Networking Layer Design

Kaylin Devchand & Cristian Stransky

Introduction

The network layer allows nodes to connect to the rendezvous server and to other nodes. Once connected, the rendezvous server and nodes can send messages back and forth using the dedicated communication protocol. For the purpose of this document, **a Rendezvous Server is a Server, and a Node is a Client.**

Communication Protocol

Messages

The communication protocol consists of sending messages back and forth. These messages are of the abstract type Message. The first field in Message is an enum called MsgKind that contains all the subclasses of Message. Other types of messages can easily be created by implementing a class that extends Message and adding that class to the MsgKind enum. Below is the Message definition.

```
enum class MsgKind { Ack, Put, Kill, Register, Directory};

class Message : public Object {
    public:
    MsgKind kind_;  // the message kind
    String* sender_; // the index of the sender node
    String* target_; // the index of the receiver node
    size_t id_;     // an id t unique within the node

    ...
}
```

A simple use case of Message is when the node wants to send a registration message to the rendezvous server. The below code constructs a Message of type Register and sends it to the rendezvous server.

```
String* node_ip = new String(“127.0.0.2”);
String* r_server_ip = new String(“127.0.0.1”);
Message* reg = new Register(node_ip, r_server_ip);
send_message(r_server_ip, reg);
```

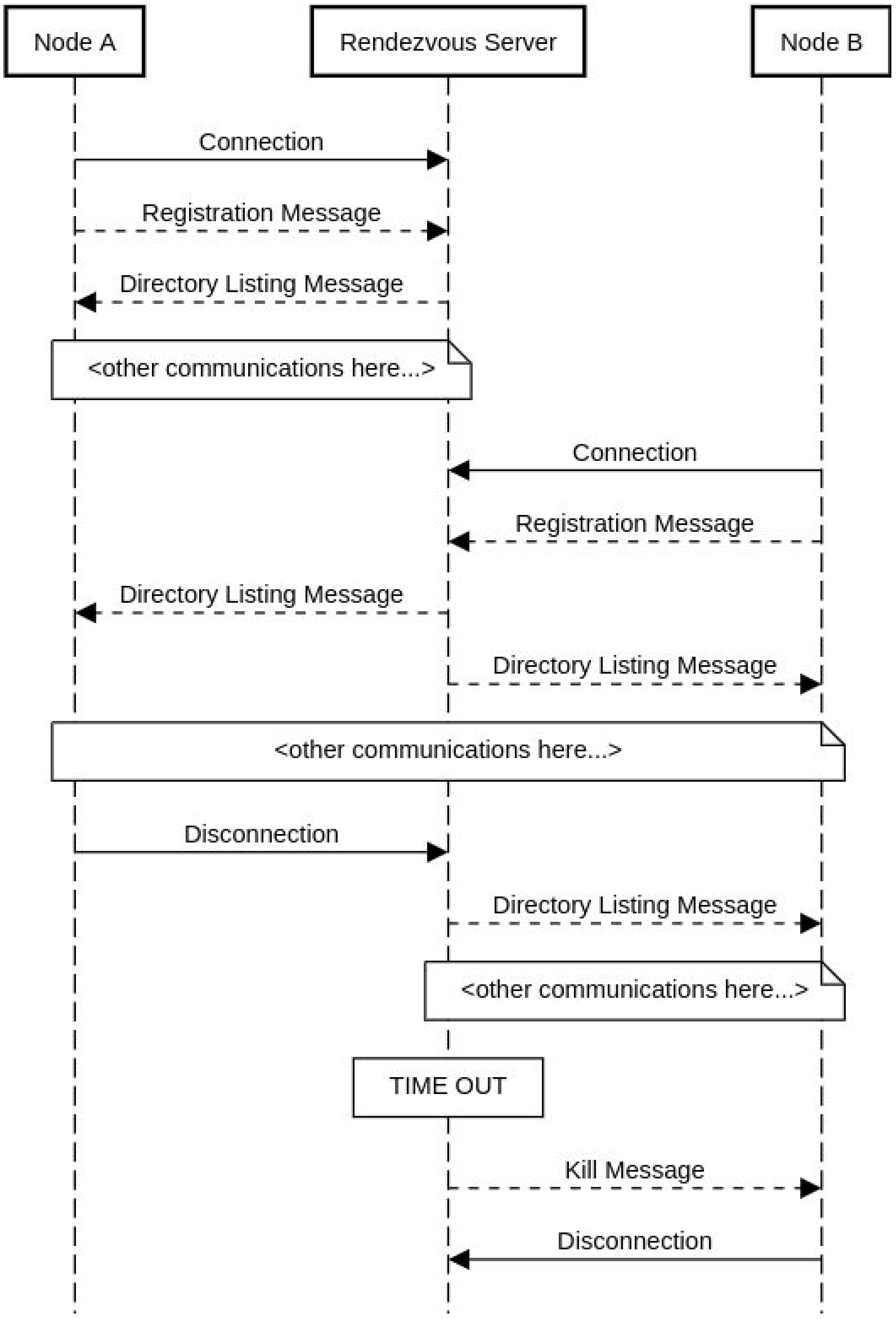
When a Message is sent, it is serialized to a byte array. The length of the serialized byte array is sent first; then the serialized byte array is sent.

Rendezvous Server to Node Sequencing

The diagram below shows how the basic communication between a node and rendezvous happens. The "<other communication here..>" notes represent where the user of the API can add their own messages.

