Distributed Networking Implementation Report

TeamRP
February 28th, 2020

Introduction

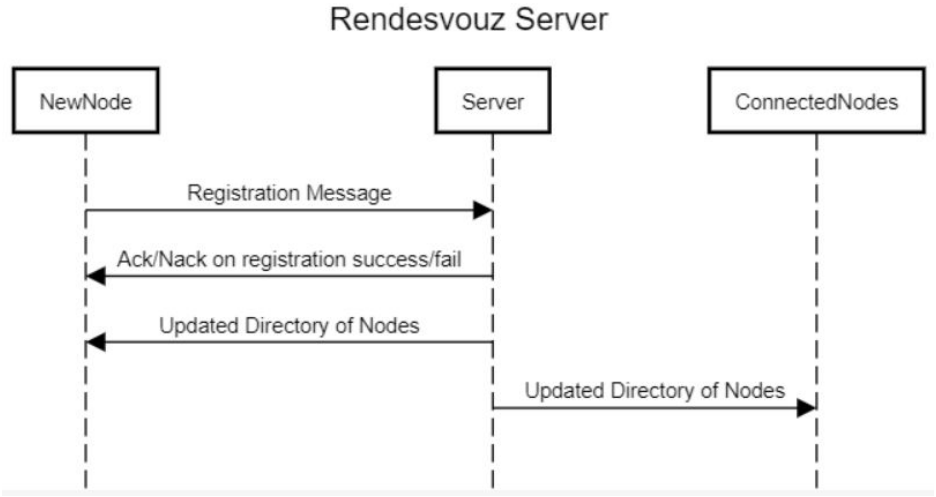
The purpose of this report is to document the overall design and implementation of a networking layer for our distributed application. The first major component of the system is one rendezvous server whose purpose is to record registration of new nodes and provide updates to all nodes when a new one has joined the network. The second major component of the system is a Node class which connects and listens for updates from the rendezvous server as well as communicates with other nodes on the network. The final socket-based implementation of the system enables peer to peer communication as well as orderly shutdown of the entire system.

Design and Implementation

Before discussing the implementation of the Node and Server classes, we must first describe the communication protocol used throughout the network. Each piece of data sent between nodes in our network is sent as a serialized version of a child class inheriting a base class called Message. The base Message class maintains a MsgKind enum representing the type of message, the index of the sender and target nodes, and a unique id field. The types of messages used within our current implementation of the network are Ack, Nack, Directory, Register, Kill, and Status. Each child class of Message contains fields unique to that type of Message as well as their corresponding serialize() and deserialize() methods. The serialize() method generates an unsigned char* representation of the class that called it. The deserialize() method reconstructs that object given an unsigned char*. These methods are important because it allows the transfer of structured objects over a socket connection without worry of data corruption or loss. Before sending any constructed message class over the network, it is first serialized into an unsigned char* data type so that it can be sent as a byte array over a socket connection. When data is received over a socket connection, the first thing to do is check the message type. All serialized message representations encode their type at the beginning of their unsigned char* representation, which means this can be checked using a helper method to determine what type of object to attempt deserialization with. This prevents any issues with attempting to deserialize an object into the wrong type.

