Accessory

Programming Guide

Version 2.1.11

Table of Contents

1. OVERVIEW 3

1.1. BASIC KNOWLEDGE 3
1.2. ARCHITECTURE 4
1.2.1. Accessory Eco-system Terminology 4
1.2.2. Functional Flow Between a Service Consumer and Provider 5
1.2.3. Accessory Peer Agent State Machine 6
1.3. CLASS DIAGRAM 8
1.4. SUPPORTED PLATFORMS 10
1.5. SUPPORTED FEATURES 10
1.6. COMPONENTS 10
1.7. IMPORTING LIBRARIES 11

2. HELLO ACCESSORY 12

2.1. HELLO ACCESSORY PROVIDER 12
2.2. HELLO ACCESSORY CONSUMER 14

3. USING THE SA CLASS 18

3.1. USING THE INITIALIZE() METHOD 18
3.2. HANDLING SSDKUNSUPPORTEDEXCEPTION 18
3.3. CHECKING THE AVAILABILITY OF ACCESSORY FEATURES 19

4. USING ACCESSORY 20

4.1. CONFIGURING YOUR APPLICATION 20
4.2. REGISTERING YOUR SERVICE PROVIDER OR SERVICE CONSUMER 20
4.2.1. Validating the Service Profile XML 22
4.3. FINDING A MATCHING ACCESSORY PEER AGENT AND INITIATING A SERVICE CONNECTION 25
4.4. HANDLING SERVICE CONNECTION REQUESTS 27
4.5. EXCHANGING DATA WITH ACCESSORY PEER AGENTS 29
4.6. DISCONNECTING AND ERROR HANDLING 30

5. GUIDE FOR PROGUARD 33

COPYRIGHT 34

1. Overview

Accessory allows you to develop applications on Samsung Smart Devices and Accessory Devices. You can connect Accessory Devices to Samsung Smart Devices without worrying about connectivity issues or network protocols.

You can use Accessory to:

- Advertise and discover Accessory Services.
- Set up and close Service Connections with one or more logical Service Channels.
- Support Service Connections using a range of connectivity options.
- Configure Accessory Service Profiles and roles for Accessory Peer Agents.

1.1. Basic Knowledge

The Accessory eco-system consists of one or more Samsung Smart Devices and Accessory Devices that support the Samsung Accessory Protocol:

- Smart Devices:

  Samsung smart phone and tablet devices.
  Later releases may include other devices, such as Samsung Smart TVs, cameras, and laptops.
  Compliant Smart Devices support the Samsung Accessory Protocol and usually include built-in support for popular Accessory Service Profiles.

- Accessory Devices:

  Auxiliary devices that connect to Smart Devices.
  Compliant Accessory Devices support the Samsung Accessory Protocol and can interact with compliant Smart Devices using a range of connectivity options.

The following figure shows the roles in the Accessory eco-system.

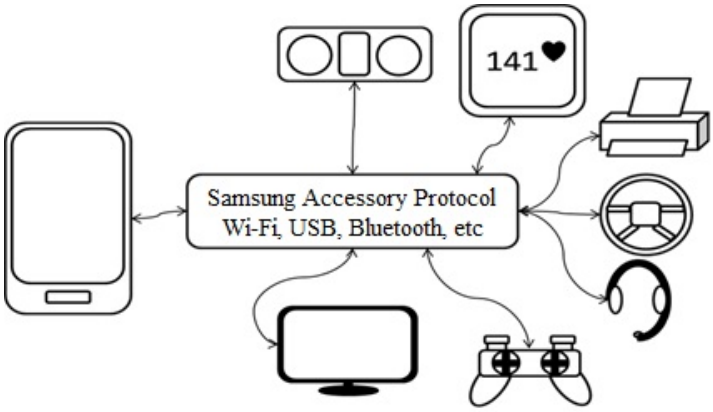

Figure 1: Accessory eco-system

Samsung Smart Devices can support one or more Accessory Services using a manager application with the Samsung Accessory Service Framework, for example, Samsung GEAR Manager. The Smart Devices and Accessory Devices described in this document have the Samsung Accessory Service Framework preloaded.

1.2. Architecture

The following figure shows the Accessory architecture.

