March 6, 2023

ECE428 Machine Programming 1

Due: 11:59 p.m. on Monday, April 10th, 2023

The objective of this MP1 is to process transactions in a distributed system in a correct order. You can receive up to 10 marks for this MP, if you work yourself or with another student, and 5 (3) if you work in a group of 3 (4) students. The student groups can be different from MP0. You can use any programming language, however, please contact the TA first, if you want to use a language other than C/C++, Golang, Java or Python. You can reuse and extend your code from MP0. You can use TCP protocol to ensure reliable, ordered unicast communications between nodes, and use TCP error messages to detect failures. Using libraries to support unicast communications, message formatting / parsing, and RPC is allowed, however, using a multicast communication library is not allowed in this MP.

Overview

It is often useful to ensure that we have a global ordering of events across processes in order to perform consistent actions across all these processes. In this MP, the events are transactions that move money between accounts. The transactions are used to maintain account balances, and must be assumed in a correct order in order to decide which transactions were successful.

Since we assume an asynchronous system, it is impossible to know for sure which of the two events happened first. As a result, we need to use totally ordered multicast to ensure the right ordering. It is also important to detect and handle potential failures of any network nodes in the system, so unlike MP0, we cannot rely on a centralized node to be a manager.

Accounts and transactions

The 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 characters such as wqkby and yxpqg. Initially, all accounts start with a balance of 0. A transaction either deposits some amount of funds into the account, or transfers funds between the accounts. For example:

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

The first transaction deposits 10 units into account wqkby, the second transaction deposits 75 units into yxpqg, and the third transaction transfers 13 units from yxpqg to wqkby. After every successfully executed transaction, the account balances should be printed using the format:

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

The first word in the line above must be BALANCES (all capital) followed by a space separated list of {account name}:{account balance} sorted alphabetically by account name. The accounts with 0 balance can be included or omitted, but all accounts with non-zero balance must always be reported.

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, assume that after the above three transactions, we process the following two additional transactions:

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

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

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

However, if the two transactions arrived in different order:

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

Then, the transfer to buyqa succeeds, and the transfer to hreap fails, resulting in:

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

Running the nodes

The command for running a 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 number of these nodes. Note that the configuration file provided to each node is different, as it excludes the identifier, hostname and the port number of that node. For example, consider a system of three nodes with identifiers node1, node2 and node3, running all locally on the same host. Each node uses port number 1234. The configuration file provided to node1 looks like this:

```
2
node2 localhost 1234
node3 localhost 1234
```

The configuration file for the second node looks like this:

```
2
node1 localhost 1234
node3 localhost 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. The node implementation may continuously try to initiate the 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.

Furthermore, make sure your node can be started using this EXACT command:

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

Handling transactions and failures

Once the 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 all 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 print out 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 are going to soon discuss in the 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, using a multicast communication library is not allowed.

Transaction generator

The functionality can be tested by entering transactions directly into each node. There is also a simple transaction generator 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 it generates only valid transactions, but it can be modified to also occasionally generate invalid transactions. Note that we will likely use a different transaction generator in testing, to explore corner cases.

Expected output

As in MP0, we want to track the bandwidth of nodes and the delay in message propagation to all nodes, i.e., the following two metrics should be tracked:

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

For the first metric, report the bandwidth for very node. For the second metric, report when the first and the last node (of those that have not failed) have processed every message sent. There is no performance goal set, but excessive overhead may be penalized.

Evaluation scenarios

Generate the graphs with two metrics described above for the following four 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

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

Submission instructions

The design document and the code should be submitted via Gitlab. The graphs can be inserted into this document, and submitted separately. Please include instructions for building and running your code. Use Makefile, if you are using a compiled language. If there are any libraries and packages that need to be installed, mention this explicitly in the description (e.g. in a README text file).

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)
- Performance 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)