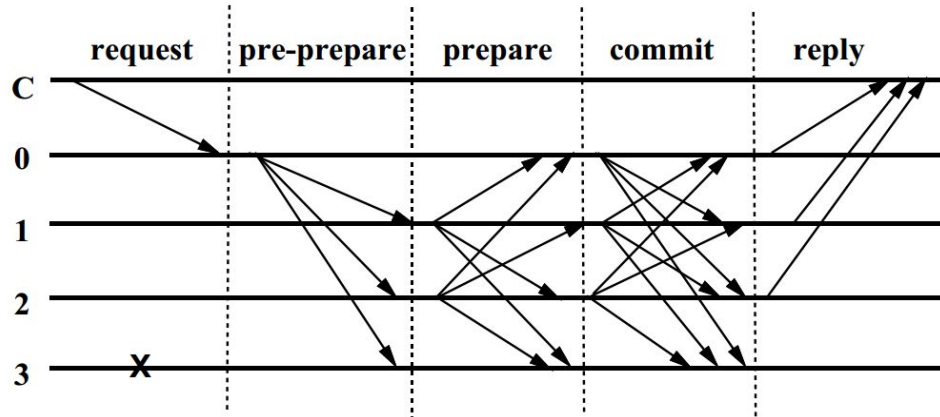
problem, but with the distributed ledger technology, the non-fault nodes can reach consensus to make the transaction succeed and correct.

One example of BFT in use is bitcoin, a peer-to-peer digital currency system. The bitcoin network works in parallel to generate a chain of Hashcash style proof-of-work. The proof-of-work chain is the key to overcome Byzantine failures and to reach a coherent global view of the system state. In our system, the application basically simulates the account transactions (deposit, withdraw, move, etc) of the bank system, which is distributed with data replicated. To simulate the Byzantine fault during the process, some fundamental standups below are giving us a

- a. Any node can crash and recover at any time.
- b. Use UDP to communicate between replicas, so the messages sent to each node might be lost, duplicated or disordered;
- c. Client send request to replica to deposit and retrieve money, and double check the consistency between replicas.
- d. Anytime with the  $3f + 1$  nodes, the system is able to survive  $f$  fault nodes.

## 2. Background

Figure 1 shows a normal case operation described in the paper. With four replicas (one primary) the system is able to tolerate one faulty node at a time.



Another important protocol is checkpoint. we will describe more detail of checkpoint endpoints in Section 3, here we only pinpoint that the checkpoint and digest are important to maintain data consistency in database.

### 3. Distributed Ledger

Basic: Build account database (including account information, balance, etc.) and replicated it into multiple nodes. Launch replica process in each replica node and each Process or some Processes share one database;

Distributed: a distributed ledger is a peer-to-peer system, each node can do transaction (communicate with other node) asynchronously;

- 1.1) support command line input to start the transaction in the client, eg:  
get current balance for A  
deposit \$100 to A  
move \$100 from B to A  
withdraw \$200 from B
- 1.2) Information updated automatically among different nodes. The consensus among the working nodes is supported by PBFT, and we also rely on database's log system to commit, redo and undo. After the consensus and commit, the information should be updated into each replica;

- 1.3) Simulate PBF to cause some nodes failed during the transaction. Since the messages are transferred with UDP, which is not reliable, the system can detect and process UDP related issues to keep accordance. We also manually shut down replica to simulate fault process.

### 4. Implementation

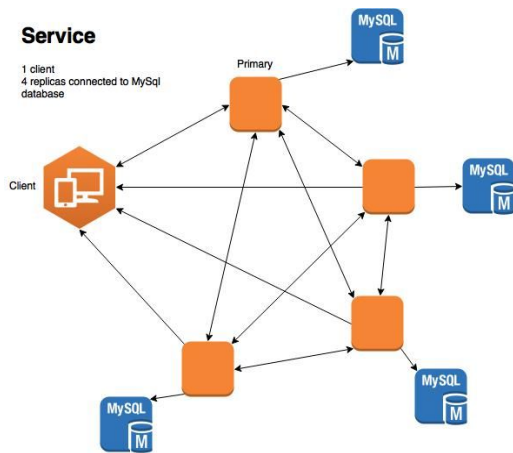
#### 4.1 PBFT Service

We reuse MIT BFT open source library [2], and sflite, a cryptography software tool [3] to help design our PBFT service. The *libbyz* library implement the PBFT algorithm described in the paper. It provides one client interface *invoke*, which sends the request operation to replicas, and one main server side interface *execute*, to receive and execute the requested operations.

The *mysql* library provides the endpoints connected to Mysql database, it has two interface *connectDB* and *executeQuery* for single-thread, as well as *connect* and *execute* for multi-thread execution.

The *PBFTservice* library provide the endpoint to connect between PBFT distributed system and databases. And it simulated some simple transaction samples described with more details in Section 4.

The service system from replica is like below:



For each replica:

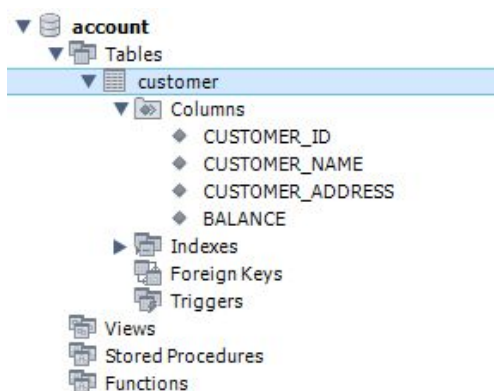
- wait infinitely for new transaction log.
- For the primary node, receive the input transaction from client and multicast to other replicas.
- Each replica will communicate with DB to execute and commit with log read and write one they receive the request.
- Communicate with other nodes with log read and write.

