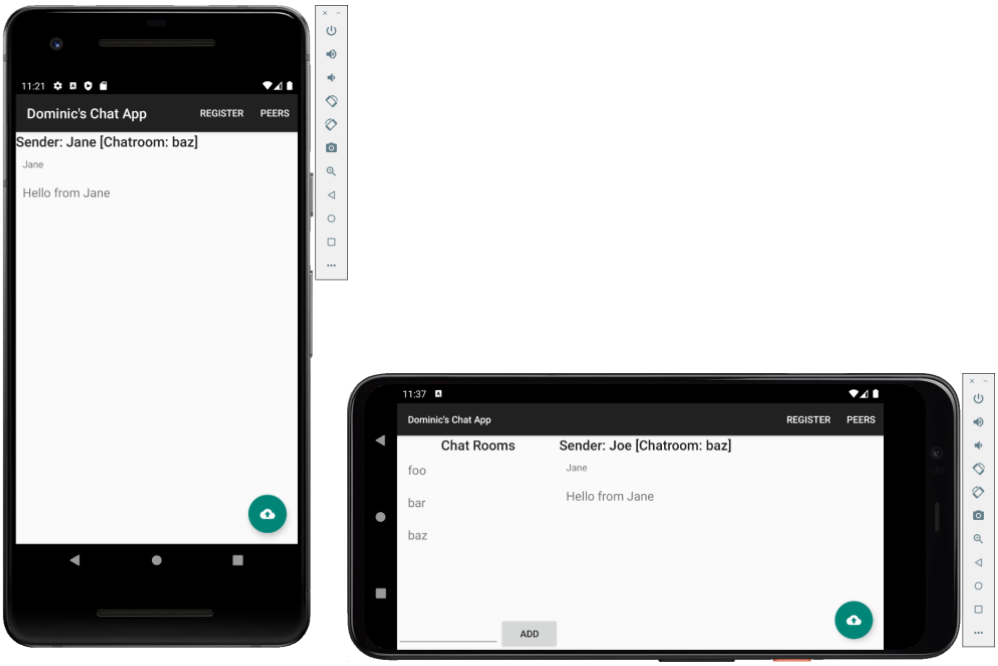
CS 522—Summer 2024
Mobile Systems and Applications
Assignment Eight—Chat App with Service

In this assignment, you will combine the client and server apps from the previous Chat app assignments into a single Chat app, that will allow bidirectional communication between different Android devices.



One of the problems with the previous server app is that it does a blocking message receive on the main UI thread when you press the "Next" button. This means that the whole app freezes up until a client sends a message. This is against best practice for Android programming, and we will fix this in the current assignment. The trick is to define the logic for waiting for incoming messages on a separate background thread. The Android component responsible for managing this thread is called a Service. It is like an Activity, but without a UI. In this assignment, you work with a single application. This has a foreground activity, `ChatActivity`, that displays messages received and also allows messages to be sent (so the app is now like a two-way radio). A background service, `ChatService`, handles the sending and receipt of messages in the background. This service should define a background thread, that waits for incoming messages without blocking on the main UI thread. It also defines a background thread for sending messages (use a handler thread for this). Define the service as one that the main chat activity binds to when it starts up.

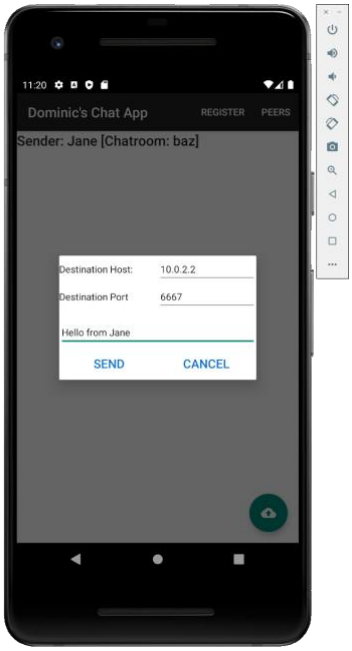
The background service needs to bind to a UDP port for sending and receiving messages. The chat service will provide functionality to the UI for sending messages. It provides a binder for allowing the UI to call into a service operation for sending a message. The interface provided to the UI client is called `IChatService`, and just provides a single send operation. For this assignment, assume that the UI and services are always in the

same process, so you can return the binder as a callback object that provides a reference to the sending service API for the main activity.

