Neil Resnik & Yifei Wang
Network Memo

Introduction

This is a simple implementation of a network that allows clients to register with a central server, and then receive a list from the server of registered clients and connect/communicate with all other clients. This is achieved using threads to allow a single client or server entity to have multiple sockets connected to other entities, and also allow it to accept more connections at the same time. Messages are encoded using a Type-Length-Value Message scheme to allow for variable length messages and easier parsing. Since the sockets use TCP, there is no need for our own checks on data reception or data corruption (we instead rely on the Linux implementation of TCP).

Design & Implementation

Both the server and the client are similar in functionality. To account for this, they both descend from an abstract NetPort class. They both bind to a specific address, and listen on that address for a new connection. On receiving a new connection, they launch an abstract Connection object, which is a subclass of Thread. They each have their own concrete implementations depending on the use. This handes the new connection in a separate thread. To avoid issues with dead connections or deadlocks all sockets are non-blocking. In the same vein, all of the infinite loops require a time point to be reset so that no more than a minute passes without any connections (for the NetPort derivatives) or, in this implementation, any messages received. The current design demonstrates functionality using this and 3 separate connection implementations. The server uses a server_connection object to handle clients. This allows clients to register or deregister with the server freely, and on a change in the registered clients, it passes the list of registered clients to the connected clients. The clients can directly connect to any of the received clients. These are handled in the same way, except using a Client-To-Client Connection (CtCConnection). This allows direct communication between clients, and is asynchronous and non-blocking allowing for other work to be done. As a proof of concept, for now my clients simply greet each other before shutting down. After the minute passes without any new connections, my server also shuts down. As you may notice, the clients deregister from the server before they shut down. This keeps the server up to date, and allows it to clean-up the unneeded threads in a timely manner.

Extension

This means it is also extensible - as the basic communication is already established extension is accomplished through overriding the correct methods and providing more functionality. One may also need to define and implement the handling of other, specialized, message types. The main places one will need to override are, in NetPort derivatives, new_connection() which launches the custom connection implementation. If one also wants to do other things in the main event loop, the _main_loop_work() function is provided to allow one to extend that functionality. The _initial() function allows one to do one time start-up tasks before

entering the main event loop. The _on_new_connection method allows one to do something when something connects.

The same idea is accomplished in the Connection sub-classes, though they allow one to override the run method for the thread. This allows one flexibility in implementing their own event loop for active connections. The Connection class simply provides implementations of a lot of needed functionality - namely sending/receiving packets. It also provides an overridable/default implementation that parses packets. The idea is that this is overridden and specific types are handled, but otherwise the sub-class can delegate to the super-class. The current layout allows a user to be very flexible in where they go with the design, and most of the boilerplate code needed to implement a Client-Server protocol is already implemented. This allows a user to focus on their own code, rather than implementing the boilerplate socket code.