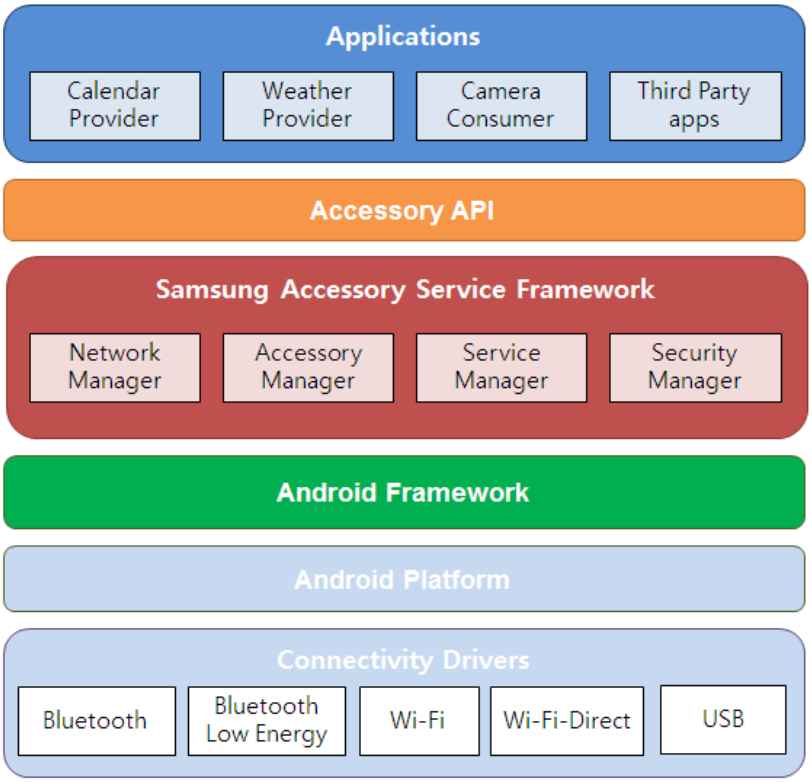

Figure 2: Accessory architecture

Application Level Entities (ALE), such as Calendar Provider and Camera Consumer, use Accessory as a facade. Accessory API communicates with the Samsung Accessory Service Framework pre-loaded on Samsung Smart Devices. The Samsung Accessory Service Framework is built on top of Android stacks of connectivity methods such as Wi-Fi, Bluetooth, and USB.

1.2.1. Accessory Eco-system Terminology

The following terms are used when describing the Accessory eco-system:

- **Accessory Service Profile**

  An Accessory Service Profile defines the roles of a Service Provider and Service Consumer, and specifies the formats for application-level protocol messages and message sequences between Service Consumers and Service Providers. The Notification Accessory Service Profile, for example, defines the JSON schemas for messages used to send and receive notifications between Samsung Smart Devices and compliant Accessory Devices. An Accessory Service Profile also defines message sequences between a notification Service Consumer and a notification Service Provider.

- **Service Provider**

  A Service Provider is an ALE with a role defined in the associated Accessory Service Profile specification. It accepts incoming Service Connections from Service Consumers and initiates

outgoing Service Connections to Service Consumers. A Service Provider registers with the Samsung Accessory Service Framework to advertise its services to Service Consumers on connected Accessory Devices. A notification Service Provider implemented on a Smart Device, for example, provides notifications from that Smart Device to interested Service Consumers on connected Accessory Devices.

- **Service Consumer**

  A Service Consumer is an ALE with a role defined in the associated Accessory Service Profile specification. It discovers a matching Service Provider using the Capability Exchange Protocol, initiates outgoing Service Connections with the matching Service Provider, and accepts Service Connection requests from Service Providers. A Service Consumer uses the information or service provided by the matching Service Provider. It has to register with the Samsung Accessory Service Framework. A notification Service Consumer implemented on an Accessory Device, for example, receives notification information from the notification Service Provider on a connected Smart Device.

- **Accessory Peer Agent**

  An Accessory Peer Agent is the main interface between the Samsung Accessory Service Framework and the ALE implementing a Service Provider or Service Consumer. The Samsung Accessory Service Framework views both Service Providers and Service Consumers as Accessory Peer Agents.

- **Service Connection**

  A Service Connection represents the dialog between a Service Consumer and a Service Provider. It includes one or more Service Channels for data exchange between a Service Consumer and Service Provider.

