Session Initiation Protocol

In this document

Key classes

- `SipManager`
- `SipProfile`
- `SipAudioCall`

Related samples

- SipDemo

Android provides an API that supports the Session Initiation Protocol (SIP). This lets you add SIP-based internet telephony features to your applications. Android includes a full SIP protocol stack and integrated call management services that let applications easily set up outgoing and incoming voice calls, without having to manage sessions, transport-level communication, or audio record or playback directly.

Here are examples of the types of applications that might use the SIP API:

- Video conferencing.
- Instant messaging.

Requirements and Limitations

Here are the requirements for developing a SIP application:

- You must have a mobile device that is running Android 2.3 or higher.
- SIP runs over a wireless data connection, so your device must have a data connection (with a mobile data service or Wi-Fi). This means that you can't test on AVD—you can only test on a physical device. For details, see Testing SIP Applications.
- Each participant in the application's communication session must have a SIP account. There are many different SIP providers that offer SIP accounts.

SIP API Classes and Interfaces

Here is a summary of the classes and one interface (`SipRegistrationListener`) that are included in the Android SIP API:

Class/Interface	Description
`SipAudioCall`	Handles an Internet audio call over SIP.

Class/Interface	Description
`SipAudioCall.Listener`	Listener for events relating to a SIP call, such as when a call is being received ("on ringing") or a call is outgoing ("on calling").
`SipErrorCode`	Defines error codes returned during SIP actions.
`SipManager`	Provides APIs for SIP tasks, such as initiating SIP connections, and provides access to related SIP services.
`SipProfile`	Defines a SIP profile, including a SIP account, domain and server information.
`SipProfile.Builder`	Helper class for creating a SipProfile.
`SipSession`	Represents a SIP session that is associated with a SIP dialog or a standalone transaction not within a dialog.
`SipSession.Listener`	Listener for events relating to a SIP session, such as when a session is being registered ("on registering") or a call is outgoing ("on calling").
`SipSession.State`	Defines SIP session states, such as "registering", "outgoing call", and "in call".
`SipRegistrationListener`	An interface that is a listener for SIP registration events.

Creating the Manifest

If you are developing an application that uses the SIP API, remember that the feature is supported only on Android 2.3 (API level 9) and higher versions of the platform. Also, among devices running Android 2.3 (API level 9) or higher, not all devices will offer SIP support.

To use SIP, add the following permissions to your application's manifest:

- `android.permission.USE_SIP`
- `android.permission.INTERNET`

To ensure that your application can only be installed on devices that are capable of supporting SIP, add the following to your application's manifest:

```
<uses-sdk android:minSdkVersion="9" />
```

This indicates that your application requires Android 2.3 or higher. For more information, see API Levels and the documentation for the `<uses-sdk>` element.

To control how your application is filtered from devices that do not support SIP (for example, on Google Play), add the following to your application's manifest:

```
<uses-feature android:name="android.hardware.sip.voip" />
```

This states that your application uses the SIP API. The declaration should include an `android:required` attribute that indicates whether you want the application to be filtered from devices that do not offer SIP support. Other `<uses-feature>` declarations may also be needed, depending on your implementation. For more information, see the documentation for the `<uses-feature>` element.

If your application is designed to receive calls, you must also define a receiver (`BroadcastReceiver` subclass) in the application's manifest:

```
<receiver android:name=".IncomingCallReceiver" android:label="Call Receiver" />
```

Here are excerpts from the **SipDemo** manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.android.sip">
  ...
     <receiver android:name=".IncomingCallReceiver" android:label="Call Receiver" />
  ...
  <uses-sdk android:minSdkVersion="9" />
  <uses-permission android:name="android.permission.USE_SIP" />
  <uses-permission android:name="android.permission.INTERNET" />
  ...
  <uses-feature android:name="android.hardware.sip.voip" android:required="true" />
  <uses-feature android:name="android.hardware.wifi" android:required="true" />
  <uses-feature android:name="android.hardware.microphone" android:required="true" />
</manifest>
```

Creating a SipManager

To use the SIP API, your application must create a `SipManager` object. The `SipManager` takes care of the following in your application:

- Initiating SIP sessions.
- Initiating and receiving calls.
- Registering and unregistering with a SIP provider.
- Verifying session connectivity.

You instantiate a new `SipManager` as follows:

