CS 522—Spring 2017
Mobile Systems and Applications
Assignment Seven—Cloud Chat Chap

In this assignment, you will complete a cloud-based chat app that you started on the previous assignment, where clients exchange chat messages through a Web service. You are given a simple chat server that apps will communicate with via HTTP. You should use the `HttpURLConnection` class to implement your Web service client, following the software architecture described in class. Set your minimum SDK version to Lollipop (Android 5.0, API 22) for your submission for this assignment. You should ensure that your app works on that platform, although you may develop on another device, e.g., your personal telephone.

Please note that the server provided for this assignment has a different API from that for the previous assignment.

The main user interface for your app presents a screen with a text box for entering a message to be sent, and a "send" button. The rest of the screen is a list view showing messages received and their senders. You should also have a settings screen where the server URI, and this chat instance's client name and UUID, can be viewed. As with the previous assignment, the chat name and client ID should be set during registration. There should also be, accessible from the options menu, a screen to see all other chat clients.

The interface to the cloud service, for the frontend activities, should be a *service helper* class. This is a POJO (plain old Java object) that is created by an activity upon its own creation, and encapsulates the logic for performing Web service calls. There are two operations that this helper class supports:

- Register with the cloud chat service.
- Post a message: In contrast with the previous assignment, this operation only adds the message, with an unassigned global sequence number, to the local database. A background alarm-driven operation synchronizes this internal database at regular intervals with the server, uploading new messages sent by this device and downloading peers and messages from other clients that have not yet been downloaded.

All of these operations are asynchronous, since you cannot block on the main thread.

Registration should generate a unique identifier (use the Java UUID class) to identify the app installation. Save this with the desired user name and the server URI in a shared preferences file when the user performs registration. Use notifications to inform the user if registration succeeds. Sending of chat messages is not enabled until registration succeeds. Your shared preferences should include a Boolean flag that is set once registration is complete. See the `Settings` class in the code with which you are provided.

Chat messages should be stored in a content provider for the app. In addition to the message text and primary key, store the name of the chat room in which the message was sent (default is "`_default`"), the timestamp of the message (a Java Date value, stored in the database as a long integer), a unique sequence number for that message (a long) set by the chat server (see below) and the identity of the sender (both their user name and a long identifier for their foreign key in the local databasex). We will be adding other metadata in future assignments. When a message is generated, it is added to the content provider with its message sequence number set to zero. The sequence number is finally set to a non-zero value when the message is eventually uploaded to the chat server; see the protocol below. The flag is always non-zero for messages downloaded from the chat server. Note that you cannot use this message sequence number as a primary key in your database, because its value is set by the chat server, but you will have to add local messages to the content provider immediately, without communicating with the server.

The content provider also stores a list of the other clients registered with the chat service. This list, and the list of chat messages, are periodically refreshed by synchronizing with the chat service. For simplicity, you can assume that a complete list of chat clients is downloaded on each request. However you should be more intelligent with downloading of new chat messages. Assuming that the chat service assigns a unique sequence number to each chat message it receives, the app can retrieve the sequence number of the most recent chat message that it has received from the content provider, and provide this to the chat server. The chat server will respond with all chat messages that it has received since that last chat message seen by the client. This synchronization should be done at the same time that the client is uploading messages to the server. So the protocol for synchronizing with the chat server, once the client is registered, is as follows:

1. The client uploads all messages stored in its content provider that have not yet been uploaded to the server. It also provides the sequence number of the last message it has received (along with its own UUID and client identifier to identify itself)
2. The server adds these messages to its own database, assigning each message a unique sequence number.
3. The server responds with a list of all of the registered clients, and a list of the messages that it has received since it last synchronized with the client.
4. The client deletes the messages it has just uploaded, and inserts the messages received from the chat server. It also replaces the list of chat clients with the list received from the server.

