MP1: Event Ordering

Due: 11:59 p.m., Thursday, October 7

Notes

- You are welcome to switch languages or groups from MP0. If you do change groups, please update us so we can update the cluster assignments.
- You can reuse or extend your code from MP0
- 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.
- Your implementation will be tested on the CS425 VMs. It is your responsibility to make sure that the code runs on these VMs.
- You can use TCP to ensure reliable, ordered unicast communication between any pair of the nodes, and you may use TCP errors to detect failures. You may also use other libraries to support unicast communication, message formatting / parsing, or RPC. You **may not** use a multicast communication library for this MP.

Overview

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. In this MP, your events will be *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.

Note that in an asynchronous system, it is impossible to know for sure which of two events happened first. As a result, you will want to use totally ordered multicast to ensure the ordering. You also will have to detect and handle the potential failure of any of the nodes in your system, so unlike MP0, you will not be able to use a centralized node as a manager.

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`. After every successfully executed transaction, you must print balances.

Note: make sure you are printing out balances in the format given below. The first word of the line must be BALANCES (all capital) followed by a space separated list of {account name}:{account balance}. The balances MUST be sorted alphabetically by account name.

```
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 `TRANFSER` 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 hreap: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 `hreap` would fail, resulting in:

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

(You may choose to include or omit accounts with 0 balances; any account with a non-zero balance **must** be reported)

Running the nodes

Each node must take three arguments. The first argument is an identifier that is unique for each node. The second argument is the port number it listens on. The third argument is a configuration file – the first line of the configuration file is the number of other nodes in the system that it must connect to, and each subsequent line contains the identifier, hostname, and the port no. of these nodes. Note the configuration file provided to each node will be different (as it will exclude the identifier, hostname and port of that node). For example, 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 to node1 should look like this:

```
2
node2 fa21-cs425-g01-02.cs.illinois.edu 1234
node3 fa21-cs425-g01-03.cs.illinois.edu 1234
```

The configuration file for the second node will look like this:

```
2
node1 fa21-cs425-g01-01.cs.illinois.edu 1234
node3 fa21-cs425-g01-03.cs.illinois.edu 1234
```

And so on. 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.

Note: make sure your node can run using the EXACT command given below.

```
./mp1_node {node id} {port} {config file}
```

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 1234 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

As in the last assignment, we want to track the bandwidth of the nodes and the delay in message propagation to all nodes. Here are the metrics you should track:

- The bandwidth at each node
- The amount of time until a message is processed at all nodes

For the former, you should report the bandwidth for each of the nodes. For the latter, report when the first and last node (of those that have not failed) have processed each message. There is no set performance goal for this MP but excessive overhead may be penalized.

Evaluation scenarios

You will need to generate graphs to evaluate 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, runing 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.

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 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 submitted using GitHub Classroom. Please use the code posted to CampusWire to accept the assignment. You will also need to submit a report through Gradescope.

Your report should include:

- The names and NetIDs of the group members
- The cluster number you are working on
- 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.
- 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)
  - Clear explanation (5 points)
  - Design ensures total ordering (10 points)
  - Design ensures reliable delivery under failures (10 points)
- Evaluation graphs (20 points)
- Functionality testing (50 points)
  - Basic functionality (10 points)
  - Illegal transactions (10 points)
  - Low-scale test (10 points)
  - Large-scale test (10 points)
  - Test with failures (10 points)