Internal implementation and working of the application

1. First of all, when the user selects the application, the MainActivity is launched. MainActivity consists of two main components. First one is the option to ask text query and get answer to that query in text format from other active users nearby. Second one is to start the background service for automatic fetching of direction of queries for a particular pair of source and destination using Google Maps' Directions API. What the user has to do is just enter appropriate source and destination as well as desired travel mode (Directions API provides 4 travel modes – driving, walking, bicycling and transit or public transport). Travel mode is set to driving by default. After giving appropriate input, we can ask for help.
2. After pressing the ASK FOR HELP button, the application will put these inputs (source, destination, travel mode) into a hashmap and use this to create a suitable message object to send using Bridgefy SDK. Bridgefy SDK has two types of message sending methods, first one is for broadcast messaging and second one is unicast messaging using unique id for every device. Broadcast message will be received by every online device within the range. Unicast message shall only be received by the device (if its online) for which it is intended (if unique device id matches).
3. Now whenever a user first launches the application, the app asks for required permissions. Upon allowance of the permissions, the app starts the initialization of Bridgefy SDK. If initialization is successful, there will be no error toast message displayed. After successful initialization, Bridefy is started using its start method and passing two listener callbacks as parameters to this method. The first one is message listener callback which listens for any incoming messages (broadcast or unicast) and second one is state listener callback which will let us know every time a successful connection has been established with a nearby Bridgefy device. It will also notify when a device has moved out of range or has disconnected for another reason.
4. When a device receives a query message (if the background service is running), it shows a notification containing the query. On clicking the notification, the app launches the HelperActivity, which collects required direction info using Directions API if the device is online and device parameters are satisfactory. If any of them is not satisfied, the user is instructed to enter his/her answer in text box (EditText) provided in the activity. After that, if the user hits the send help button, the application wraps all these information into a hashmap and converts it into a Bridgefy message object. Then, unique device id, extracted from the received from query, is used to set the id for the recipient of answer or help. Now, this message object will be sent to this particular device (if its online) which initiated the query in the first place.
5. Assuming that the background service for sending queries and receiving answers is still on in the query initiator device, when any answer corresponding to any query generated by this particular device is received, the app shows a notification, which displays the query as well as the answer in Q & A format. If it is clicked, it launches an activity, which displays this information in a scrollable text view.
6. If a certain device wants to disconnect, the stop button in the notification for the background process must be pressed or Bluetooth can be disabled to indirectly stop the process. Both of these methods call Bridgefy.stop() method which shuts down the Bridgefy SDK in that device.

Related Code Snippets

The first step is to generate an API key at http://bridgefy.me.

App Requirements The Bridgefy SDK supports Android 5.0 (**API Level 21**) or higher and the following permission are required.

```
android.permission.BLUETOOTH
android.permission.BLUETOOTH_ADMIN
android.permission.INTERNET
```

Internet access is required during the first run in order to check for a valid license in our servers.

If we're targeting devices with Android 6.0 (**API Level 23**) or higher, either one of the following permissions is also required:

```
android.permission.ACCESS_FINE_LOCATION

android.permission.ACCESS_COARSE_LOCATION
```

Note.- Devices with Android 6.0 (API Level 23) or higher also need to have Location Services enabled.

Hardware requirements

This version is fine tuned for Bluetooth Low Energy (BLE) capable devices. While it is not required, it is preferable that the *BLE advertising mode* is also supported. The Bridgefy SDK will let us know during initialization if our devices support *BLE advertising* or not. At least one device should support advertising mode in order for connections and, thus, messaging functionality, to be successful.

Initial Setup

In order to include the Bridgefy SDK in our project, first add the Bridgefy maven repository in our app's gradle file. We also need to make sure that our module is compatible with Java 8 (https://developer.android.com/studio/write/java8-support.html):

```
android {
    ...
    repositories {
         ...
         maven {
             url "http://maven.bridgefy.com/artifactory/libs-release-local"
             artifactUrls = ["http://jcenter.bintray.com/"]
              }
```

```
        ...
        }
    ...
    compileOptions {
        targetCompatibility JavaVersion.VERSION_1_8
        sourceCompatibility JavaVersion.VERSION_1_8
    }
}
```

Then, add the dependency:

```
implementation 'com.bridgefy:android-sdk:1.1.+'
```

Initialize Bridgefy

The Bridgefy SDK needs only a call to the static **initialize()** method in order to create all required objects and to be ready to start operations.

