MP1 Report

Group members:

- yiyanw3
- hx13

Cluster Number:

- sp22-cs425-g10-01.cs.illinois.edu
- sp22-cs425-g10-02.cs.illinois.edu
- sp22-cs425-g10-03.cs.illinois.edu
- sp22-cs425-g10-04.cs.illinois.edu
- sp22-cs425-g10-05.cs.illinois.edu
- sp22-cs425-g10-06.cs.illinois.edu
- sp22-cs425-g10-07.cs.illinois.edu
- sp22-cs425-g10-08.cs.illinois.edu
- sp22-cs425-g10-09.cs.illinois.edu

How to run

Go to the directory of /MP1

Notice: you have to add the address and port in config.txt, then run the command below

```
make
python3 -u gentx.py [frequency] | ./mp1_node [node] config.txt > [node.txt]
```

Design document

Basic data structure

message format (defined in util.h):

C++ code:

```
struct message {
int messageType;          // 0: proposal, 1: reply, 2: decision; 3 - 4: for failure broadcast
int transactionType; // 0: Deposit, 1: Transfer
int amount;
int proposalServerID;
int messServerID;      // for messageType = 0 or 1, it's the server's ID of which send the message;
                       // otherwise, it will not be changed. It is used to break the ties of priority
```

```
    int priority;
    uint64_t time;              // the time, this message is generated
    char transferer[20]; // transactionType = 0: the account of the depositor;
                                    // transactionType = 1: the account of the transferer;
    char receiver[20];      // transactionType = 1: the account of the receiver;
};
```

Design ensures total ordering

*** **The total order is realized based on ISIS algorithm. There are three threads:****

* `transactionGenerator`: Generate transaction
* `localSender`:
  * Set up connection with other servers
  * After setting up connection with all other servers in the configure document, it sends messages to other servers
* `localReceiver`:
  * Listen to messages from other servers
  * Record the connection and disconnection of the servers
  * Analysis different messageType and make reaction accordingly to different message types.

*** **The general algorithm is:****

  * ISIS algorithm:
    * Each server maintains a varible `timestamp` for priority. At first, it equals to 0. And on sending a proposal or reply message, it increase by 1. When delivering a message, timestamp will be set as the maximum between timestamp and the priority of the message plus 1.
    * The sender send the proposal message to all other servers, and record the current priority in `inProcessMsg` and current message in `deliverBuffer`.
    * On receiving a proposal message:
      * reply its current `timestamp` and its `messServerID` for priority comparision.
      * Record the current in `deliverBuffer`.
    * After receving all replies from other servers, the sender choose the max priority (max(priority.messServerID) in ISIS). Deliver this message and send the decision message to all other servers.
    * On receving a decision message:
      * Update the message's messageType to final decision and modify the priority accordingly.

    * Deliver all the messages that could be delivered at the front of the priority queue.
  * Check validation of a transaciton:
    * When a decision message could be delivered, if the transaction is TRANSFER, check whether the amount of transferer's money in the account is not less than transfer amount. If this true, update the account information and print the BALANCE. Otherwise, ignore the illegal transaction.

*** **More detailed:****

* `transactionGenerator`: generates a tranction and puts the message in the `sendBuffer` (queue\<message>)

* `localSender`: read message from the `sendBuffer` and deal with the messages here in the order of they are pushed in the queue.

There are two messageType that it should deal with the context of the message.
- messageType == 0 (proposal message):
- set messServerID and proposalServerID as current servers' ID
- set current `priority` of this sender: `timestamp` is increased by 1 when sending a proposal message or a reply message
- put the message into `deliverBuffer` (priority_queue<message, vector<message>, cmp_pq> based on *priority.messServerID* in ISIS), to record the message and prevent other decision messages with higher priority delivered before this message.
- record the undelivered message produced by this server in `inProcessMsg`
- messageType == 1 (reply message):
- set messServerID as current servers' ID
- set current `priority` of this server

After that, send the message:
- messageType == 1: send the message to the server where it was proposed
- otherwise: send the message to all other servers

* `localReceiver`: poll the socket file descripter to get the messages from other servers

There are three messageType in basic function (without failure)
- messageType == 0: `produceReplyMessage`
- change the messageType to 1, messServerID to current server ID
- put the message in the `sendBuffer` (reply message)

