MP1: Event Ordering

Due: 11:59 p.m., **Monday, March 6**

Notes

- You are welcome to switch languages or groups from MP0. If you do change groups, please let Manoj (gmk6@illinois.edu) know, so we can update the cluster assignments.
- You are allowed to use any programming language; however, the TAs will only help with the four supported languages: C/C++, Go, Java, and Python. Recall that MP2 must be in Go, so we encourage the use of Go in this MP as well.
- Your implementation will be tested on the CS425 VMs. It is your responsibility to make sure that the code runs on these VMs.

Overview

In this MP, you will build a distributed system of multiple processes (or nodes) that maintain accounts and transactions. It is often useful to ensure that we can have a global ordering of events across processes in order to perform consistent actions across all processes. Your events in this MP are *transactions* that move money between accounts; you will process them to maintain account balances, and use ordering to decide which of the transactions are successful. *A transaction delivered to any node must be delivered to all other nodes in your system, and all transactions must be processed in the same order by all nodes.*

You will therefore need to use totally ordered multicast to ensure the same ordering across all nodes in your system. You also will have to detect and handle the potential failure of any of the nodes in the system, and ensure that your multicast is reliable.

Accounts and Transactions

Your system needs to keep track of *accounts*, each of which has a non-negative integer balance. Each account is named by a string of lowercase alphabetic characters such as `wqkby` and `yxpqg`. Initially, all accounts start with a balance of 0. A transaction either deposits some amount of funds into an account, or transfers funds between accounts:

```
DEPOSIT wqkby 10
DEPOSIT yxpqg 75
TRANSFER yxpqg -> wqkby 13
```

The first transaction deposits 10 units into account `wqkby`; the second one deposits 75 units into `yxpqg` and the third one transfers 13 from `yxpqg` to `wqkby`. Every time each node delivers and processes a transaction, it should print out a list of balances:

```
BALANCES wqkby:23 yxpqg:62
```

All `DEPOSIT` transactions are always successful; an account is automatically created if it does not exist. A `TRANSFER` transaction must use a source account that exists and has enough funds; the destination account is again automatically created if needed. A `TRANSFER` that would create a negative balance must be rejected. For example, if after the above 3 transactions we processed:

```
TRANSFER wqkby -> hreqp 20
TRANSFER wqkby -> buyqa 15
```

The first transfer would succeed, but the second one would be rejected, because there would not be enough funds left in `wqkby` for the second transfer. The balances at the end would be:

```
BALANCES hreqp:20 wqkby:3 yxpqg:62
```

On the other hand, if the two transactions arrived in the opposite order:

```
TRANSFER wqkby -> buyqa 15
TRANSFER wqkby -> hreqp 20
```

then the transfer to `buyqa` would succeed and the transfer to `hreqp` would fail, resulting in:

```
BALANCES buyqa:15 wqkby:8 yxpqg:62
```

Each node must print the balance in each account every time it delivers and processes a transaction. You may omit reporting accounts with 0 balances; any account with a non-zero balance **must** be reported. Additionally, make sure the accounts in the balances are sorted alphabetically. Your output balances will be used to check the functionality of your code (exact same balances across all alive nodes).

Running the nodes

Setting up the connections

Each node must take two arguments. The first argument is an identifier that is unique for each node. The second argument is the path to a configuration file – the first line of the configuration file is the number of total nodes, and each subsequent line contains the identifier, hostname, and the port no. of these nodes. Additionally, make sure your code can run via an executable named 'mp1_node' as follows:

```
./mp1_node <identifier> <configuration file>
```

Consider a system of three nodes with identifiers node1, node2 and node3, where a node runs on each of the first 3 VMs in your group (say g01), and each node uses port no. 1234. The configuration file provided should look like this:

```
3
node1 sp23-cs425-0101.cs.illinois.edu 1234
node2 sp23-cs425-0102.cs.illinois.edu 1234
node3 sp23-cs425-0103.cs.illinois.edu 1234
```

The first line in the configuration file lists the number of nodes in the system (and the number of remaining lines in the file). Each node uses its own identifier (passed as the first argument) to find the corresponding port number it should listen on (as specified in the configuration file). It uses the remaining lines in the configuration file to know the address and port numbers of other nodes it needs to connect to.

We will use our own configuration files when testing the code, so make sure your configuration file complies with this format.

Each node must listen for TCP connections from other nodes, as well as initiate a TCP connection to each of the other nodes. Note that a connection initiation attempt will fail, unless the other node's listening socket is ready. Your node's implementation may continuously try to initiate connections until successful. You may assume no node failure occurs during this start-up phase. Further ensure that your implementation appropriately waits for a connection to be successfully established before trying to send on it.

