**General Overview of Design**

All of the files in the network package are relevant to the network’s design with the exception of the 3 activities in the package. Those files were necessary for the demo but are likely useless for the actual app since much of the network’s code is meant to be run in the background.

The design for the network is as follows: when the application is launched, it first takes the user to a new activity I created called WelcomeActivity. That activity is currently incomplete but only contains a welcome message and a spinning loading screen along with some text informing the user of what it is doing. The WelcomeActivity starts 2 services: the TransmitterService and the RelayService. Services are akin to Android activities except that they run in the background and therefore have no UI. Once the WelcomeActivity has started those services in the background, it automatically moves on to the MainActivity, which has all of the previously created buttons for the user to view the map or manually enter in data.

TransmitterService spawns a thread on startup to try to connect to the network (this code hasn’t been included yet as of 8/24 morning; I’ll try to get it in this evening). Bear in mind that services automatically still run on the UI thread, so it is your responsibility to spawn new threads to ensure that long-running background tasks don’t crash the UI. The connectNetwork thread essentially listens for a periodic server UDP broadcast signal at port 9877 with a default timeout of 10 seconds. If it does not receive any broadcasts at that port, then it assumes that it is the first device to connect to the network. It initializes its deviceAddresses list (the list containing the IP addresses of all the devices currently connected on the network) to include itself and no one else, and initializes its rsaEncryptKeys list (the list containing the public RSA keys of all the devices on the network) to include just its own key (that it generates using the RSA.generateKeyPair() and then keyPair.getPublic() methods). If the device DOES receive a broadcast signal while listening at port 9877, then that means that there is at least 1 other device on the network. The message it should have received while listening on that port is the IP address of that other device sending out the broadcasts, so it then creates a TCP connection with that device on port 9879. The new device and the server device (which is the one that had been broadcasting its IP address) then engage in the following exchange.

1. New device sends the server device its IP address.
2. Server device confirms receiving the IP address.
3. New device converts its public key into a base64-encoded string and sends that to the server device.
4. Server device confirms receiving the public key.
5. Server device sends a comma-separated string representation of all the devices on the network, including itself and the new device’s IP address.
6. New device confirms receiving the string-based list.
7. Server device sends a comma-separated base64 representation of all the public keys of the devices on the network.
8. New device confirms receiving the string-based list.
9. New device converts the string addresses list and the string public keys list into ArrayLists and stores them in deviceAddresses and rsaEncryptKeys.
10. The connection is closed and the new device exits the connectToNetwork thread.

At this stage, the new device is aware of the necessary network information to begin transmitting and relaying. As a result, it starts a thread that is initialized with the Transmitter class, which implements the Runnable interface. It also starts a thread that is initialized with the UpdateNetwork class, which also implements the Runnable interface.

**UpdateNetwork and Gateway**

The connectToNetwork thread relied on some server device broadcasting its IP address to be connected to and then provide the new device with all of the IP addresses and public keys necessary to function. The design of the network infrastructure, at the moment, is such that the newest device on the network assumes that role as the server device to a single new device that tries to connect to the network. After servicing that device (i.e., updating the list of device addresses the list of public RSA keys and updating the rest of the network of the new device’s arrival), the device STOPS broadcasting its IP address and allows the device that’s now connected to replace it as the new “gateway” device. How is this implemented? The updateNetwork runnable, upon startup, immediately launches yet another thread initialized with the Gateway class, which implements the Runnable interface. The Gateway thread simply pings every single device on the network (in other words, broadcasts) with its IP address at 1 second intervals on port 9877. Meanwhile, the updateNetwork thread listens for incoming TCP connections at port 9879. As soon as the updateNetwork thread receives an incoming client connection at port 9879, it shuts down the Gateway thread because a new device has now joined the network. At port 9879, the same exchange as listed in the discussion about connectToNetwork earlier ensues, except this time it’s from the server’s perspective. Upon receiving the new device’s IP address and freshly-generated RSA public key, the updateNetwork thread adds those values to its copy of deviceAddresses and rsaEncryptKeys. Then, it converts each of those lists into a comma-separated String and sends them back to the new device. Upon confirmation of receipt from the new device, the connection is shut down. At this stage, the server device and the new device are synchronized, meaning they both have the same devices in their copies of deviceAddresses and the same keys in each of their copies of rsaEncryptKeys. However, all the other devices that may be connected to the network need to be updated with the new device’s information. As a result, updateNetwork then loops through all the addresses on deviceAddresses (skipping itself and the new device that just joined, of course) and connects to them at port 9881. It then sends the string versions of deviceAddresses and rsaEncryptKeys to each of them, waits for confirmation that they received them, and then closes the connection. This ensures that all other devices are now in-sync with the Gateway server device and the new device. Finally, updateNetwork goes into a while(true) loop that simply listens for new TCP connections at port 9881. In effect, it has completed its duty as the gateway device now that a new device has been connected to the network to replace it. As a result, it joins the rest of the devices, who simply wait for connections from the new Gateway device to send them updated copies of deviceAddresses and rsaEncryptKeys. Whenever they receive that, they simply convert the strings received into Lists and save the contents in deviceAddresses and rsaEncryptKeys.

**Transmitter**

The Transmitter thread checks to see whether the local copy of deviceAddresses has a length of at least 3 before beginning. If there are less than 3 devices on the network, this means that there aren’t enough devices to create an Onion route to some receiver. Every 10 seconds, the Transmitter updates its location object (which is initialized from within the TransmitterService) and then picks a random path through the network. If there are less than 5 devices on the network (including itself), then all of the other devices are used in the path (this means 1-2 relays and a receiver). If there are 5 or more devices on the network, then exactly 4 other devices are randomly selected from the network repo to constitute the rest of the path (3 relays and a receiver). The transmitEncrypt method creates the Onion encryption by generating a JSON at each layer that contains the encrypted message for the next device, the AES key needed to decrypt the message, and the IP address of the next device that the message needs to be forwarded to. The number of layers of encryption depends on the length of the path. After encrypting the message, the transmitter sends it to the first relay in the generated path at port 3899 and then repeats the process 10 seconds later.

**RelayService**

All of the above threads are created and maintained by the TransmitterService. When the application is shut down, the TransmitterService is responsible for gracefully terminating the running updateNetwork and Transmitter threads before shutting itself down. The RelayService also works in the background, but spawns a single RelayServer thread to listen at port 3899 for incoming connections. When it receives a connection, it spawns an instance of RelayConnection to handle the incoming message. When a message is read in from RelayConnection, it decrypts the fields of the message using its RSA private key and the AES key stored in the key field of the JSON. Upon decryption, it can determine whether the message needs to be forwarded or not depending on the value of the IP field. If it is 0, the device is the receiver. If not, a new connection to the IP address specified is created and the message is forwarded there. At the moment, nothing is done if a device is a receiver. However, this would likely be the optimal spot to integrate with ML code. There are a few different options of how this can be done, but an idea I have is to simply spawn a new thread to call the ML method that generates the probability distribution and wait on that thread. As soon as the probability distribution is generated, it needs to be converted into bytes (which can then be base64-encoded) and then encrypted using the same key that was used to encrypt the incoming message. This can be sent back using the same exact path that was used to send the message, and since each device waits for a response from the device that it forwarded its message to, all of the connections should still be alive and don’t need to be recreated. In this manner, the message will be re-encrypted, layer-by-layer, until it reaches the transmitter which has all of the keys to decrypt the layers and have the probability distribution. This is yet to be done, but can likely be accomplished relatively quickly after the other functionality is debugged.