- **Service Channel**

  A Service Channel is a logical data channel between a Service Consumer and a Service Provider. The channel ID, data rate, priority, and delivery type distinguish Service Channels from each other. While a Service Connection is a multi-lane highway between a Service Consumer and a Service Provider, the Service Channel is an individual lane of that highway.

1.2.2. Functional Flow Between a Service Consumer and Provider

The Samsung Accessory Protocol supports multiple connectivity methods, such as Wi-Fi, Bluetooth classic, Bluetooth Low Energy (v4.0), and USB, while freeing you from connectivity-specific details. The Samsung Accessory Service Framework supports the discovery of features (services) and enables the establishment of Service Connections between ALEs for data exchange.

The following figure shows the functional flow in the Samsung Accessory Service Framework between a Service Consumer and a Service Provider. Peer Device 1 and 2 are either Samsung Smart Devices or Accessory Devices with applications acting as Service Providers or Service Consumers.

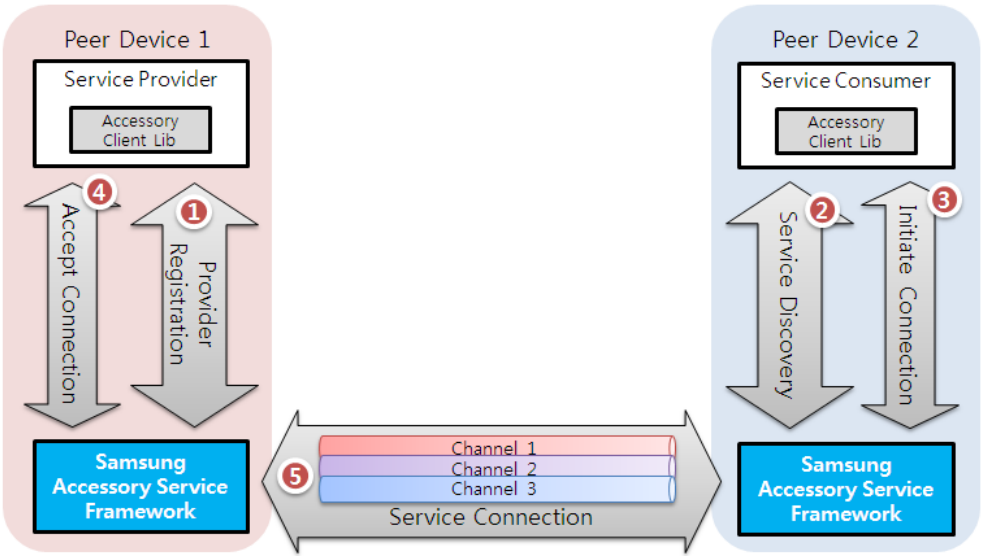

Figure 3: Functional flow between Service Provider and Service Consumer

The Service Provider and Service Consumer applications register their service capabilities with the Samsung Accessory Service Framework. The Samsung Accessory Service Framework advertises and exchanges the capabilities of the registered Service Providers and Service Consumers.

The Service Consumer looks for Service Providers of interest, and queries the Samsung Accessory Service Framework, which in turn queries the services offered by connected Accessory Devices.

The Service Consumer attempts to establish a Service Connection with the Service Provider.
A Service Provider can also try to establish Service Connections with Service Consumers.

The Service Provider decides to accept or reject the Service Connection request. If the Service Provider attempted to establish a connection, the Service Consumer decides to accept or reject the Service Connection request.

The Service Connection is established, creating all the Service Channels defined by the associated Accessory Service Profile. The Service Consumer and Service Provider use the established Service Connection to read and write data following the associated Accessory Service Profile specification on the Service Channels.

1.2.3. Accessory Peer Agent State Machine

Accessory Peer Agents, both Service Providers and Service Consumers, handle concurrent instances. A Service Provider can accept incoming Service Connections from multiple Service Consumers with the same Accessory Service Profile, e.g., the notification service. Similarly, a Service Consumer can accept incoming Service Connections from multiple Service Providers with the same Service Profile.

Every accepted Service Connection request results in the creation of a SASocket object, which represents the dialog between a Service Provider and a Service Consumer. The Samsung Accessory Service Framework establishes one or more Service Channels with the QoS parameters defined by the Accessory Service Profile. The SASocket object encapsulates these Service Channels.