In addition, when a message is received by the background service, it needs to update the UI with the new message received. In the case where the new message is added to the message database, this is catered for automatically by the observer that the UI registers with the live data when the database is initially queried. Note that since the message will be received in a background thread (managed by the service), you are allowed to use synchronous persist operations to add data to the database. *You should also add any messages sent by this device to the message database, so the list of messages will include messages both sent and received.*

Since every message has a reference to the sender in the database, a record for the sender at the current device will need to be added to the database. Before the app can send any messages, the user must first "register" (available as an action from the task bar context menu). When a user registers, they choose a name, a peer record with this name is added to the database, and the name is saved in settings. The app will not send any messages until the user has registered.

There is no longer a "Next" button in this app, for synchronously requesting the next message. However, we still have a floating action button, for sending a message, as with the chat client in the previous assignment:



The sending of the message will again be performed on a background thread. For simplicity, we will use a single (bound) service to manage the threads for sending and receiving messages. The sending thread will use a `ResultReceiver` to report back to the activity when a message is sent; the activity should display a toast message when this is done. The `ResultReceiver` should **not** deliver its notification to the activity if the

activity is in the background (There is a five-second pause in the sending thread when sending a message, so you can demonstrate this for your app by going to the home screen as soon as you press SEND in the dialogue).

To test this app, you will now run two or more instances of the same app on different devices. *You should demonstrate communication in both directions between at least two devices.* Create two or more devices as before and run the Chat app on each of the two devices. Let's assume that you are doing UDP forwarding[1], so you have this code in your service:

```
chatConnection = factory.getUdpConnection(chatPort);
```

In the code, the Chat app always binds to UDP port 6666 for receiving messages. The `DatagramConnection` library always sends messages to the local host (identified by address `10.0.2.2` on the Android emulator). The user of the chat app identifies the device they want to send a message to by a UDP port number on the **host** machine (i.e., your laptop). You must set up forwarding from that port on the host machine to UDP port 6666 on the recipient machine. You can set up this redirection as you have done for earlier assignments. For example:

```
$ telnet localhost 5554
auth auth-token
redir add udp:6666:6666
quit
$ telnet localhost 5556
auth auth-token
redir add udp:6667:6666
quit
```

This forwards messages on (UDP) port 6666 on the host machine to port 6666 on the first device that was started (so the emulator allows telnet connections at TCP port 5554), and forwards messages on port 6667 on the host machine to port 6666 on the second device that was started (so the emulator allows telnet connections at TCP port 5556).

Both machines listen on local port 6666. With the definitions above, the first machine should send to port 6667 on the host machine, while the second machine should send to port 6666 on the host machine. The emulator forwards UDP packets sent to ports 6666 and 6667 to port 6666 on the appropriate device (`emulator-5554` and `emulator-5556`, respectively, in this example).

As in the previous assignment, your app should provide an activity for listing the peers with whom you have been in communication with, and another activity for listing details of a particular peer. You use the same database as in the previous assignment for this app, now used to save both messages received and messages sent.

---

[1] The `DatagramConnection` library also provides you with the options of UDP or SMS connection. UDP is the option that makes the most sense for this kind of application, if it works on the emulator.

Submitting Your Assignment

Once you have your code working, please follow these instructions for submitting your assignment:

1. Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory `Humphrey_Bogart`.
2. In that directory you should provide the Android Studio project for your Android app.
3. You should also provide an APK file for your compiled project.

In addition, record short mpeg videos of a demonstration of your assignment working. Make sure that your name appears at the beginning of the video. For example, put your name in the title of the app. *Do not provide private information such as your email or cwid in the video*. See the rubric for what the videos should demonstrate. You must also provide a completed rubric for the assignment.

Your solution should be uploaded via the Canvas classroom. Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have a single Android Studio project, for the app you have built. You should also provide a completed rubric in the root folder, as well as videos demonstrating the working of your assignment.