MP1 Report

Instructions for Building and Running the Code

Clone GitLab Repository to Your Local Machine

Run

```
git clone https://gitlab.engr.illinois.edu/yl128/Group50-MP1.git mp1
cd mp1
```

(Optional) Compile Code

Our code is already compiled and included in the Git repository. We use Oracle JDK 19 to compile our code.

(Optional) Configure JDK version to JDK 19:

Run the following command:

```
sudo update-alternatives --config 'java'
```

Then select JDK 19 (or the latest version).

To re-compile, run the following commands:

```
cd src
bash compile.sh
```

Upload Code to VMs

You can clone our repository from GitLab on VMs or use our script (*upload.sh*). Please note that you will need to modify *upload.sh* because the script only works for our VMs.

Instructions for using *upload.sh*:

(1) Install *sshpass*

(2) Clone our repository to the local machine

(3) Run the following command:

```
bash upload.sh
```

(4) Enter your NetID, password, and the number of VMs to upload to

Run the Program

Connect to the VMs and run the following commands:

```
cd src
python3 -u your_generator frequency| java Launcher node_id your_configuration_file
(E.g., python3 -u gentx.py 0.5 | java Launcher node1 config.txt)
```

The program prints the balance of each account after every transaction to *stdout* and diagnostic messages to *stderr*. It also creates *account_log.txt* and *time_log.txt* in the *log* directory. Please note that writing into files is slower than printing to *stdout* so the log file may not match the last few lines in *stdout* if the process is interrupted.

- *account_log.txt* records the balance in each account that has a positive balance after every transaction
- *time_log.txt* records the timestamp of each transaction

Design

Message Object

We implement a Message class to store information relevant to a transaction message, such as message*Content, initialSender, timestamp, type, identifier, etc*. We use a string combining the initial sender's name and the timestamp as the unique identifier of a Transaction Message.
There are three types of Message, identified by its *type* field:

(1) Transaction Message that requires getting proposed priority from all other alive nodes to be delivered
(2) Proposed Priority Message sent from the receiver of a Transaction Message
(3) Final Priority Message multicast to other alive nodes from the initial sender of a deliverable Transaction Message

Design R-Multicast Protocol:

Initialization:

- A node will first try to connect to other nodes using the IP address (hostname) and port number provided in the configuration file. It will ignore any connection failure because the destination node may not be initialized at this time. After trying to connect with other nodes, it will keep listening to its port. After getting connected to a total number of N nodes (N provided in the configuration file), it will create a thread for each socket and start reading *stdin*.

Multicasting:

- When a node reads input from *stdin*, it will create a Message object (Transaction Message) to store information and multicast the Message using the socket pool.
- When a node receives a Message from any of its sockets, it will check the Message's type to determine whether it needs to multicast the Message to other nodes. If the Message is not a Proposed Priority Message, the node will multicast it to other nodes. We use unicast for the Proposed Priority Message so that a receiver of a Transaction Message only replies to the initial sender of a Transaction Message. The reason is that we only have to ensure that Total Ordering works for every alive node. If the sender fails before sending the Proposed Priority Message, its priority for that Transaction Message will not be considered.

Design of Total Ordering:

We use the ISIS algorithm to implement total ordering.
For each Transaction Message created at a node, we set its initial priority using the node's local priority (initial priority = local priority + 1) and add it to a PriorityQueue. We R-multicast this Transaction Message and track other nodes that reply with a Proposed Priority Message using unicast. For each Transaction Message, we store other nodes' names in a HashSet and associate the Transaction Message with the HashSet using a HashMap (key: Message identifier string, value: HashSet of socketID). We will first remove the Transaction Message from the

PriorityQueue after receiving a Proposed Priority Message from every other alive node (the HashSet becomes empty). Then we update the Transaction Message's priority and deliverable status and put it back to the PriorityQueue. After processing the PriorityQueue (handling transactions), the node will R-multicast a Final Priority Message to all other alive nodes.