The result of the initialization will be delivered asynchronously to the **RegistrationListener** object.

```
//Always use the Application context to avoid leaks
Bridgefy.initialize(getApplicationContext(), OUR_API_KEY, new RegistrationListener()
{
    @Override
    public void onRegistrationSuccessful(BridgefyClient bridgefyClient) {
        // Bridgefy is ready to start
        Bridgefy.start(messageListener, stateListener);
    }

    @Override
    public void onRegistrationFailed(int errorCode, String message) {
        // Something went wrong: handle error code, maybe print the message
        ...
});
```

Alternatively, we can remove the **API_KEY** parameter if we included one in our **AndroidManifest.xml** file.

```
<meta-data
        android:name  = "com.bridgefy.sdk.API_KEY"
        android:value = "..." />
```

This call requires an active Internet connection on the device in order to check the status of our Bridgefy license. As long as our license is valid, an Internet connection won't be needed again until the time comes to renew or update it.

The following error codes may be returned if something went wrong:

```
-66    registration failed (check specific reason in message)
-1     registration failed due to a communications error (e.g. no Internet available)
```

```
-2     registration failed due to a misconfiguration issue
-3     registration failed due to an expired or unauthorized license
```

A unique **userId** string is also generated locally for our convenience in order to identify the local device. This field is method accesible on the **BridgefyClient** object returned on the successful callback.

Starting Operations

Once the Bridgefy SDK has been correctly registered the **onRegistrationSuccessful** method is called and we are now ready to start the Bridgefy SDK. Use the following method to begin the process of nearby devices discovery as well as to advertise presence to other devices.

```
Bridgefy.start(messageListener, stateListener);
```

We can also provide a custom **Config** object to set additional options

```
Config.Builder builder = new Config.Builder();
    builder.setEnergyProfile(BFEnergyProfile.HIGH_PERFORMANCE);
    builder.setEncryption(false);

Bridgefy.start(messageListener, stateListener, builder.build());
```

At this point, the **StateListener** callback will let us know every time a successful connection has been established with a nearby Bridgefy device. It will also notify us when a device has moved out of range or has disconnected for another reason.

```
@Override
public void onDeviceConnected(Device device, Session session) {
    // Do something with the found device
    device.sendMessage(...);
}

@Override
public void onDeviceLost(Device device) {
    // Let our implementation know that a device is no longer available
    ...
}
```

Sending Messages and receiving Messages

In order to send Messages we will need to build a **Message** object which is basically a **HashMap** tied to a **UUID** represented as a string; this way, Bridgefy will know where to send it.

```
// Build a HashMap object
HashMap<String, Object> data = new HashMap<>();
data.put("foo","Hello world");
```

```
// Create a message with the HashMap and the recipient's id
Message message =new
Message.Builder().setContent(data).setReceiverId(device.getUserId()).build();

// Send the message to the specified recipient
Bridgefy.sendMessage(message);
```
We can send messages to other devices even if they haven't been reported as connected or in-range. The Bridgefy SDK will do the best effort to deliver the message to it's recipient through intermediate devices. Message content is secured through a 256-bit encryption which is managed seamlessly for us so we don't have to worry about other users tapping into our private message.

We can also send public messages which will be propagated to all nearby devices. Those are even easier to send:

```
// Send a Broadcast Message with just the HashMap as a parameter
Bridgefy.sendBroadcastMessage(data);
```
The MessageListener callback will inform us of new messages that we have received. Check the Javadoc documentation for the full list of method callbacks.

```
@Override
    public void onMessageReceived(Message message) {
    // Do something with the received message
    ...
}

@Override
    public void onBroadcastMessageReceived(Message message) {
    // Public message sent to all nearby devices
    ...
}
```
Note.- Occasionally, the Bridgefy SDK may produce a duplicated call on these methods some time after the message was first received. Depending on our implementation, we might want to prepare for these scenarios.

Stopping Bridgefy

Once we have put on a show, always make sure to stop the Bridgefy instance in order to free up device resources.

```
Bridgefy.stop();
```