There is an obvious race condition: what happens if the user adds another chat message between uploading messages and downloading messages from server. That message will be lost when local unsent messages are deleted and re-inserted. We will avoid this by only deleting the first N unsent messages, where N is the number that of messages that are uploaded to the server. We still need to group the deletion and reinsertion into a single transaction, say using a bulk insert operation that sandwiches this deletion and reinsertion between a `beginTransaction/endTransaction` pair in the content provider. But this means building the list of insertions in memory, which belies the purpose of streaming data

with the server[1]. In the end, the size of the data here is small enough to warrant doing this, although in a production app we should make a special case for when an app is performing its first download of messages from the server, when it is downloading the entire server database. You should still stream the synchronization operation with the server, as an exercise to see how streaming should be done.

The service helper class should use an intent service (`RequestService`, subclassing `IntentService`) to submit request messages to the chat server. This ensures that communication with the chat server is done on a background thread. Single-threaded execution of requests will be sufficient, and greatly simplify things. There are three forms of request messages: `RegisterRequest`, `PostMessageRequest` and `SynchronizeRequest`. Two of these messages are sent by the UI using the service helper, as before. The third is triggered by a repeating alarm. Define three concrete subclasses of an abstract base class, `Request`, for each of these cases. The basic interface for the `Request` class is as follows:

```
public abstract class Request implements Parcelable {
   public Date timestamp;
   public Double latitude;
   public Double longitude;
   // App-specific HTTP request headers.
   public abstract Map<String,String> getRequestHeaders();
   // Chat service URI with parameters e.g. query string parameters.
   public String getRequestEntity() throws IOException;
   // Define your own Response class, including HTTP response code.
   public Response getResponse(HttpURLConnection connection,
                               JsonReader rd /* Null for streaming */);
}
```

The time and location information is used to record the last known location of the client at the server. For the registration and message posting requests, the service helper will create a request object and attach it to the intent that fires the request service (hence the necessity to implement the `Parcelable` interface).

The business logic for processing these requests in the app should be defined in a class called `RequestProcessor`. This is again a POJO class, created by the service class, which then invokes the business logic as represented by three methods, one for each form of request:

```
public class RequestProcessor {
   public Response perform(RegisterRequest request) { … }
   public Response perform(RegisterRequest request) { … }
   public Response perform(SynchronizeRequest request) { … }
}
```

[1] The fundamental problem here is that Android's content provider API has no support for bracketing transactions outside the content provider.

The request processor in turn will use an implementation class, `RestMethod`, that encapsulates the logic for performing Web service requests. See the lecture materials for examples of code that you can use for this class. This class should use `HttpURLConnection` for all Web service requests. You should not rely on any third-party libraries for managing your Web service requests. The public API for this class has the form:

```
public class RestMethod {
   ... // See lectures.
   public Response perform(RegisterRequest request) { … }
   public StreamingResponse perform(SynchronizeRequest request,
                                    StreamingOutput out) { … }
}
```

There are two forms of requests:

1. A *fixed-length request* sends the request data in HTTP request headers, as part of the URI (e.g. as query parameters) and/or in a JSON output entity body. Use this form of request for the registration request. You can supply all of the request parameters as (URL-encoded) query string parameters appended to the original service URI[2].
2. A *streaming request* is more appropriate for the case where there is potentially a lot of data to be uploaded or downloaded, so building a JSON string in memory is not advisable. This is the case for the synchronization request. Stream both the chat messages to be uploaded to the service, and the lists of clients and messages to be downloaded. Use the `JsonWriter` class to write the uploaded messages, and use `JsonReader` to read the streaming response from the server.

For streaming requests, the streaming is done in the request processor, **not** in the REST implementation (`RestMethod`). The latter just handles the mechanics of managing the network connection with the server. Therefore the input to the synchronization request has a second argument of this type:

```
public interface StreamingOutput {
    public void write(OutputStream out);
}
```

and the response from the synchronization request has this type:

```
public class StreamingResponse {
   public HttpURLConnection connection;
   public Response response;
}
```

Unlike the other `RestMethod` operations, that perform all necessary I/O on the network connection, and then close this connection before returning to the request processor, the streaming request operation (for synchronization) executes the request with HTTP

---

[2] For all requests except registration, the client id (returned from the registration request) should be provided as the last segment in the URI itself.

request headers and URI alone set, and then returns the open connection to the request processor. The latter can send data to the server by writing to the connection output stream (layering a `JsonWriter` stream over this) and receive data by reading from the connection input stream (layering a `JsonReader` over this). It is the responsibility of the request processor in this case to close the connection when done!