The following figure shows the state machine of an Accessory Peer Agent with a remote Accessory Peer Agent. If there is more than one remote Accessory Peer Agent, the Accessory Peer Agent can have different

states with different remote Accessory Peer Agents. For example, some remote Accessory Peer Agents can be in a connected state, while others are in a registered (disconnected) state.

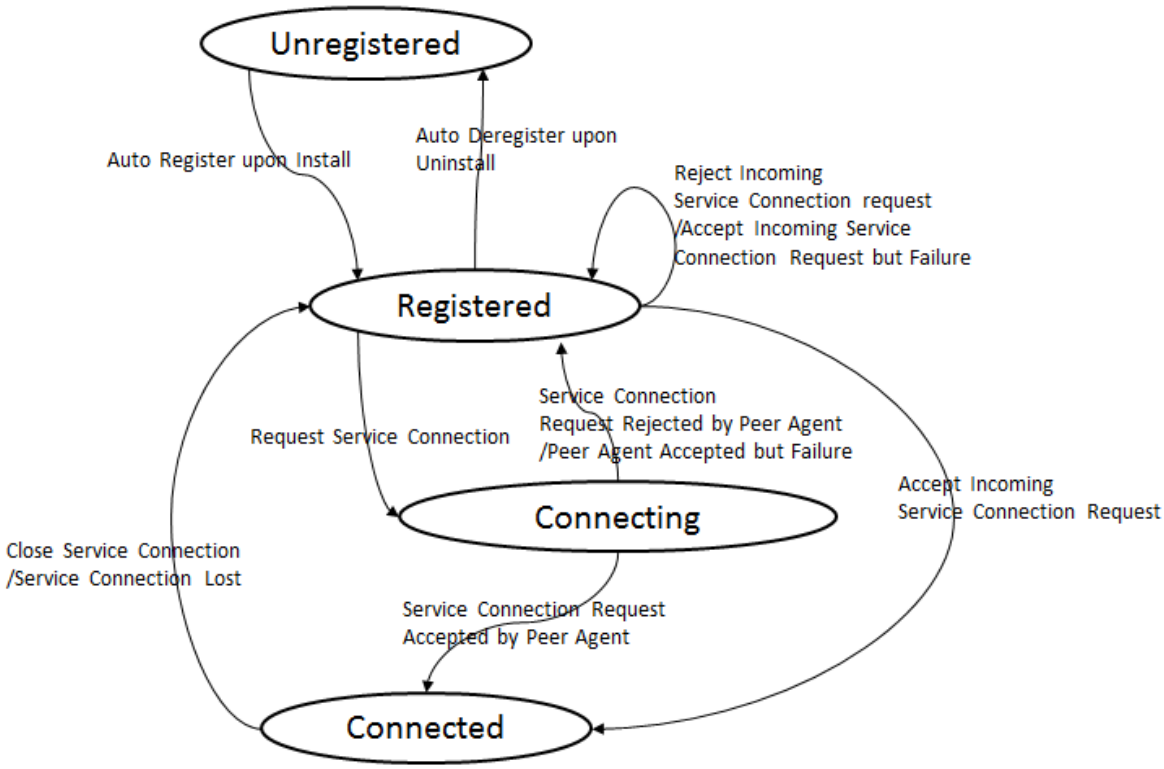

Figure 4: State machine of an Accessory Peer Agent

The figure illustrates the following states:

- A Service Provider or Service Consumer application automatically registers with the Samsung Accessory Service Framework upon installation, and enters a “registered” state. Similarly, the application automatically deregisters upon uninstallation and goes to an “unregistered” state.
- The Accessory Peer Agent enters a “connecting” state when it initiates an outgoing Service Connection with a matching remote Accessory Peer Agent with the same Accessory Service Profile and a complementary Provider/Consumer relationship.
- The Samsung Accessory Service Framework establishes a Service Connection if a remote Accessory Peer Agent accepts a Service Connection request. The Accessory Peer Agent enters a “connected” state on success. If the remote Accessory Peer Agent rejects a Service Connection request or if there is a failure, the Accessory Peer Agent goes back to the “registered” state.
- When a Service Connection request from a remote Accessory Peer Agent is received, the Service Provider or Service Consumer application is notified and the application accepts or rejects the incoming Service Connection request. If the application accepts the request and the Service Connection is successfully established, the Accessory Peer Agent enters the “connected” state. Otherwise, it remains in the “registered” state.