#### 4.2) AWS environment setup

We initiated five AWS instances for simulating the distributed system. We will not give too much detail about the environment. But we are using network packet IO data to evaluate the throughput of the system. This can be optimized to use other AWS services to have more accurate evaluation.

#### 4.3) Database

Install MySQL Workbench to create bank account database, the table structure is like:



We implement connection pool in the replica to communicate between each replica and database, so the transaction is committed within the connection pool managed by each replica. Also MySQL provide interfaces to maintain and manage the log and data and called by the replicas and clients.

### 5. Simulation

The main procedure involves three steps:

- 1) The client main procedure start transaction;
- 2) The replica main procedures start the pbft phases;
- 3) In the The replica execute *exec\_command* and-launch SQL work in database;

We will present more specific details in the Appendix Cases. You can have a better understanding of how we simulate the protocols.

### 6. Discussion

1. We run 10 times of simulation with 1000, 5000 and 10000 iterations each time. for each of the simulation, there are 500 read operations and 500 hundred write operations executed in random sequence. From the result we can tell if there are no crash/recover in replica, the data will maintain consistent in all replicas. But when there are crash/recover server, there are 2 out of 10 simulation in 5000 case, and 1 out of 10 simulation in 10000 case fails consistently. This should not happen since the protocol ensure the operations should be committed locally before reply to client. So we gave some assumptions here. First is the UDP is not stable, many packets might be lost. The second is the library

we use might have  
nondeterministic choices which

make the operations not executed  
at all.

Table 1 Simulation result.

No. iterations	read	write	time(s)	view_change	crash/recover	Accuracy
1000	500	500	3.15	3	0	100%
5000	2500	2500	11.6	5	1	80%
10000	5000	5000	23.47	4	2	90%

2. The combination of *view\_change* and *check\_point* enable the replicas to have consistency data after a node recover from the crash. Basically the new alive replica can obtain missing messages from other replica.
3. While restart the fault server the operation slow down dramatically. We assume it is because after the replica comes back to work, there are more communications between replicas and clients to resume lost messages, which will consume more times.

## 7. Improvement

An interesting implementation of PBFT is the peer-to-peer transaction system. For instance, some of the blockchain techniques are using the *Proof-of-work* based on the PBFT algorithm, which

would be an future development of our system.

## 8. Conclusion and Acknowledgement

The main goal of the project is to get all of us more familiar with the PBFT protocols. Even though the algorithm is published about 20 years ago, we can still find its significant influence over the technology world. We started from implementing the algorithm, integrating open source tools, and then proceeded to simulating the transaction and distributing it to databases. Finally now we fully understood the PBFT not only its protocol, but also its service and the way to implement it in the real world case. Even the simulation is simple and may contain flaws while handling complicated cases, but it is enough to maintain as a base case, which can be scale up to larger systems, and can be further designed to peer-to-peer transaction system.

## 9. Reference

- [1] Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." OSDI. Vol. 99. 1999.
- [2] Programming Methodology Group, MIT, <http://www.pmg.csail.mit.edu/bft/>
- [3] Sfs-lite, <https://github.com/OkCupid/sfs-lite/wiki>
- [4] Source code: [https://github.com/cmuhao/CS244b\\_final\\_project](https://github.com/cmuhao/CS244b_final_project)

## Appendix

- 1) Experiment 1: write one and read, which means replicas show the specified one customer's balance related information after it wirted.

Request from client: **deposit A 100, get A**

Result from replicas:

```
cmd Command Prompt - ssh -i *.pem ec2-user@13.57.32.246
Replica 1:
+-----+-----+-----+-----+
| CUSTOMER_ID | CUSTOMER_NAME | CUSTOMER_ADDRESS | BALANCE |
+-----+-----+-----+-----+
|          3 | Michael Chen | NULL              |    200 |
+-----+-----+-----+-----+

cmd Command Prompt - ssh -i *.pem ec2-user@54.193.27.25
Replica 2:
+-----+-----+-----+-----+
| CUSTOMER_ID | CUSTOMER_NAME | CUSTOMER_ADDRESS | BALANCE |
+-----+-----+-----+-----+
|          3 | Michael Chen | NULL              |    200 |
+-----+-----+-----+-----+

cmd Command Prompt - ssh -i *.pem ec2-user@52.53.168.124
Replica 3:
+-----+-----+-----+-----+
| CUSTOMER_ID | CUSTOMER_NAME | CUSTOMER_ADDRESS | BALANCE |
+-----+-----+-----+-----+
|          3 | Michael Chen | NULL              |    200 |
+-----+-----+-----+-----+

cmd Command Prompt - ssh -i *.pem ec2-user@54.183.251.79
Replica 4:
+-----+-----+-----+-----+
| CUSTOMER_ID | CUSTOMER_NAME | CUSTOMER_ADDRESS | BALANCE |
+-----+-----+-----+-----+
|          3 | Michael Chen | NULL              |    200 |
+-----+-----+-----+-----+
```

- 2) Experiment 2: 1000, 5000, and 10000 iterations, random deposit and withdrawal.

Result from replicas:

[illegible]

(1000 iterations)

(10000 iterations)

- Request from client: **deposit A 100, get A, deposit B 200, get B, withdraw C 100, get C**  
 Result from replicas: