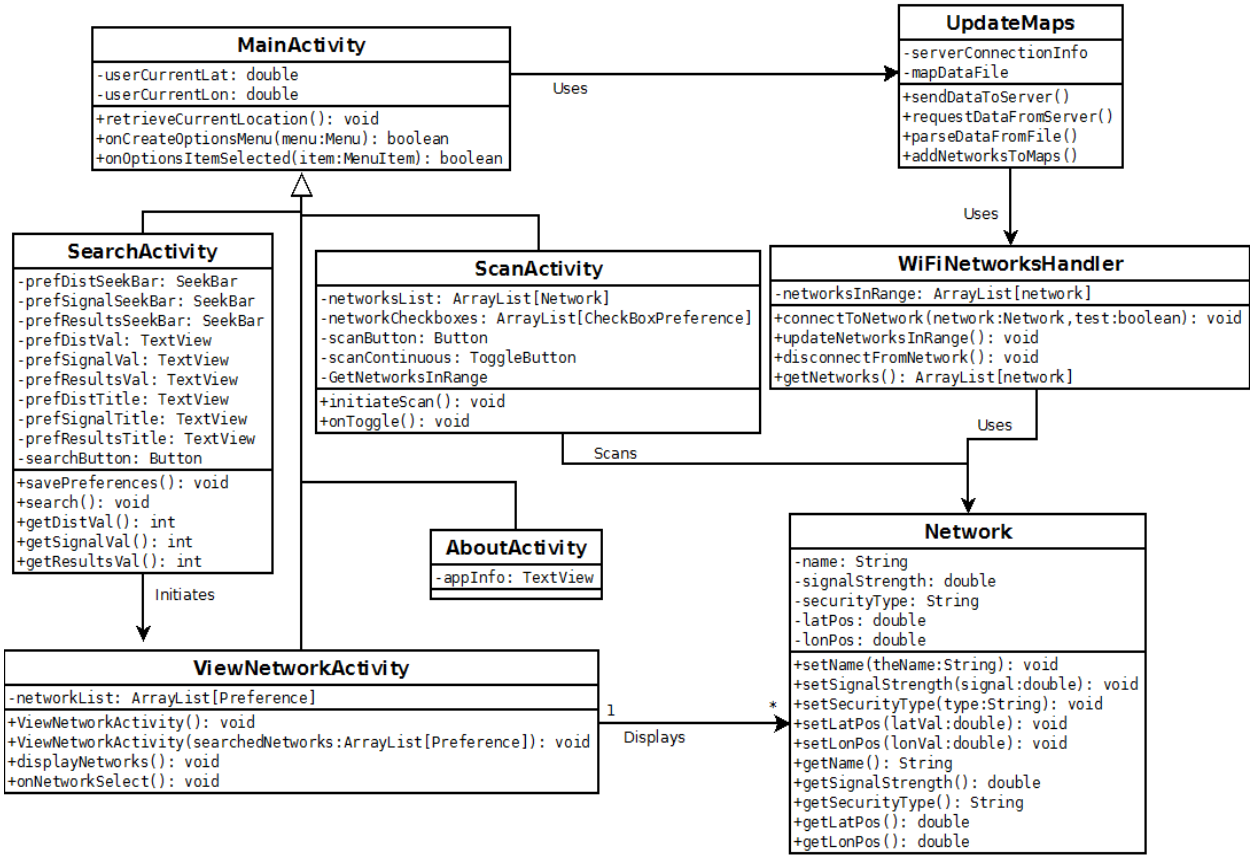
Group: Mighty Group G, LLC
Members: Michael Catanzaro, Miguel Guzman, Katie Isbell, Ryan Schmitt, and Ethan Wells
Due Date: Tuesday, October 29, 2013

Objective

The goal for this assignment was to create class diagrams to describe our wardriving application and include descriptions along with each of these class diagrams. Two class diagrams were used for our project since we have two separate implementations: the implementation of the android application and the implementation of the server. Below this, we also include information on how we obtained information from our users and how this has impacted our design.

GUI Class Diagram



GUI Class Diagram Description

The **MainActivity** class is the placeholder for our map. Its methods are as follows:

- currentLat - Stores the latitude value of the user's current location
- currentLon - Stores the longitude value of the user's current location

Its methods are as follows:

- retrieveCurrentLocation() - This method calculates the user's approximated current location using the GPS radio on the phone. It stores the latitude and longitude values in the userCurrentLat and userCurrentLon variables
- onCreateOptionsMenu(Menu menu) - This method creates our menubar which will create a way for users to navigate to each of the pages in the application.
- onOptionsItemSelected(MenuItem item) - This contains a switch case statement which opens a new page in the application based on the menu item selected by the user.

The SearchActivity, ScanActivity, ViewNetworkActivity, and AboutActivity classes display our main pages of the application. They will inherit the functionality of the MainActivity class using the keyword "extends", which will allow these pages to display the same menubar.

The **Network** class describes one of the networks within our database. The fields for this class are as follows:

- name- A String representing the name of the network
- signalStrength- A double representing the signal strength of the network
- securityType - A String value describing the security type of the signal
- latPos - The latitude value of the network
- lonPos- The longitude value of the network

The methods for this class simply include the accessor/mutator methods for obtaining/changing our variables.

- setName(String theName) - Set the name of the network
- setSignalStrength(double signal) - A precise value representing the signal strength
- setSecurityType(String type) - Set the security type
- setLatPos(double latVal) - Set the latitude value of the network
- setLonPos(double lonVal) - Set the longitude value of the network
- getName() - returns name
- getSignalStrength() - returns signalStrength
- getSecurityType() - returns securityType
- getLatPos() - returns latPos
- getLonPos() - returns lonPos

The **SearchActivity** class allows users to search for networks based on preferences they will set. The fields for this class are as follows:

- prefDistSeekBar- A seekbar whose value represents the maximum distance between the user's current location and the networks that will be displayed
- prefSignalSeekBar - The seekbar whose value represents the minimum signal strength for the networks that are returned from a search
- prefResultsSeekBar - The seekbar whose value represents the maximum number of networks that will be returned from a search
- prefDistVal - A textview located under the distance seekbar used to display the value of the distance seekbar
- preSignalVal- A textview located under the signal seekbar used to display the value of the signal seekbar
- preResultsVal- A textview located under the results seekbar used to display the value of the results seekbar
- prefDistTitle- A textview displaying the title of the distance seekbar
- prefSignalTitle- A textview displaying the title title of the signal strength seekbar
- prefResultsTitle- A textview displaying the title of the results seekbar
- searchButton - The button that will conduct a search for networks

The methods for this class are as follows:

- savePreferences() - This method saves the preferences set by the user. It is called when the search button is pressed.
- search()- This is also called when the search button is pressed. It conducts a search for the networks based on the preferences set by the user and opens a new page listing the networks.
- getDistVal() - returns the value of the maximum distance set by the user
- getSignalVal() - returns the value of the minimum signal strength set by the user
- getResultsVal() - returns the value of the results preference set by the user

The **ScanActivity** class allows users to scan for specific networks within range or run a continuous scan on all networks within range. The fields are as follows:

- networksList - An ArrayList holding all the networks within range
- networkCheckboxes - An ArrayList holding the checkboxes for the networks in the networksList
- scanButton - The button used to initiate the scan
- scanContinuous- The ToggleButton for allowing/unallowing a continuous scan in the background
- GetNetworksInRange - Class instance that handles getting the list of networks in range

and storing these networks in the networksList vector

The methods for this class include the following:

- initiateScan() - Scan for the networks that have been selected by the user. This method is called when the user presses the scan button. It will also be called every few seconds if the “scanContinous” toggle is high.
- onToggle() - Event handler that triggers when the user switches the “scanContinous” toggle high. Sets all of the checkboxes in networkCheckboxes enabled to false, disallowing the user to check these boxes. It then runs the initiateScan() method every few seconds until “scanContinous” is toggled low.

The **WiFiNetworksHandler** class is a helper class that takes care of all communication with the WiFi radio in the phone and does such things as returning an ArrayList of networks in range, connecting to a network, performing network connectivity to Internet tests, etc. Its fields are as follows:

- networksInRange - An ArrayList containing all of the public WiFi networks in range of the radio.

The methods are as follows:

- connectToNetwork(Network network, boolean test) - a method that will attempt to connect to the network that is passed in, and a passed in boolean will determine whether it will perform connection tests on it and disconnect.
- updateNetworksInRange() - a method that updates the ArrayList of networks with current information.
- disconnectFromNetwork() - a method that simply disconnects the phone from any network.
- getNetworks() - an accessor method for the private data ArrayList of networks.

The **ViewNetworkActivity** class opens a new page and displays networks in a list. Its fields are as follows:

- networkList - An ArrayList of Preferences, each representing a network.

The methods for this class include the following:

- ViewNetworkActivity() - This method is called after the user selects “ViewAllNetworks” menu item. It initializes networkList with all of the networks in our database. It then calls the displayNetworks() method to display these networks in a list.
- ViewNetworkActivity(ArrayList[Preference] searchedNetworks) - This method is called after the user presses the button to search for networks on the SearchActivity page. It

initializes networkList with the preferences passed through the parameter, and then calls the displayNetworks() method.
- displayNetworks() - This reads through the networkList and displays the networks on the screen.
- onNetworkSelect() - When the user selects a particular network, a new activity will be opened displaying that network on a map. The user can then use this map to get directions to that network's location.

The **AboutActivity** class serves as a page to display information about the application including the application name, application license, authors, and a list of external libraries that were used. Its fields are as follows:
- appInfo- a TextView listing authors, external libraries, and information about our application.

There are no methods within the AboutActivity class.