- push the message into `deliverBuffer` for local reference
- messageType == 1: `produceDecisionMessage`
- modify the message priority recorded in `inProcessMsg`
- if (recoreded priority < priority of this message) or
(recoreded priority == priority of this message and recored messServerID < messServerID of this message)
update the *priority.messServerID* according to this message
- if the server has collected all other servers' reply:
- put the final decision from `inProcessMsg` in to `sendBuffer`
- call function `deliverMsg`
- messageType == 2: `deliverMsg`
- erase the record of current message pushed before with messageType == 0 or 1, in `deliverHash`
- push current message into `deliverBuffer`
- get message with the least priority in the `deliverBuffer`
- For proposal or reply messsage, which doesn't exist in `deliverHash`. Pop the message
- For other messages, check the transaction, if it's valid, show the BALANCE. Pop the message
- Repeat until `deliverBuffer` is empty or the top message could not be popped

*** **Justify****
* total ordering
It's quite similar as the proof in the lecture.
Suppose server delivers m_2 after m_1, when delivering m_1, there are three possible states of m_2

- Decision message of m_2 is in the queue, the corresponding proposal or reply priority (or message) of m_2 is not in `deliverHash` or even not in `deliverBuffer`, so $decisionPriority(m_2) > decisionPriority(m_1)$

- Proposal message of m_2 is in the queue and the server hasn't received decision message of m_2. As m_1 could be delivered st this point, we have $proposedPriority(m_2) > decisionPriority(m_1)$
According to the method of using max proposedPriority as decisionPriority for a message, we have
$decisionPriority(m_2) \geq proposedPriority(m_2)$
Therefore,
$decisionPriority(m_2) > decisionPriority(m_1)$
- Proposal message of m_2 is not in the queue, as

As `timestamp` will be set larger than priority of decision message m_1, we have $proposedPriority(m_2) > decisionPriority(m_1)$
similar like case 2,
$decisionPriority(m_2) > decisionPriority(m_1)$
And if anothor server delivers m_2 before m_1, we have
$decisionPriority(m_2) < decisionPriority(m_1)$ which conflicts with $decisionPriority(m_2) > decisionPriority(m_1)$

* reliable
When a proposer deliver its message, it also send the decision message to all other messages. Assisted by the socket API, it could guarantee this decision message sent to all other servers.
As there's no server failure in this step, all proposal or reply message will finally be replaced by the according decision message. Therefore, a decision message will finally be delivered. Therefore, all server will deliver this message.

Design ensures reliable delivery under failures

*** **The general algorithm is:****

- Each server maintains `lastMsgs`(vector\<message>) to record the last decision messages from all other servers.

- When server failure is detected, the server will check this vector, and if it has received decision messages from the failure server before, it will modify this message's type to type 3 and mutilcast this message to all other servers.

- On receiving a message with type 3, the server check the priority of this message with what it records in `lastMsg` of that failure server. If the priority of received message is higher, it means the failure server didn't have time to send this decision message to current server. So the server just deliver the message and update the corresponding item in `lastMsgs`.

- If the server deliver a message with type 3, it will also muticast this message to all other alive server to maintain reliability.

*** **More detailed:****

As there's may be a sudden failure of servers, the following modification should be made:

- `localSender`

  As we should only send messages to the alive servers, before call function send in the for-loop, we should check whether this server is alive.

- `localReceiver`

  - `deleteFD`: When a server fails, current server will call this function.

- Send the last decision message from this failed server to all other connected servers. (We need to change the message type to 3 before sending.)

- Change the `connectdServer` to record the disconnect with this failed server.

- Search for the message only waiting for the reply from this failed server to be delivered. If found, deliver it and send the decision message to all other alive servers.

- Add two messageType cases

- messageType == 3: `serverFailureHandler`

- If this decision message is after what is recorded in `lastMsgs` in current server, the server will:

- Update the corresponding item in `lastMsgs`

- Call `deliverMsg` to deliver this decision message

- Change the message type to 3 and muticast this message to all other connected servers

*** **Justify reliability****

We have server set {P_1, ..., P_N}, at some time, some server fails. Let the failed server set $C_F = {P_1, ..., P_i}$ and the alive server set $C_A = {P_{i+1}, ..., P_N}$

Let's consider the failure server P_1 with the following cases:

1. P_1 has sent its final decision message m_1 before its failure to all other alive servers.

- All alive servers must have delivered m_1.

