CS 522—Spring 2016
Mobile Systems and Applications
Assignment Five—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. You will provide two different solutions, each one involving a chat app that can both send and receive messages. They will differ in their implementation of the background service for sending a message.

All of your projects for this submission should target Marchmallow (Android 6.0, API 23).

Part 1: Single-Process Chat App

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, `ChatApp`, that displays messages received and also allows messages to be sent (so the app is now like a two-way radio). A background service, `ChatReceiverService`, handles the 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. Define the service as one that is explicitly started and stopped by the foreground activity. There is no longer a "Next" button in this app.

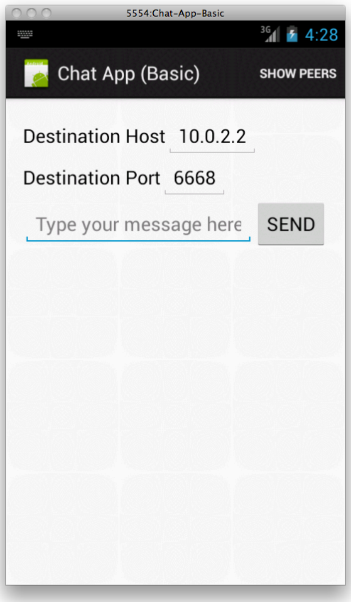
The background service needs to bind to a UDP port for receiving messages. Define another background service, called `ChatSenderService`, for sending messages. The same port should be used for sending messages as for receiving messages. Since the logic for sending a message is driven by the UI (pressing the "Send" button), the foreground activity needs to bind to the background chat service to send a message. Therefore the chat service 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 `IChatSendService`, and just provides a single send operation. For this part of the 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 content provider for received messages, this is catered for automatically by the update notification that is sent to the cursor (and hence the cursor loader and loader manager) when the content is updated (in your implementation of insert). However there may be more general updates on wishes to do on the UI, beyond just refreshing a list view when the cursor is updated. To give you some practice with this, we will have the background service broadcast when a new message is received (and the messages content provider is updated). The UI should register a receiver for this broadcast, and it should display a toast message when a broadcast is received from the background service that a message has been received.

To test this app, you will now run two instances of the same app on different devices. Create two devices as before, and define run configurations for running the Chat app on each of the two devices. We will assume that the Chat app always binds to UDP port 6666 for receiving messages. We extend the UI to specify both the destination host and port for a message that is being sent. The latter is unnecessary on "real" devices. However for testing on virtual devices, we will specify the development host loopback interface 10.0.2.2 as the destination host, as before. The destination port is then a UDP port on the development host, that should be redirected to port 6666 on the target Android device. You can set up this redirection again using the AVD console. For example, to get UDP port 6667 to redirect to the chat port on device emulator-5554, and to get UDP port 6668 to redirect to the chat port on device emulator-5556, type the following:

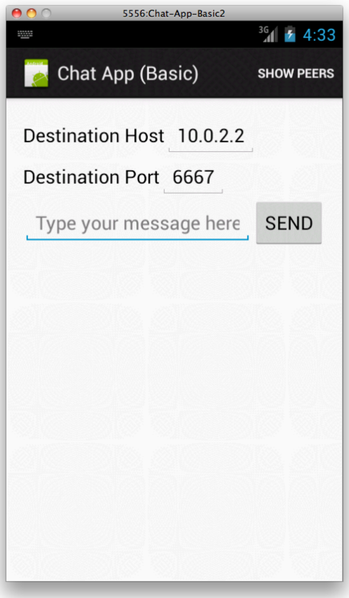
```
telnet 5554
redir add udp:6667:6666
quit
telnet 5556
redir add udp:6668:6666
quit
```

When you run these two devices, say with the first device named Server and the second device named Client, specify 6668 as the target port for the former and 6667 as the target port for the latter.



Destination Host 10.0.2.2

Destination Port 6668



Destination Host 10.0.2.2

Destination Port 6667

Unless you have defined a foreground activity for entering the name of the user, both app instances will call themselves "Client" (since that is the name defined in the resources). As before, use a PreferenceActivity to save a client name in the app. The name is prepended to every message that the client sends. On the server, the name is separated from each incoming request. As in the previous assignment, your should 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, including the messages received from that peer.

As with earlier assignments, you must ensure that a query on a content provider does not execute on the UI thread. Build on your solution from the previous assignment, where you used entity managers, loader managers and asynchronous content resolvers to encapsulate the details of data storage and retrieval, and asynchronous execution on background threads. Use services and background threads in this assignment to handle the network communication on background threads. To summarize, you should have two services: a bound service for handling the sending of messages, and an explicitly started and stopped service for handling receipt of messages.

Part 2: Chat App with Separated Service

In this part of the assignment, you will consider a variation on the previous part of the assignment, where it was assumed that the services and the UI were in the same process. Now we will play with the idea that they are in separate processes. You will not change the service for receiving messages. However the service for sending messages should rely on message channels for communicating with the main thread, using the `Messenger` class. The binder that is returned to the UI client should be a message channel, on which the client can send messages to be sent to other devices. Use the `HandlerThread` class, with your definition of a handler class, to ensure that the receipt of these messages in the service is processed on a background thread, not on the main thread.

To give you as much practice as possible with these concepts, also include an acknowledgment that is sent by the message sending service back to the UI client. Use the `ResultReceiver` class, with the wrapper class defined in the lecture materials, to send an acknowledgement to the client when a message has been received at the service and sent out over the network. Have the UI display a toast message when the acknowledgement is received.

Submitting Your Assignment

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

1. Export your two Android Studio projects to the file system: a single chat app that can operate as both client and server, one for single-process execution, the other allowing for the service to be in a separate process
2. 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.

3. In that directory you should provide the Android Studio projects for your Android apps.
4. You should also provide APK files for your compiled projects.
5. Also include in the directory a report of your submission. This report should be in PDF format. Do not provide a Word document.

In addition, record short flash, mpeg or Quicktime 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*.

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 Eclipse project, for the app you have built. You should also provide a report in the root folder, called README.pdf, that contains a report on your solution, as well as videos demonstrating the working of your assignment (unless the videos were uploaded to Google Drive instead). The report can just be a short summary of what you managed to accomplish (with screenshots and reference to the videos).