CS 271 Fall 2020
Programming Assignment I

Assigned: October 5th, 2020
Due On Demo Day: Wednesday, October 21st, 2020

PLEASE NOTE:

- This programming assignment can be done in teams of TWO.

1 Introduction

Blockchain is a list of blocks which are linked using cryptography. Each block consists of multiple transactions and in each transaction, money is transferred from one (sender) to another (receiver). That is, when a sender sends money to a receiver, this interaction is stored as a transaction. In a blockchain, each block may contain one or more transactions along with the hash value of the previous block. Each client in the system has a copy of the blockchain, and one of the main challenges is how to insert a new block in the blockcahin consistently across all copies at the clients.

In this first project, you will develop a simplified distributed blockchain, where the local copies of the blockchain are consistent using clock synchronization. You should first implement clock synchronization using Cristian's algorithm, then develop the distributed blockchain protocol as described below.

2 Project Overview

We want you to implement a distributed blockchain in a synchronized distributed system. That is to say:

1. Every client saves a copy of the blockchain locally. Each block in the blockchain has a single transfer transaction.
2. There is an upper bound τ for the delivery time of each message sent in the network.
3. The clocks of all the clients are synchronized (using Cristian's Algorithm) with a maximum discrepancy of δ.

If all the clients have perfectly synchronized (ie, identical) clocks **and** communication delay is **zero**, then we can design a simple protocol that achieves consistency among these clients as follows: When a client wants to issue a transfer transaction, it just broadcasts the **timestamped** transfer transaction, and immediately inserts it in the blockchain in timestamp order (breaking ties using process ids).

However, in any real synchronous distributed system, clocks are not perfectly synchronized and the network does have communication delays. Hence, a more realistic protocol would proceed as follows: When a client wants to execute a transfer transaction, it broadcasts this transfer transaction to all the other clients. The message is timestamped using the synchronized local clock. Then it waits *long enough*, until it makes sure that it has received every possible transaction broadcast by the other clients earlier than its own transaction and has added such transactions to its local blockchain (again breaking ties using process ids). Finally it can add its own transaction to the blockchain.

Clock synchronization among all the clients can be achieved using Cristian's algorithm. An additional time server should be introduced to the system, and it should provide a simulated clock accepted by all the clients hosting the blockchain. As discussed in the lecture, assume the drift of all clocks is ρ and we want to ensure that clocks do not differ from each other by more than δ. Each client will send a message to the time server asking for the current time every $\delta/2\rho$ seconds. The time server responds with a message containing the current time C_{UTC}. The client sets its clock halfway between C_{UTC} and C_{UTC}+ReturnTime (C_{UTC}+$T_{round}/2$).

Please note:

- When dealing with drift, it is **usually not** a good idea to set the clock back, as discussed in class (eg, moving time backwards can confuse software development environments and event management). **However, for simplicity, you are allowed to set the clock back if it is needed in this project.**

3 Implementation details

- Each client should have multiple threads, one handles the clock synchronization, and the others handle the blockchain operations.
- The blockchain can be implemented as a list or linked list, with each node containing a single transfer transaction. A transfer transaction $\langle S, R, amt \rangle$ consists of a sender S, a receiver R, and the amount of money amt. This is a simplified blockchain which does not require any hash pointers or cryptography.
- You can use one thread to continuously listen to other clients' broadcasts, and adds all the received transfer transactions to a local buffer. Each transfer transaction should be timestamped using the synchronized clock, along with the process ids to break ties.
- Your blockchain client should accept two types of transactions:

1. balance transaction: the balance transaction should reflect the user's balance at the time of request. Each client can only check **its own balance**. Any transactions that occurred before this time should be included, and any transactions after this time should be excluded. The client first logs its current synchronized clock as the time of request, then waits $\delta + \tau$, sorts all the transactions in the buffer, and moves every transaction from the buffer to its local blockchain. It then traverses the local blockchain to calculate the balance, and stops if one transaction's timestamp is after the logged time of request.
2. transfer transaction: each client can only send **its own money** to the others. For example, client A can only issue transfer transactions with $S = \text{A}$, and $amt >= 0$. The blockchain first checks if the client has enough money to issue this transaction by traversing its local copy of the blockchain. If valid, it broadcasts this transaction to all the other clients, timestamped using its own current synchronized clock, and adds it to the local buffer. Then it waits $\delta+\tau$, sorts all the transactions in the buffer, and moves every transaction from the buffer to its local blockchain.

- Other than the implementation of all the required processes and network connections, you will also have to implement a simulation of a **drifted clock**. A simulation of drifted clock can be implemented by recording the system time ($sys\_time\_at\_sync$) and the simulated time ($sim\_time\_at\_sync$) each time the clock is synchronized. The time for the drifted client clock ($current\_sim\_time$), at any given moment($current\_sys\_time$), can be calculated by using the recorded information, i.e.,

  $current\_sim\_time = sim\_time\_at\_sync + (current\_sys\_time - sys\_time\_at\_sync) * (1 \pm \rho)$

  This is followed by an update of both $sys\_time\_at\_sync$ and $sim\_time\_at\_sync$.

- The clock drifts for every client could be the same ρ.

- For the purpose of the demo, be clear on the values of δ and ρ, which you can set appropriately to demonstrate protocol performance during the short demo time.

4 User Interface

1. We do not want any front end UI for this project. Your project will be run on the terminal and use stdio for the input/output.
2. When starting a client, it should connect to all the other clients and the time server. When starting the time server, it should also connect to all the clients. One way of doing this is to provide every client's IP or other identification info that uniquely identifies each client via a configuration file. Other appropriate methods are also accepted.
3. Through the user interface, we can issue transfer or balance transactions to an individual client. Once a client receives the transaction request from the user, the client

executes it and displays on the screen "SUCCESS" or "INCORRECT" (for transfer transactions) or the balance (for balance transactions).

4. You should log all necessary information on the console for the sake of debugging and demonstration, e.g. message sent to client XX, message received from client YY, or message sent to the time server. When the local clock gets synchronized, output the current clock value. When the client issues a transaction, output its current balance before and after.

5. You should include some delay when sending a message. This simulates the time for message passing via the network, and makes it easier for demoing concurrent events. The delay should be a random value with an upper bound τ, i.e., uniformly sampled from 1 to 5 seconds, where $\tau = 5$.

6. Use message passing primitives TCP/UDP. You can decide on which alternative and explore the trade-offs. We will be interested in hearing your experience.

5 Deployment and Demo

You can deploy the clients and the time server on one single machine by using several processes to simulate the distributed environment.

For the demo, you should have 3 clients and one time server. Initially, each client should have $10. Then the clients issue transactions to each other: A gives B $4. etc. You will need to display the blockchain it stores locally.