- As for the other proposal messages, the servers will all discard them.

In this case, the reliability is satisfied.

2. P_1 has sent its final decision message m_1 to some of the servers in C_A, say C_R and they keeps alive before finish sending m_1 with message type 3

- On detecting the failure of P_1, servers in C_R will send m_1 to all servers in C_A. So each server in C_A will be able to deliver m_1

In this case, the reliability is satisfied.

3. P_1 has sent its final decision message m_1 to some of the servers in C_A, say C_R' and suddenly, all servers in C_R' fail before sending m_1

- All the alive servers will all discard m_1.

In this case, the reliability is satisfied.

4. P_1 has sent its final decision message m_1 to some of the servers in C_A, say C_R' and after servers in C_R' send message with message type 3 to some servers,say C_R,servers in C_R' all dead.

- The servers in C_A which receives m_1 with message type 3 will send m_1 with message type 3 to all alive servers. So all the alive servers will finally deliver m_1
In this case, the reliability is satisfied.
The other cases could be generated from these 4 cases. So the reliability is satisfied.
In a word, if the decision message is sent to at least one of the server in the alive server set, this message will finally be delivered by all the alive servers notified by message with message type 3.

Safety
Let's consider an alive server P_j where j > i with the following cases:
1. No proposal message without decision is sent from P_j to P_1 before P_1 fails.
- When collcecting the following reply message, P_j will ignore P_1 as it's not in the list of connected server.

2. P_j waits for reply from P_1 and other alive servers
- When P_j receives the replies from the other alive servers, the message will be delivered because `coonectedServer` shows P_1 is not connected, so P_j doesn't need to wait for its reply.

3. P_j waits for reply only from P_1
- In function `deleteFD`, when detecting the failure of P_1, the server will check whether there's a message could be delivered because the failure of other servers.
Therefore, the server failure will not have influence on other messages' delivery.

Graph Evaluation

*** **Graph Explantion****

1. We use Message Proposed ServerID and generated time to mark the delivery of each message in different servers.

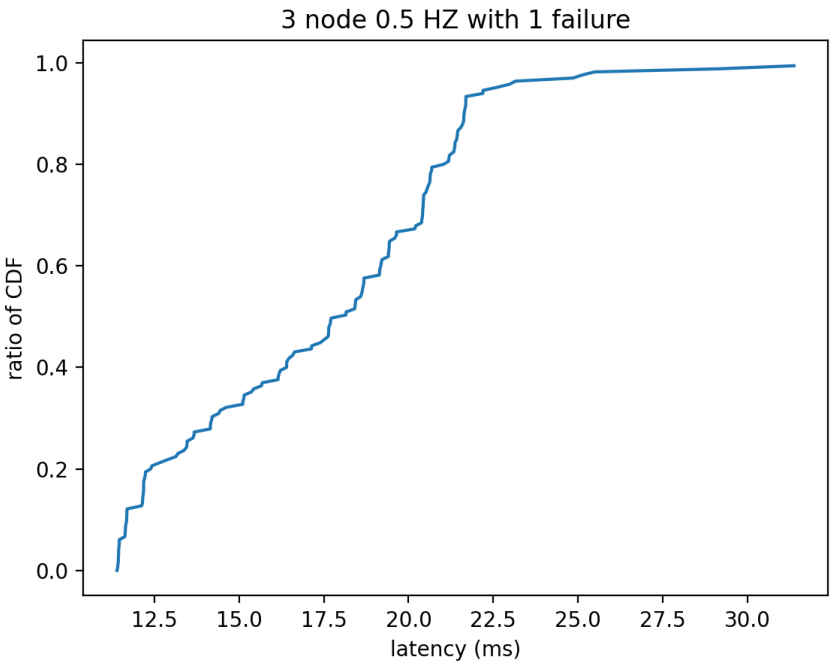
2. It took some time to establish the connections between the servers, and thus some of the messages has a very long time to be delivered. Therefore, we exclude the extremely large time. To be more specific, for each node scenarios, we calculate the medium of the time, and delivery time greater than 2 * medium is excluded.

*** **How to generate graph based on the result:****

```python3 code :
    python3 graph.py [number of nodes]
```

Generate Graphs

* **CDF with Scenario 3 nodes without failure**



* **CDF with Scenario 3 nodes with 1 failure**