**Team:** Livar Cunha, Luciano Mandryk, Sergio Clemente

# Compiling & Executing

Just run ./compile.sh or ./execute.sh

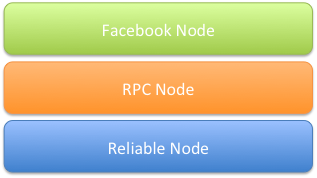
If you run into any issues make sure you set your java home to something like the following:

|  |
| --- |
| [~] JAVA\_HOME=/Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/Home/bin  [~] export JAVA\_HOME |

Also, make sure the *classes* folder exist.

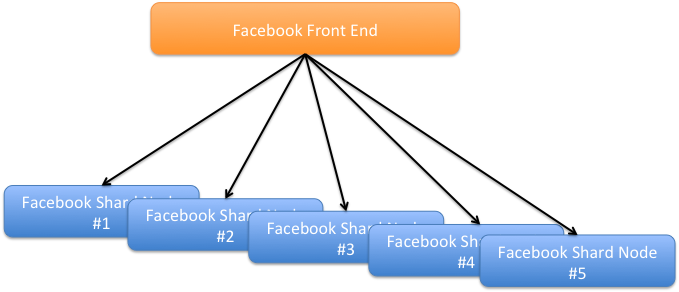
# Facebook System

We reused some of the code of assignment1. For example, the reliable message delivery system was kept intact. We changed our RPC node in order to have a cleaner API for returning values (We noticed we were duplicating too much code to handle exceptions or return values). The basic class architecture of the node is given below.



The main idea is that the reliable node provides reliable message delivering. The RPC node allows to define and call functions with a well defined API. The Facebook Node contains the logic of this assignment.

Let’s dig more into the Facebook Node which contains the logic for the assignment. The facebook nodes are divided into two roles as can be seen below:



* Front End: Contains logic that routes the request to the proper shard. For example, in our assignment we hardcoded the number of shards to 5.
  + Shard#1: Responsible for storing the information of users that starts with letter a-e
  + Shard#2: Responsible for the range f-j
  + Shard#3: Responsible for the range k-o
  + Shard#4: Responsible for the range p-t
  + Shard#5: Responsible for the range u-z

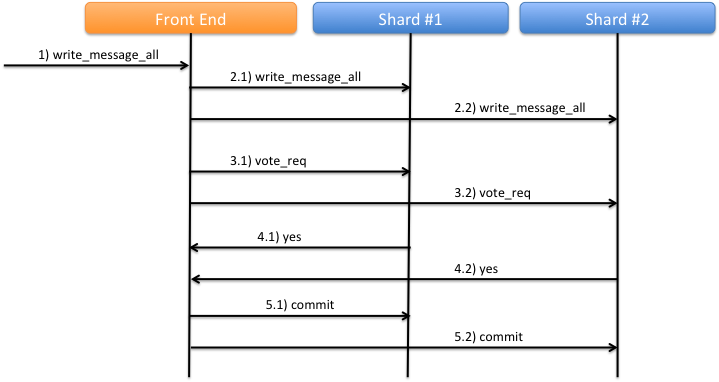
Node: We are assuming the users are evenly distributed. We could use a hash value based on the userId + some salt to make the distribution more independent of the login bug it was not required per the specification.

The front end node also contains the logic to coordinate the 2 phase commit protocol (More details below).

* Shard Node: Contain all data related to the users that were mapped to that node.

# Two Phase Commit

The two phase commit logic was written outside of the node and both the frontend node and shard nodes call them separately. We just implemented the two phase commit in the write\_message\_all operation. The details is as follows:



When the *write\_message\_all* command is received, the frontend starts the 2pc context and sends the *write\_message\_all* to all shards. Each shard figures out which users are friends with the user posting the message and stores the posted messages in memory (the reason for keeping them in memory at this point is that in case of a crash, the transaction is automatically aborted from that shard’s perspective).

The frontend waits for all shards to reply, up to some timeout. After every shard replies saying that they either succeeded or failed, the frontend starts the two phase commit protocol (in case of 100% success) or aborts the transaction.

Upon receiving a *prepare* request from the coordinator, each shard stores the messages in durable storage, but on a temp file. Upon receiving the *commit,* the temp file is made the official state file, therefore making the changes visible and effectively committing the transaction. The operation realized by *commit* is idempotent, so it can be retried multiple times (e.g. during recovery, if it keeps crashing) without incurring data loss.

From the shard’s perspective, aborting a transaction is as simple as deleting the temp file created during *prepare* and forgetting the transaction id. The coordinator maintains a list of all known transactions and the decision made for each of them. Upon recovery, the shards ask the coordinator (because of the uncertainty period) about the state of any pending transaction and commit or abort appropriately. The coordinator is never uncertain, so in case of recovery it aborts any pending transaction (Stating the obvious: “pending” implies that the coordinator hasn’t got to the point of making a commit / abort decision for that transaction. Transactions which have been decided are not touched during recovery).

# Example of running

The example below creates the frontend and shard nodes (As we mentioned below, they are hardcoded from 0-5) and adds users a,k and z. adds some friends and them broadcasts some message.

|  |
| --- |
| start 0  start 1  start 2  start 3  start 4  start 5  0 create\_user a a  0 create\_user k k  0 create\_user z z  0 login a a  0 login k k  0 login z z  0 add\_friend a;1234 z  0 add\_friend a;1234 k  0 accept\_friend z;1234 a  0 accept\_friend k;1234 a  0 write\_message\_all a;1234 message to k and z |

# Assumptions

The main assumptions we’ve made during the implementation are listed below:

* Node 0 works as the frontend, nodes 1-5 are the backend processing the requests.
  + This is only to simplify debugging and testing. In fact, every node can be used as a frontend with no changes (just use a different address before the command). Also, every node can be the backend, it’s just a matter of changing the hashing function to include node 0 or nodes higher than 5.
* No concurrent transactions.
  + We block (i.e. return error) any attempt to execute a write\_message\_all command while another is already running. This is to avoid the pain of dealing with interfering changes or having to implement shadow storage or record-level locking, which would be impossible to do correctly in the timeframe of this assignment.
  + We also fail any commands (e.g. create\_user) that are issued while 2PC is running (between *prepare* and *commit/abort*). Allowing other commands would require changing the state that was persisted during *prepare* and violate the re-do rule, potentially causing data loss.

# Failure Handling and Recovery

The following primary cases have been tested seem to be working properly:

* Coordinator fails after sending *prepare-request* messages to all participants but before receiving (all) votes:
  + The participants all vote Yes, but eventually timeout waiting for a decision. We expect the coordinator to be back online by then. The participants ask the coordinator about the decision for the transaction, to which the coordinator replies Abort, and the participants correctly abort despite having voted Yes.
* Participant fails after voting Yes and before receiving Commit/Abort:
  + Upon recovery, the participant realizes that it has a pending transaction to which it has voted Yes and requests the decision from the coordinator. The coordinator replies with the decision and the participant commits/aborts accordingly.

The other cases have been less tested but are expected to be working as well. There’s one side issue that we haven’t had time to debug and fix: some combination of failures and disconnections may leave some shards thinking that they still have an active transaction (in the front end code, not in 2PC), and since we don’t allow concurrent transactions, that shard will refuse to execute subsequent write\_message\_all commands. Restarting the node fixes the issue.