```
public SipManager mSipManager = null;
...
if (mSipManager == null) {
    mSipManager = SipManager.newInstance(this);
}
```

Registering with a SIP Server

A typical Android SIP application involves one or more users, each of whom has a SIP account. In an Android SIP application, each SIP account is represented by a `SipProfile` object.

A `SipProfile` defines a SIP profile, including a SIP account, and domain and server information. The profile associated with the SIP account on the device running the application is called the *local profile*. The profile that the session is connected to is called the *peer profile*. When your SIP application logs into the SIP server with the local `SipProfile`, this effectively registers the device as the location to send SIP calls to for your SIP address.

This section shows how to create a `SipProfile`, register it with a SIP server, and track registration events.

You create a `SipProfile` object as follows:

```
public SipProfile mSipProfile = null;
...

SipProfile.Builder builder = new SipProfile.Builder(username, domain);
builder.setPassword(password);
mSipProfile = builder.build();
```

The following code excerpt opens the local profile for making calls and/or receiving generic SIP calls. The caller can make subsequent calls through `mSipManager.makeAudioCall`. This excerpt also sets the action `android.SipDemo.INCOMING_CALL`, which will be used by an intent filter when the device receives a call (see Setting up an intent filter to receive calls). This is the registration step:

```
Intent intent = new Intent();
intent.setAction("android.SipDemo.INCOMING_CALL");
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intent, Intent.FILL_IN_DATA);
mSipManager.open(mSipProfile, pendingIntent, null);
```

Finally, this code sets a `SipRegistrationListener` on the `SipManager`. This tracks whether the `SipProfile` was successfully registered with your SIP service provider:

```
mSipManager.setRegistrationListener(mSipProfile.getUriString(), new SipRegistrationListener() {

public void onRegistering(String localProfileUri) {
    updateStatus("Registering with SIP Server...");
}

public void onRegistrationDone(String localProfileUri, long expiryTime) {
    updateStatus("Ready");
}

public void onRegistrationFailed(String localProfileUri, int errorCode,
    String errorMessage) {
    updateStatus("Registration failed.  Please check settings.");
}
```

When your application is done using a profile, it should close it to free associated objects into memory and unregister the device from the server. For example:

```
public void closeLocalProfile() {
    if (mSipManager == null) {
       return;
    }
    try {
       if (mSipProfile != null) {
          mSipManager.close(mSipProfile.getUriString());
       }
     } catch (Exception ee) {
       Log.d("WalkieTalkieActivity/onDestroy", "Failed to close local profile.", ee);
     }
}
```

Making an Audio Call

To make an audio call, you must have the following in place:

- A `SipProfile` that is making the call (the "local profile"), and a valid SIP address to receive the call (the "peer profile").
- A `SipManager` object.

To make an audio call, you should set up a `SipAudioCall.Listener`. Much of the client's interaction with the SIP stack happens through listeners. In this snippet, you see how the `SipAudioCall.Listener` sets things up after the call is established:

```
SipAudioCall.Listener listener = new SipAudioCall.Listener() {

   @Override
   public void onCallEstablished(SipAudioCall call) {
      call.startAudio();
      call.setSpeakerMode(true);
      call.toggleMute();
         ...
   }

   @Override
   public void onCallEnded(SipAudioCall call) {
      // Do something.
   }
};
```

Once you've set up the `SipAudioCall.Listener`, you can make the call. The `SipManager` method `makeAudioCall` takes the following parameters:

- A local SIP profile (the caller).
- A peer SIP profile (the user being called).
- A `SipAudioCall.Listener` to listen to the call events from `SipAudioCall`. This can be `null`, but as shown above, the listener is used to set things up once the call is established.
- The timeout value, in seconds.

For example:

```
call = mSipManager.makeAudioCall(mSipProfile.getUriString(), sipAddress, listener, 30);
```

Receiving Calls

To receive calls, a SIP application must include a subclass of `BroadcastReceiver` that has the ability to respond to an intent indicating that there is an incoming call. Thus, you must do the following in your application:

- In `AndroidManifest.xml`, declare a `<receiver>`. In **SipDemo**, this is `<receiver android:name=".IncomingCallReceiver" android:label="Call Receiver" />`.
- Implement the receiver, which is a subclass of `BroadcastReceiver`. In **SipDemo**, this is `IncomingCallReceiver`.
- Initialize the local profile (`SipProfile`) with a pending intent that fires your receiver when someone calls the local profile.
- Set up an intent filter that filters by the action that represents an incoming call. In **SipDemo**, this action is `android.SipDemo.INCOMING_CALL`.

Subclassing BroadcastReceiver

To receive calls, your SIP application must subclass `BroadcastReceiver`. The Android system handles incoming SIP calls and broadcasts an "incoming call" intent (as defined by the application) when it receives a call. Here is the subclassed `BroadcastReceiver` code from the SipDemo sample.