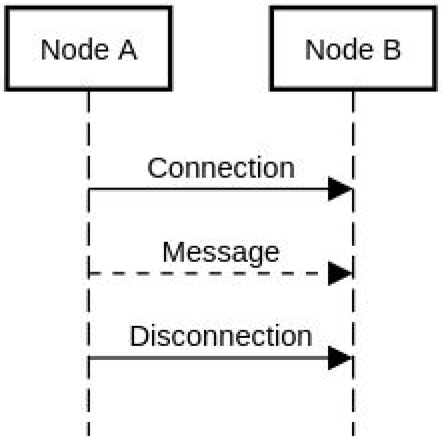
The sequencing starts when a Node connects to the Rendezvous Server. The Node will first send a Registration Message to the Server with its IP then the Rendezvous Server will send the Node a Directory Message that contains all the IPs of the Nodes currently connected to the server. Every time a Node connects or disconnects, the Rendezvous Server sends an updated Directory to all connected Nodes.

A Rendezvous Server can also be given a timeout parameter that tells it how long a period of inactivity can be before it shuts down. When the Rendezvous Server shuts down, it will send a Kill Message to all connected Nodes. The Nodes will then also shut down.


Node A
Rendezvous Server
Node B
Connection
Registration Message
Directory Listing Message
<other communications here...>
Connection
Registration Message
Directory Listing Message
Directory Listing Message
<other communications here...>
Disconnection
Directory Listing Message
<other communications here...>
TIME OUT
Kill Message
Disconnection

Node to Node Messaging

The diagram below shows the messaging between nodes. When Node A sends a message to Node B, it creates a connection with Node B, sends the message, then disconnects from Node B. If Node B needs to send a response message back to Node A, it needs to make a new connection to Node A, send the message, then disconnect. Nodes do not register with each other.



API Design

The high level design consists of three classes, Server, RServer, and Node. Server is the parent class and contains most of the implementation while RServer and Node are subclasses.

Server

There are four public methods of Server and a constructor:

1) `Server(const char* ip_address):` The constructor opens a socket to accept incoming connections.
2) `void run_server(int timeout):` Run server monitors all sockets and responds to incoming messages. The timeout parameter the maximum seconds of inactivity before this function returns.
3) `void shutdown():` This method shuts down the server by closing all of its connections
4) `void send_message(String* ip, Message* message):` Sends the given message to the client with the given ip.
5) `void send_message(int fd, Message* message):` Sends the given message to the client on the given socket.

Use case: Create a server, send a message to a client, then shut down.

```
String* server_ip = new String("127.0.0.1");
String* client_ip = new String("127.0.0.2");
Server* server = new Server(server_ip);
```

```
server->run_server(10);
String* hi = new String("hi");
Message* m = new Put(server_ip, client_ip, hi);
server->send_message(client_ip, m);
server->shutdown();
```

RServer

The RServer (rendezvous server) is a subclass of Server. There are no public methods of RServer in addition to Server's public methods.

Use case: Start rendezvous server, run the server until timeout, shutdown

```
String* server_ip = new String("127.0.0.1");
Server* server = new Server(server_ip);
server->run_server(60);
server->shutdown();
```

Node

The Node is a subclass of Server. There are three public methods of Node in addition to Server's public methods.

1) `Node(const char* client_ip_address, const char* server_ip_address)`: Constructs the Node and initializes values.
2) `void connect_to_server()`: Opens a socket and connects to the server.
3) `void send_message_to_node(String* node_ip, Message* message)`: Sends a message to another node.
4) `void send_put_message_to_node(String* message, int index)`: Sends a Put Message to the node at the given index
5) `int get_num_other_nodes()`: Gets the number of connected nodes

Use case: Start a Node, connect to server, send message to server, send message to node

```
String* server_ip = new String("127.0.0.1");
String* node_ip = new String("127.0.0.2");
String* other_node_ip = new String("127.0.0.3");
Node* node = new Node(node_ip, server_ip);
node->connect_to_server();
node->run_server(10);
String* hi = new String("hi");
Message* m = new Put(node_ip, server_ip, hi);
node->send_message(server_ip, m);
m = new Put(node_ip, other_node_ip, hi);
node->send_message_to_node(other_node_ip, m);
node->shutdown();
```

Implementation

Most of the implementation is in the Server class. A Server is constructed and a socket is created for incoming connections. This socket binds to the Server's IP. When the `run_server(int timeout)` method is called, the server enters a while loop where it monitors all sockets. This loop terminates when there is no activity for the timeout amount of seconds. The first command in the loop creates a set of file descriptors that are the active sockets. The set of fds is then passed to the `select` syscall which returns when there is activity on one fd in the set.

Next, the file descriptors are each checked. If there is activity on the incoming connection socket, that means there is a new connection and a socket is created for the new connection. The ip of that client is set to a default value until the client registers. If there is activity on a client socket, the socket is read from. If `read` returns zero, the client has disconnected and that socket is closed. If `read` returns a positive value, that is how many bytes the incoming serialized message is. The serialized message is then read and deserialized.

The `decode_message_` method is called with the message after it has been deserialized. It is up to the subclasses of Server to implement this method with actions to take in response to specific messages. The Server also has the `send_message` method which sends a serialized message to the ip or fd given to the method.

When the `shutdown` method is called, the server closes all of its sockets and stops running.

The RServer class inherits from Server and contains the implementation for rendezvous server. It implements the `decode_message_` method to respond to the Register Message by sending out a Directory Message (list of client IPs) to all of its clients. A Directory Message is also sent out to all clients when a client disconnects.

The Node class inherits from Server as well. It implements the `decode_message_` method to respond to a Directory and a Kill Message. When the Node receives a Directory Message, it updates its directory field. The response to a Kill Message is to shutdown the Node.

The Node adds the functionality to open a socket to connect to a Server. The IP of the Server is given on construction. After the node is successfully connected to the Server, it sends a Registration Message to the Server with its IP.

A Node can send a message to another node through the `send_message_to_node` method. This method opens a socket, connects to the given IP, sends the given message to the other node using the socket, and then closes the socket.