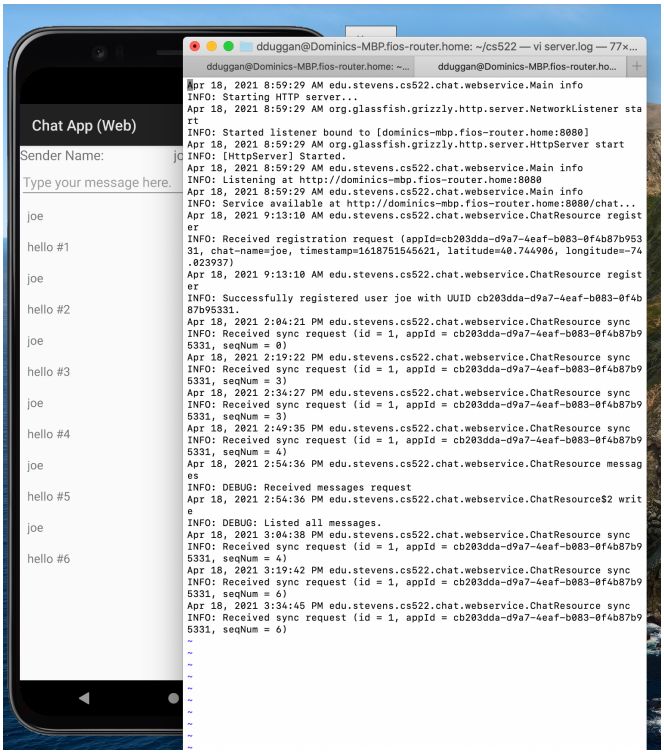
CS 522—Summer 2024
Mobile Systems and Applications
Assignment Ten—Chat App gRPC

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. This time, you are given a simple chat server that apps will communicate with via gRPC. You will use gRPC to implement your Web service client, following the software architecture described in class.



The main user interface for your app presents a screen with a list of messages posted by or received by this app, as well as the name under which you are registered at the server. The floating action button launches a dialog for sending a message from the current chatroom. The user chat name can only be set when registering with the server.

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 several operations that this helper class supports:

- Register with the cloud chat service.
- Post a message: Add a chat message to a request queue, to be uploaded to the server. *Unlike the previous assignment, this upload will not be done immediately.* Instead, the message will be added to the local chat message database, and this database periodically synchronized with the server in the background. This synchronization should also refresh the list of peers registered with the chat service.

- Start and stop message synchronization: For this assignment, these operations will enable and disable background synchronization of the chat database with the server. They are invoked by the lifecycle methods in the main activity.

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

When a message is generated, it is added to the database 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 database immediately, without communicating with the server.

The database also stores a list of the other clients registered with the chat service. This list, the list of chatrooms 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 database 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 the sequence number of the last message it has received (along with its own UUID and chat name to identify itself). It will also eventually upload all of its chatrooms to the server.
2. The server responds with a list of all of the registered clients, and a list of all the chatrooms, and a list of the messages that it has received since it last synchronized with the client. The client updates the list of chat clients and chatrooms with the lists received from the server, inserting new records and updating existing client records with client metadata[1]. Since we have been using sender chat names as foreign key references for senders of messages, you will be able to maintain the correct relationships between clients and messages in your database.
3. The client uploads all messages stored in its database that have not yet been uploaded to the server. The server adds these messages to its own database, assigning each message a unique sequence number.
4. The client updates each message it has just uploaded with its sequence number, when it is sent back by the server, and saves the new messages (from other clients) received from the chat server.

[1] So, all peers, chatrooms and messages downloaded from the server are “upserted” into the local database.

As before, the service helper class should use the WorkManager Lite API 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.

Request Processing

There are three forms of request messages: `RegisterRequest`, `PostMessageRequest` and `SynchronizeRequest`. Two of these messages are spawned by the UI using the service helper, as before. The third is periodically spawned by a background service. Define three concrete subclasses of an abstract base class, `Request`, for each of these cases. The basic interface for the `ChatServiceRequest` class is as before:

```
public abstract class ChatServiceRequest implements Parcelable {
   public UUID appId;  // installation id
   public String chatName;
   public long version;
   public double latitude;
   public double longitude;
   // Define your own Response class, including HTTP response code.
   public ChatServiceResponse getResponse();
}
```

The location information is used to record the last known location of the client at the server.

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

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

The request processor in turn will use an implementation class, `RestMethod`, that encapsulates the logic for performing Web service requests. This class uses `gRPC` to perform Web service requests: a blocking call for registration, and a streaming upload of local messages to, and download of message updates from, the server. The public API for this class has the form[2]:

```
public class RestMethod {
```

[2] We have no use for the case for `PostMessageRequest` in `RestMethod` in this assignment, because the message is logged in the database and asynchronously uploaded by the WorkManager Lite service. You can leave the case in the API for `PostMessageRequest`, for the sake of making this assignment upward-compatible with the previous assignment, even though `RequestProcessor` will never execute it.

```
    ... // See lectures.
    public ChatServiceResponse perform(RegisterRequest request) { … }
    public UploadObserver perform(SynchronizeRequest request,
                                  DownloadObserver downloadObsrvr) { … }
}
```