```
/**
 * Listens for incoming SIP calls, intercepts and hands them off to WalkieTalkieActivity.
 */
public class IncomingCallReceiver extends BroadcastReceiver {
    /**
     * Processes the incoming call, answers it, and hands it over to the
     * WalkieTalkieActivity.
     * @param context The context under which the receiver is running.
     * @param intent The intent being received.
     */
    @Override
    public void onReceive(Context context, Intent intent) {
        SipAudioCall incomingCall = null;
        try {
            SipAudioCall.Listener listener = new SipAudioCall.Listener() {
                @Override
                public void onRinging(SipAudioCall call, SipProfile caller) {
                    try {
                        call.answerCall(30);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            };
            WalkieTalkieActivity wtActivity = (WalkieTalkieActivity) context;
            incomingCall = wtActivity.mSipManager.takeAudioCall(intent, listener);
            incomingCall.answerCall(30);
            incomingCall.startAudio();
            incomingCall.setSpeakerMode(true);
            if(incomingCall.isMuted()) {
                incomingCall.toggleMute();
            }
            wtActivity.call = incomingCall;
            wtActivity.updateStatus(incomingCall);
        } catch (Exception e) {
            if (incomingCall != null) {
                incomingCall.close();
            }
        }
    }
}
```

Setting up an intent filter to receive calls

When the SIP service receives a new call, it sends out an intent with the action string provided by the application. In SipDemo, this action string is `android.SipDemo.INCOMING_CALL`.

This code excerpt from **SipDemo** shows how the `SipProfile` object gets created with a pending intent based on the action string `android.SipDemo.INCOMING_CALL`. The `PendingIntent` object will perform a broadcast when the `SipProfile` receives a call:

```
public SipManager mSipManager = null;
public SipProfile mSipProfile = null;
...

Intent intent = new Intent();
intent.setAction("android.SipDemo.INCOMING_CALL");
PendingIntent pendingIntent = PendingIntent.getBroadcast(this, 0, intent, Intent.FILL_IN_DATA);
mSipManager.open(mSipProfile, pendingIntent, null);
```

The broadcast will be intercepted by the intent filter, which will then fire the receiver (`IncomingCallReceiver`). You can specify an intent filter in your application's manifest file, or do it in code as in the **SipDemo** sample application's `onCreate()` method of the application's `Activity`:

```
public class WalkieTalkieActivity extends Activity implements View.OnTouchListener {
...
    public IncomingCallReceiver callReceiver;
    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {

       IntentFilter filter = new IntentFilter();
       filter.addAction("android.SipDemo.INCOMING_CALL");
       callReceiver = new IncomingCallReceiver();
       this.registerReceiver(callReceiver, filter);
       ...
    }
    ...
}
```

Testing SIP Applications

To test SIP applications, you need the following:

- A mobile device that is running Android 2.3 or higher. SIP runs over wireless, so you must test on an actual device. Testing on AVD won't work.
- A SIP account. There are many different SIP providers that offer SIP accounts.
- If you are placing a call, it must also be to a valid SIP account.

To test a SIP application:

1. On your device, connect to wireless (**Settings > Wireless & networks > Wi-Fi > Wi-Fi settings**).
2. Set up your mobile device for testing, as described in Developing on a Device.
3. Run your application on your mobile device, as described in Developing on a Device.
4. If you are using Android Studio, you can view the application log output by opening the Event Log console (**View > Tool Windows > Event Log**).
5. Ensure your application is configured to launch Logcat automatically when it runs:
   a. Select **Run > Edit Configurations**.
   b. Select the **Miscellaneous** tab in the **Run/Debug Configurations** window.
   c. Under **Logcat**, select **Show logcat automatically** then select **OK**.