Design of Failure Handling:

There are four types of failures:

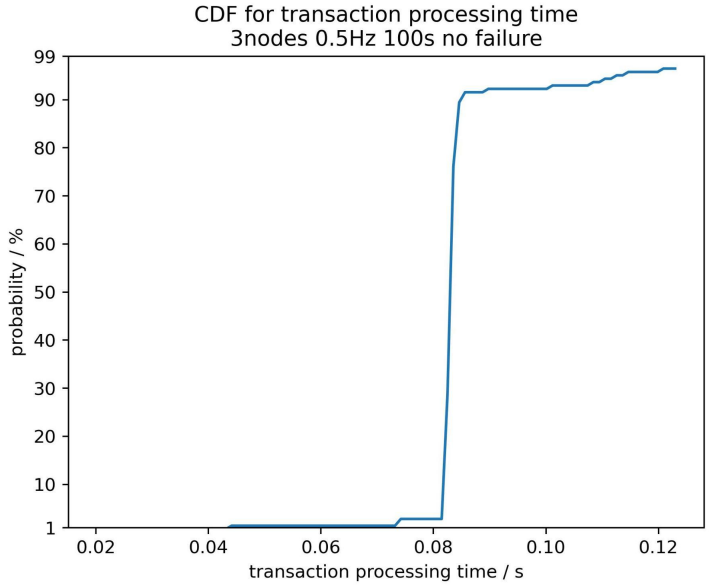
(1) Sender fails while multicasting a message
The other nodes don't know whether the sender successfully multicasts the final priority of its Transaction Messages to any other alive node so they cannot simply remove the sender's Transaction Messages from the PriorityQueue. The other nodes will first record the timestamp of the sender's disconnection. They will keep the Transaction Message from the failed sender in the PriorityQueue and continue to process other Messages.
If any Transaction Message becomes deliverable, the node will check the Head of its PriorityQueue and see if (1) the Head's (Transaction Message's) sender is already disconnected (2) the sender's failure timestamp is earlier than the current timestamp by at least 10 seconds (2 * maximum message delay between any two nodes). If both are true, the node will remove the Message from the PrioriyQueue and mark it as unsuccessful because no Final Priority Message was sent successfully from the sender to any alive nodes.

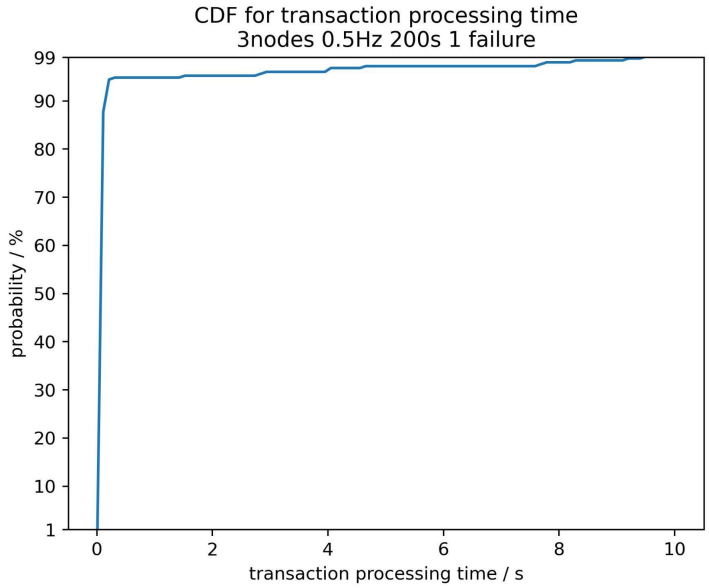
(2) Sender fails while multicasting the final priority of a message
Handled in the same way as for failure (1).