Handling transactions and failures

Once nodes are connected to one another, each node should start reading transactions from the standard input, and multicast any transactions it receives on stdin to other nodes. This should follow the constraints of totally ordered, reliable multicast. Briefly, all nodes should process the same set of transactions in the same order, and any transaction processed by a node that has not crashed must eventually be processed by all other nodes. As mentioned above, each node must report all non-zero account balances after processing a transaction.

You should detect and handle node failures. Any of your nodes can fail, so your design should be decentralized (or, if you use a centralized node in some way, you should be able to handle its failure). Note that a node failure is an abrupt event, you should not expect that a failing node sends any sort of "disconnect" message. Your system should remain functional with 1 out of 3 nodes failing in the small-scale scenario and 3 out of 8 nodes failing in the large scale scenario (evaluation scenarios have been detailed below).

As we will soon discuss in class, truly achieving a total reliable multicast is impossible in an asynchronous system. However, to simplify this MP, you are allowed to make some reasonable assumptions. In particular, you can assume the use of TCP ensures reliable, ordered unicast communication between any pair of the nodes. Moreover, rather than writing your own failure detector, you can directly use TCP errors to detect failures (similar to how you detected disconnected nodes in MP0). You may further assume that a failed node will not become alive again. Finally, you may (conservatively) assume that the maximum message delay between any two nodes is 4-5 seconds. You must ensure that your system works as expected for the four evaluation scenarios described below in this specification.

Note that you may use other libraries to support unicast communication, message formatting / parsing, or RPC (over TCP). However, you must **not** use a multicast communication library for this MP.

Transaction generator

You can test out functionality by entering transactions directly into each node. We have also provided a simple transaction generator for you `gentx.py`. As in MP0, it takes an optional rate argument:

```
python3 -u gentx.py 0.5 | ./mp1_node node1 config.txt
```

By default it uses 26 accounts (`a` through `z`) and generates only valid transactions, but you are welcome to modify it any way you wish during your testing and enable occasional invalid transaction attempts. Note that we will likely use a different transaction generator in testing to explore corner cases.

Graphs

The key metric we will track in this MP is the "transaction processing time", measured as the time difference between when a transaction is generated at a node, and when it is processed by the last node in the system. You need to plot the CDF (cumulative distribution function) of transaction processing time. A datapoint (X,Y) on the CDF indicates that the Y%ile value of the metric is X. Your y-axis must range from 1-99%ile.

Evaluation scenarios

You will need to test your system in the following scenarios:

1. 3 nodes, 0.5 Hz each, running for 100 seconds
2. 8 nodes, 5 Hz each, running for 100 seconds

3. 3 nodes, 0.5 Hz each, running for 100 seconds, then one node fails, and the rest continue to run for 100 seconds
4. 8 nodes, 5 Hz each, running for 100 seconds, then 3 nodes fail simultaneously, and the rest continue to run for 100 seconds.

Plot the CDF of transaction processing time for each of the above scenarios.

Design document

You should write up a short description of the design of the protocol you are using, and explain how it ensures reliable message delivery and total ordering. Also explain how your protocol handles node failures. Your document should justify the correctness of your design.

Submission instructions

Your code needs to be in a git repository hosted by the Engineering GitLab. After creating your repository, please add the following NetIDs as members with *Reporter* access:

- gmk6
- jw22
- eashang2
- sm106
- radhikam
- ruiqiy3
- ayxu2
- hanbog2
- srm14

You will also need to submit a report through Gradescope. In your report, you must include the URL of your repository and the full revision number (git commit id) that you want us to grade, as shown by `git rev-parse HEAD`. Make sure that revision is `push`ed to GitLab before you submit. Only one submission per group – it is set up as a "group submission" on Gradescope, so please make sure you add your partner to your Gradescope submission.

In addition to this, your report should include:

- The names and NetIDs of the group members
- The cluster number you are working on
- Clear instructions for building and running your code. Please include a `Makefile` if you're using a compiled language! If there are any libraries or packages that need to be installed, please list those, too. As with MP0, we will run your code ourselves on our VMs and check that it works as expected, so it is important that your instructions are sufficiently clear.
- Design document described above
- Graphs of the evaluation as described above

High-Level Rubric

- Correct submission format and build instructions (5 points)
- Design document (25 points)
  - Design ensures total ordering (10 points)
  - Design ensures reliable delivery under failures (15 points)
- Evaluation graphs (10 points)
- Functionality testing (70 points)
  - 3 node test, no failures (10 points)
  - 8 nodes test, no failures (15 points)
  - Test with illegal transactions (15 points)
  - 3 nodes test with failures (15 points)
  - 8 nodes test with failures (15 points)

Acknowledgement

We would like to acknowledge Prof. Nikita Borisov for the initial design of this MP.
The source code of the website has been borrowed from Nikita Borisov