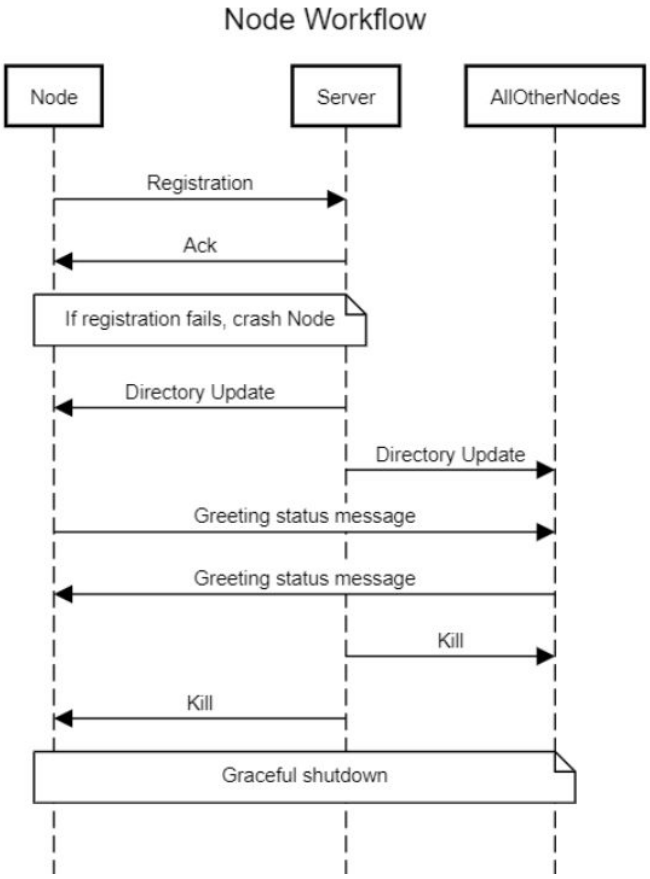
The first component of the system that was designed was the rendezvous server. The main functionality we had to support was registering and keeping active connections with Nodes who connected to our server so that we could send out directory updates anytime a new Node registered. This means that the properties that a Server object needs to maintain are its IP and

port configuration, a master listening socket, and a list of active Node sockets for updating nodes as well as a directory of its connected Nodes ip and port configurations. In order to implement this, we decided to make use of one central event loop in our code that manages the workflow for registration and node updates. A Server object is initialized with a given IP, port, and maximum node count for the new network. On calling the serve() method, the primary socket listening for new connections is initialized to the provided port. The server then enters a loop listening for new activity on any of its connected nodes as well as the main listening socket. When a new message is received, it is read into a buffer for temporary data storage and then passed to the handleMessage() method with the file descriptor and data. The purpose of this method is to determine its type and behave accordingly. In the case of a new registration message, the data is deserialized into a Register class and the contained IP and port are added to the server's Directory of nodes if it does not already exist. On successful update of the server's directory, an Ack message is sent back to the registering Node to notify them of successful registration. If the server was unable to register the Node, a Nack is sent instead. The Server's next task is to update each of its connected Nodes with the new Directory object. This is done by serializing the Directory, iterating through the maintained list of socket connections, and sending the data. When the shutdown() method is called, a Kill message is sent to all active socket connections and the Server is exited. The processing lifecycle for the rendezvous server can be seen below:



Each Node on a network is initialized with an IP and port to bind to, the server's IP and port. The requirement for each Node is that it must maintain an active connection with the server to listen for directory updates and also listen for connections and data sent directly from other Nodes on the network. Because of this clear separation of responsibility regarding the actions done by a Node, we decided to use a multithreaded approach when implementing our design. Listening to the server for updates is handled independently on one thread, while listening to active neighbor Node connections is handled on another. This removes the danger of running into issues with blocking related to standard socket methods in C. Each node maintains its IP

configuration, the server's IP configuration, a Directory of other nodes, one buffer each for both server messages and node messages, and a list of active neighbor node sockets. When the registerWithServer() method is called, a serialized Register message is sent with the corresponding IP and port contained inside. At the same time, a thread is launched to permanently listen to the server for updates. When a message is received from the server, it is passed to the handleIncoming() method, which directs the data to the right handler method depending on the message type. If an Ack is received from the server regarding the registration message, it means the Node was able to register successfully. If a Nack is received, the Node exits as the server already has a connected client with the same IP and port. After successfully registering, another thread is launched for listening to activity on the socket from other Nodes in the network. In our current implementation, whenever an updated Directory is received from the server, each Node opens connections with all other nodes and sends a greeting message with a type of Status. The overall lifecycle for any given Node is as shown:



When a Node receives a Kill message from the server, it closes all of its active connections with neighbors as well as the server.

Testing

In order to demonstrate a basic working network, the associated test.cpp file in the part1/test directory contains a test made up of a network of seven nodes and one server. Seven new threads are launched after initializing the server and each one waits 1 second and creates a new Node on the network. After each node joins the network, every connected node greets every other connected node.

Conclusion

Our current rendezvous server and Node network implementation supports all critical functions required to build the distributed application for our project. The robust serialization/deserialization methods utilized throughout our Message implementations allow us to send and rebuild structured data on both the client and server side which eliminates the possibility of corrupt or inconsistent data.