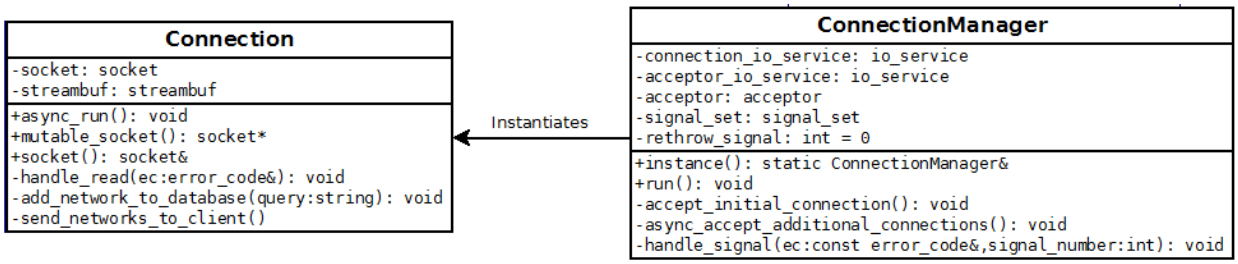
The **UpdateMaps** class handles communication with the server on updating the copy of data on the phone that is hosted on the server, as well as sending data to the server of the new networks the phone finds. It also handles parsing the data that the phone receives from the network and loads it into the Maps program on the phone. The fields are as follows:

- serverConnectionInfo - a data structure for holding the server connection information that will be used to connect to the server.
- mapDataFile - a path to the data file that the server returns containing all of the network information.

The methods are as follows:

- sendDataToServer() - a method that sends the data to the server. The data includes a command string like "ADD" or "VIEW". The "ADD" command includes the raw network data such as SSID, signal strength, and the latitude and longitude of the user.
- requestDataFromServer() - a method that is called once a day (default) or a user-defined interval that requests the new data from the server.
- parseDataFromFile() - a method that reads in the file retrieved from the server and parses it correctly to call the addNetworksToMaps() method when appropriate.
- addNetworksToMaps() - a method that is called when parsing the file and a network has to be added to a map. It is passed in the network details, like the SSID, signal strength, and the user's location.

Server Class Diagram



Server Class Diagram Description

The server is new-style daemon written in C++ that is socket-activated by systemd. It is started when a client attempts to connect, accepts new clients as necessary while running, exits when the last client has disconnected, and will start again only when a new client connects.

The **Connection** class corresponds to a single client connection. It is responsible for all interactions with its associated client after the connection has been accepted. Its member variables are as follows:

- socket - a TCP socket connected to the client
- streambuf - a buffer for holding data received from the client

The member functions are:

- async_run() - asynchronously reads queries from the client socket, calls handle_read() to handle the response
- mutable_socket() - accessor for the socket member variable
- socket() - const accessor for the socket member variable
- handle_read() - interprets the client's query and calls either add_network_to_database() or send_networks_to_client() as appropriate
- add_network_to_database() - adds a network to the database
- send_networks_to_client() - sends all networks in the database to the client

The **ConnectionManager** class is a singleton. It receives a listening socket from systemd when a client attempts to connect. It creates a Connection object to manage the initial connection. It continues accepting connections from the initial listening socket in case other clients attempt to connect before the first disconnects. ConnectionManager ensures the server terminates gracefully when all clients have disconnected. It has the following member variables:

- connection_io_service - Handles i/o operations for all Connection objects. The server exits when this io_service runs out of work (after the last connection has been closed).
- acceptor_io_service - Handles accepting additional connections and, secondarily, the signal_set used to catch POSIX signals. This io_service runs in a secondary thread.

- acceptor - socket listening for new connections
- signal_set - used to catch SIGTERM (to perform graceful shutdown) and SIGHUP (to be ignored as the server has no configuration files)
- rethrow_signal - int set by handle_signal() to indicate that a fatal signal has been caught. run() will raise this signal after the io_services stopped.

The member functions are:

- instance() - accessor for the singleton instance
- run() - called by main() to start the server. Calls accept_initial_connection. Creates a thread to run the acceptor_io_service and runs the io_service in that thread. Runs the connection_io_service in the main thread. Stops the acceptor_io_service when the connection_io_service has finished.
- accept_initial_connection() - create a new Connection object and synchronously accept the initial client connection with the Connection's socket
- async_accept_additional_connections() - create a new Connection object and asynchronously accept a new connection with the Connection's socket. Calls itself when complete.
- handle_signal() - Catches POSIX signals. Stops both io_services on receiving SIGTERM. Ignores SIGHUP.

User Requirements

Requirements were collected for our application mainly through team discussions and also by communicating with friends and family members outside of the group. Within our discussions, we brainstormed a list of requirements necessary to perform our end goals. For example, we determined it was necessary to use the WiFi radio for getting a list of networks for our database. Therefore, a requirement would be to access the raw WiFi network data on the phone. Our discussions with others outside of the group have helped us by allowing us to consider other possible functionality such as conducting a search based on the location (city) in addition to searching by distance, signal strength, and number of results. These discussions have also allowed us to realize other alternatives to our design such as using OpenLayers as opposed to Google Maps API.