For synchronization, your service should transparently handle the case where communication is not currently possible with the server, either because the device is not currently connected to the network or because communication with the server times out. The client-side information that needs to be persisted is already in the content provider: Those messages that have a sequence number of zero, indicating that they have not yet been uploaded, and the maximum sequence number for the messages so far downloaded from the server. The latter can be obtained by querying the local database.

One way to upload messages is to rely on the user to do it manually, but this is obviously unsatisfactory. If the server is not available, the client will have to manually retry later. When you send an email, your email client should not rely on you to force resending if the network is not available when the email is initially sent. Therefore use an alarm, and the `AlarmManager` service, to periodically synchronize messages (and clients) with the server. The alarm handler should perform synchronization, provided registration has succeeded, in order to refresh the app state with state on the server.

Running the Server App

You are provided with a server app, written using Java JAX-RS (Jersey). You can use it just by executing the jar file. It takes two optional command line arguments: The host name and the HTTP port (default 8080). If you want to see the behavior of the server, without relying on your own code, you can use the curl command- line tool to send Web requests to the server. For example:

```
curl -X PUT -H 'X-Latitude: ...' ... \
   'http://localhost:8080/chat/client-id?chat-name=Joe'
```

This sends a POST request to the specified (registration) URI, and places the response headers in a file called `headers`. The contents of the header file will be of the form:

```
HTTP/1.1 200 OK
Content-Encoding: UTF-8
Content-Location: http://localhost:8080/chat/123e4567-e89b-12d3-a456-
426655440000
Content-Type: application/json
Date: Fri, 28 Feb 2014 14:57:25 GMT
Content-Length: 8
```

There is no output from the server for the registration request. The server contains a couple of debug commands that allow you to interrogate it. For example, you can test if a client is registered as follows:

```
curl -X GET \
   'http://localhost:8080/chat/client-id/'
```

The following command will synchronize messages with the server:

```
curl -X POST -H "Content-Type: application/json" \
   -d @messages.json –H 'X-Latitude: ...' ... \
   'http://localhost:8080/chat/client-id/sync?last-seq-num=0'
```

The query string parameter specifies the sequence number of the last message received by the client (The first message has a sequence number of 1). The file `messages.json` should contain messages to be uploaded, in JSON format. For example:

```
[
   {
      "chatroom" : "_default",
      "timestamp":...,"latitude":...,"longitude":...,
      "text" : "hello"
   },
   {
      "chatroom" : "_default",
      "timestamp":...,"latitude":...,"longitude":...,
      "text" : "is there anybody out there?"
   }
]
```

This will produce a JSON output of the form:

```
{"clients": [{username:"joe",timestamp:...,latitude:..., longitude:...}],
 "messages":
     [{"chatroom":"_default",timestamp:...,latitude:..., longitude:...,
       "seqnum":1,"sender":"joe","text":"hello"},
      {"chatroom":"_default",timestamp:...,latitude:..., longitude:...,
       "seqnum":2,"sender":"joe","text":"is there anybody out there?"}]}
```

The response includes a list of all clients registered with the service, and a list of messages uploaded to the service since the last time this client synchronized (including messages just uploaded by the client itself). Your app should delete the messages that have been uploaded, and then the downloaded messages should be inserted into the messages database. The peer database should be replaced with the client information downloaded from the server. Using usernames as foreign keys in the local database will ensure that existing messages remain correctly linked to their sender records. You can query for the messages that have been uploaded as follows:

```
curl -X GET \
   'http://localhost:8080/chat/client-id/messages'
```

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 app.
3. Also include in the directory a report of your submission. This report should be in PDF format. Do not provide a Word document.
4. In addition, record short flash, mpeg, avi 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*. Make sure that the output on the server is visible in the video (The server app will display messages as they are received).

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 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. The report should describe how to test your app (running on an AVD) against the server, as well as any parts of the spec that you were unable to complete. Your testing should demonstrate at least two devices registered at the server, messages being added, and messages from one device becoming visible at the other device (for both devices). Show the log of the server while messages are being exchanged.