The registration request performs a blocking service call to the server, where the server URI for registration should have the form:

`grpc://`*`ip-address`*`:9090/`

For synchronization, the client provides a `DownloadObserver` callback to asynchronously process the updates that are downloaded from the server.

```
public interface DownloadObserver {
    public void onChatroom(Chatroom chatroom);
    public void onPeer(Peer peer);
    public void onMessage(Message message);
    public void onCompleted();
    public void onError(Throwable t);
}
```

The call to synchronize returns an `UploadObserver` that the client can use to upload its local data to the server:

```
public interface UploadObserver {
    public void onSync(long lastSequenceNumber, ...);
    public void onChatroom(Chatroom chatroom);
    public void onMessage(Message message);
    public void onCompleted();
    public void onError(Throwable t);
}
```

The request headers include this information identifying the client:

1. "`X-App-Id`", (a UUID identifying the installation).
2. "`X-Chat-Name`", (the user's chat name).

These headers are added by `RequestProcessor.process(`*`request`*`)`, which then uses the visitor pattern to dispatch to the logic for processing the request. GPS information is associated directly with the request messages, since a single request channel may be kept open and reused for several requests.

Synchronization with the Server

The background synchronization of messages with the server is done using the WorkManager Lite API, invoked in `ChatHelper.startMessageSync` (which is called in `ChatActivity.onStart`). Define a periodic request by instantiating this class:

```
public class PeriodicWorkRequest extends WorkRequest {
    public <T extends Worker> PeriodicWorkRequest (Class<T> _class,
                                                   Bundle data,
                                                   int interval);
}
```

The request specifies the class of the worker and the interval between executions of the periodic request (in minutes):

```
PeriodicWorkRequest syncRequest =
    new PeriodicWorkRequest(SynchronizeWorker.class, ...);

WorkManager.getInstance(context)
           .enqueuePeriodicUniqueWork(syncRequest);
```

The `SynchronizeWorker` object instantiates the request processor and calls it with a `SynchronizeRequest` message. The worker object is invoked with a period specified as one of the arguments to the `PeriodicWorkRequest` constructor. The reason that we use our own home-brewed WorkManager Lite API is that WorkManager requires this period to be no less than fifteen minutes!

Syncronization is disabled in `ChatHelper.stopMessageSync`, which should be invoked in `ChatActivity.onStop`, where it is important that the original synchronization request object be passed to the work manager operation for cancelling that operation:

```
WorkManager.getInstance(context)
           .cancelPeriodicUniqueWork(syncRequest);
```

As before, the logic for synchronizing with the server is defined in a worker object (whose class is provided in the `PeriodicWorkRequest` object enqueued with Work Manager Lite):

```
public class SynchronizeWorker extends Worker {
    ...
}
```

The worker just instantiates the request processor and invokes the logic for performing synchronization. For streaming requests, the streaming download is processed by the `UploadObserver` callback provided by the request processor to RestMethod, while RestMethod returns an `UploadObserver` object that is used by the request observer to push uploads to the server.

Within RestMethod, the Web service call that streams uploads and downloads has this signature:

```
StreamObserver<UploadItem> sync(StreamObserver<DownloadItem> callback);
```

This stub is generated from a gRPC protobuf description:

```
message UploadItem {
    oneof item {
        SyncRequest request = 1;
        Chatroom chatroom = 2;
        Message message = 3;
    }
}

message DownloadItem {
    oneof item {
        Chatroom chatroom = 1;
        Peer peer = 2;
        Message message = 3;
    }
}

service ChatService {

    rpc register (RegistrationRequest) returns (google.protobuf.Empty);

    rpc sync (stream UploadItem) returns (stream DownloadItem);

}
```

The `SyncRequest` message includes the sequence number for the last message downloaded from the server. This is maintained in a database (a table with a single record) and updated each time a message is downloaded, in case the network connection breaks during synchronization.

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 database: 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.

Running the Server App

You are provided with a server app, that provides the gRPC service. You can use it just by executing the jar file. It takes several optional command line arguments:

- `--host` *`host-name`*: The name of the host the server is running on (default is the result of executing `InetAddress.getLocalHost().getCanonicalHostName()`).
- `--grpc` *`port-number`*: The port the gRPC service is binding to (default 9090).

- `--bg true|false`: Whether the server is running in the background (it does not read a line of input to terminate, if running in the background; default false).
- `--log log-file-name`: Where to write the server log (default "server.log").

Besides the gRPC service, the server also runs the Web server from the last assignment, so you can query for the messages that have been uploaded by sending a GET command (e.g., with a web browser) to the URI:

`http://ip-address:8080/chat/messages`

As before, you can query for the log at the server by sending a GET command to the URI:

`http://ip-address:8080/chat/log`

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. In addition, record 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*.
4. For this assignment, you should demonstrate your app working against a running server in EC2 that you will be provided with. Make sure that this is defined as the base URI for the server when you are registering. Use the debugging commands to show the messages on the server, at the beginning and end of your demo videos.

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 videos demonstrating the working of your assignment. **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. Do this for both devices. Your final submission should include a demonstration of your app running against the chat server that will be provided running in the cloud.**