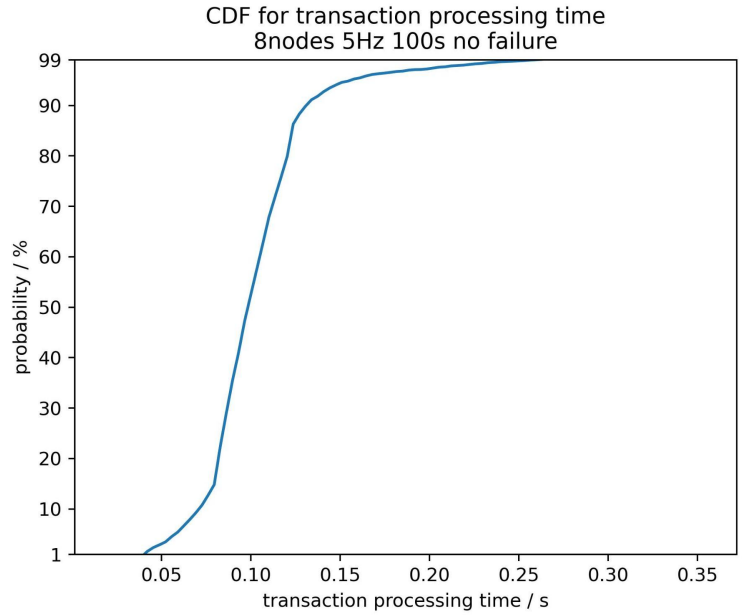
(3) A node fails before sending the proposed priority for a message
We use the above HashMap (key: Message identifier string, value: HashSet of socketID) to track which node has replied with a Proposed Priority Message. Each alive node will remove the disconnected node from the HashSet associated with every Message waiting for proposed priority.

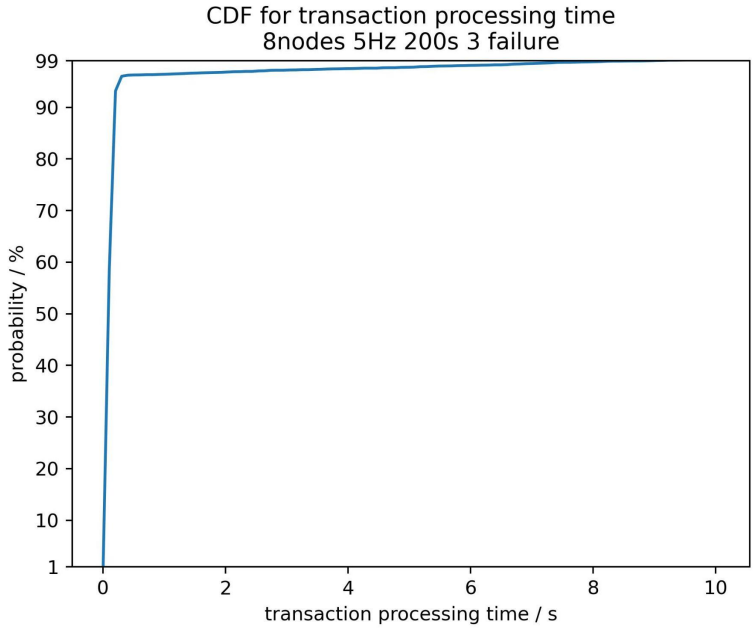
(4) A process fails after sending the proposed priority for a message
We will keep this proposed priority because it won't affect Total Ordering.

Graphs of the evaluation


CDF for transaction processing time
3nodes 0.5Hz 100s no failure
probability / %
99
90
80
70
60
50
40
30
20
10
1
0.02
0.04
0.06
0.08
0.10
0.12
transaction processing time / s


CDF for transaction processing time
3nodes 0.5Hz 200s 1 failure
probability / %
99
90
80
70
60
50
40
30
20
10
1
0
2
4
6
8
10
transaction processing time / s


CDF for transaction processing time
8nodes 5Hz 100s no failure
probability / %
99
90
80
70
60
50
40
30
20
10
1
0.05
0.10
0.15
0.20
0.25
0.30
0.35
transaction processing time / s


CDF for transaction processing time
8nodes 5Hz 200s 3 failure
probability / %
99
90
80
70
60
50
40
30
20
10
1
0
2
4
6
8
10
transaction processing time / s