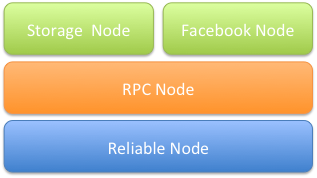
**Team members:** Luciano Mandryk, Livar Cunha, Sergio Clemente Filho

# First assignment

For building the Storage and Facebook layers, we created two abstractions to help the implementation:

1. Reliable Node: Implements in order delivery, at most once semantics, message retransmission and a connection oriented protocol similar to TCP (We will talk about the similarities below)
2. RPC Node: Implements a RPC layer that serializes automatically the method name and arguments. This layer also implements some command queuing to avoid sending a command to the server when there is an outstanding command still being executed (More explained below)

The diagram can be seen below:



## Compiling

To compile just run ./compile.sh like shown below.

|  |
| --- |
| [sergiombp15 ~/Documents/distributed\_systems/assignment1] ./compile.sh  [sergiombp15 ~/Documents/distributed\_systems/assignment1] |

If you have any issues make sure that the classes folder exists.

## Running

To run just run either:

|  |
| --- |
| ./execute.pl -s -n node.facebook.FacebookRPCNode |

Or

|  |
| --- |
| ./execute.pl -s -n node.storage.StorageSystemServer |

## Reliable node

The two most important methods are:

1. sendReliableMessage(int targetNode, byte[] payload): Sends a message reliably using the underlying unreliable transport.
2. onReliableMessageReceived(int from, byte[] payload): It’s called when a packet is received (It makes sure to discard duplicates, and properly order the packets)

The package format is given below:

1. From (self explanatory)
2. To (self explanatory)
3. Package Type (unknown=0, data=1, ack=2, connect=3, reset=4)
4. Connection ID (uniquely identifier of the connection)
5. Sequence Number (packet sequence number, used for detecting duplicates and reordering)

Each field contains 32 bits for simplicity. We also decided to include the from/to in the packet for debugging purposes (With the current framework we didn’t explicitly needed those fields). The Package Type can be of the following 4 types:

1. Connect: Sent initially to establish the connection. Notice this is a little different from the TCP 3 way handshake (syn, ack, syn/ack). There is no need for the last syn/ack since the connection is stabilized on each direction. We have the connection establishment for mainly handling crashes, 2 scenarios are explained below:
   * We can detect stale packets when a node crashes (For example, the client sent packet A and B, A was received, then the server crashed. Then when the server restarts it can detect that B was a packet from the previous session)
   * We can notify the client that a given connection doesn’t exist so it can reset
2. Ack: For each either Data or Connect packet, an ACK is sent. If an ACK is not received within 3 clocks, the packet is retransmitted. The timer increases next time to 6, then 9. Which after that the connection will abort (onConnectionAborted will be fired and a reset will be sent to the other end)
3. Data: For each user packet, a Data packet is sent. We are not handling if a packet doesn’t fit in a MTU (Maximum transmit unit).
4. Reset: The connection reset is fired currently under 2 scenarios:
   * We receive a stale packet (For example, a packet from a different connection)
   * We give up retrying to send the packet 3 times

The method that is called when the connection gets reset is onConnectionAborted.

## Facebook Node

As mentioned earlier the facebook server is a RPCNode and implements the following rpc calls:

1. create\_user <login> <password>
2. login <login> <password>
3. logout <token>
4. add\_friend <token> <friend\_login>
5. accept\_friend <token> <friend\_login>
6. write\_message\_all <token> <message>
7. read\_message\_all <token>

Operation #2 returns the token of the user session and is used in the other operations (3,4,5,6,7). The operation #7 returns all read messages in the format:

From: <from>

Content: <message>

Etc.

One example of client server interaction is given below:

|  |
| --- |
| start 0  start 1  1 create\_user a apass  1 create\_user b bpass  1 login a apass  1 login b bpass  1 add\_friend 9561723318 b  1 accept\_friend 1583707579 a  1 write\_message\_all 9561723318 hello world  1 read\_message\_all 1583707579 |

Note1: Due to the limitation of the framework, the server is hardcoded to be at address 0. The client can be any address that is not 0.

### Recoverability of Facebook node

For the previous example, commands will be recorded to the log file:

1. create\_user
2. add\_friend
3. accept\_friend
4. write\_message\_all

The log file looks like:

|  |
| --- |
| create\_user a apass  create\_user b bpass  add\_friend a b  accept\_friend b a  write\_message\_all a hello world |

Notice that we omit the token and just use the login for identification purposes. This makes the logfile easier to read and don’t compromise security since the operations were already validated before being logged to the disk.

When the server restarts it will execute the operations in the same order as in the log file.

## Storage System Node

A class called StorageSystemServer implements the storage API. Both the Client and the Server interfaces are implemented by it.

The client overrides the executeClientCommand method and handles any inputs from the Manager. It parses the string commands and calls the appropriate methods. The client has two methods to implement each of the functionalities from the Storage System: begin and end. When a new command comes through executeClientCommand, it calls the appropriate begin method. For instance, if the command is a create, it will call beginCreate. When the server terminates executing the create command, it will call endCommand. When the client receives end command, it checks that the server didn’t send any error messages (if it does, the client shows the error message). After checking that there was no error message, the client removes the current command from the command queue and starts executing the next command (if any). In case the command returns any value, like get, then the server will call endGetCommand, and the client will treat that differently. If there is no error, the client will check the parameters returned and print them.

The server overrides onMethodCalled. Much in the same way as the client, it parses the received command and calls the appropriate method. Differently from the client, the server only has one method for each functionality from the Storage System. For instance, for a create command, it calls createFile. Each of the server methods, after they are done executing their respective functionality, they will call endCommand (or endGetCommand) on the client and return to the client any return values or any error found.

It is worth of note that the StorageSystemServer inherits from RPCNode and therefore uses the same queue mechanism that is implemented there. In this way, any commands received from onCommand is put in a queue and executed in order, one after the other.

Also, StorageSystemServer overrides onConnectionAborted to deal with timeouts where the server takes too long to responde. When onConnectionAborted is called, a timeout error message is printed, the current command is removed from the queue and the next command in queue begins executing (if any).

One example of client server interaction is given below:

|  |
| --- |
| start 0  start 1  start 2  1 create 0 exampe.txt  1 put 0 example.txt Example Text  2 get 0 example.txt  2 append 0 example.txt Continuing the example  1 delete 0 example.txt |