The following figure shows the sequence flow of the Accessory Peer Agent.

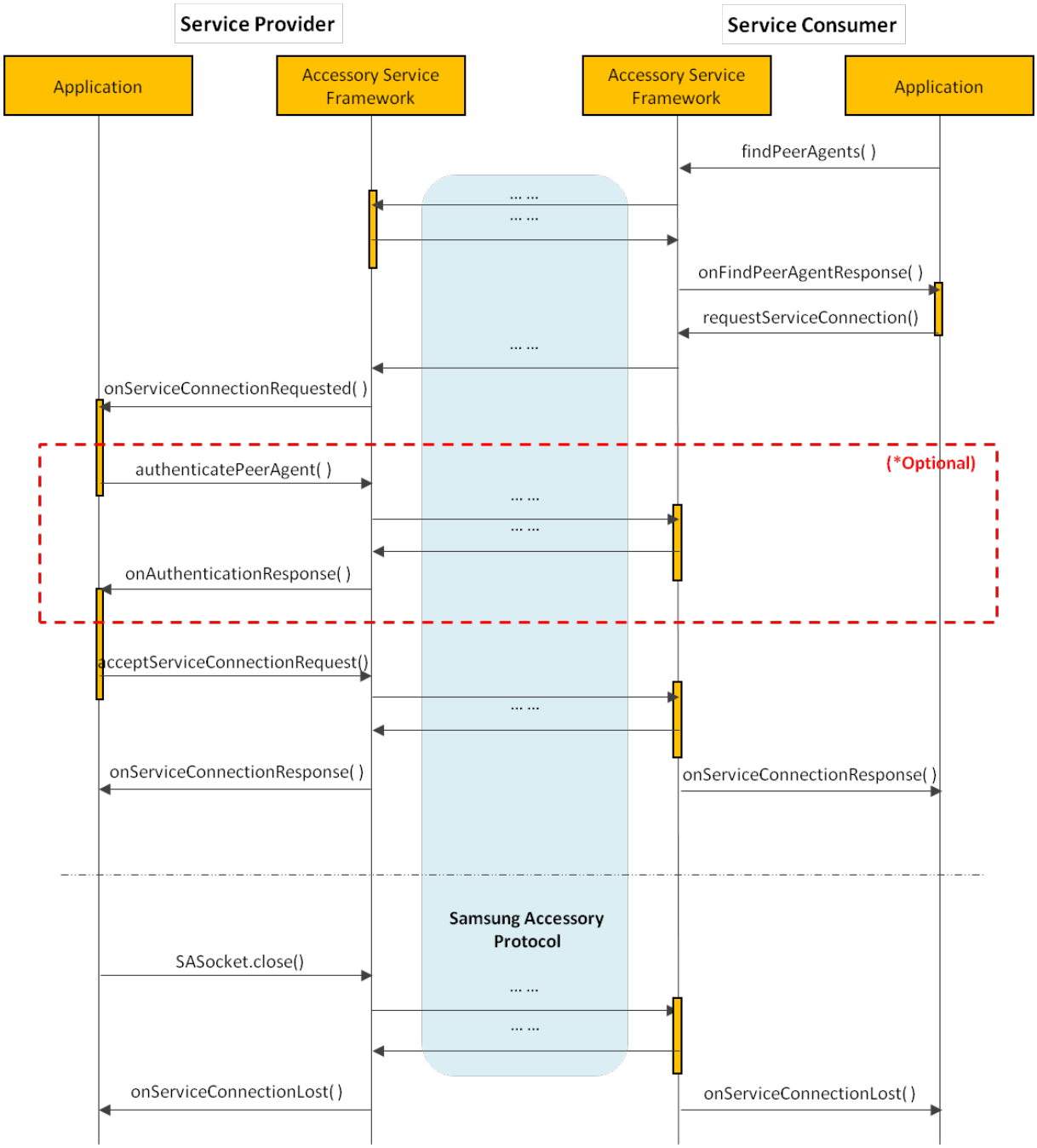

Figure 5: Sequence flow of the Accessory Peer Agent

1.3. Class Diagram

The following figure shows the Accessory classes and interfaces that you can use in your application.

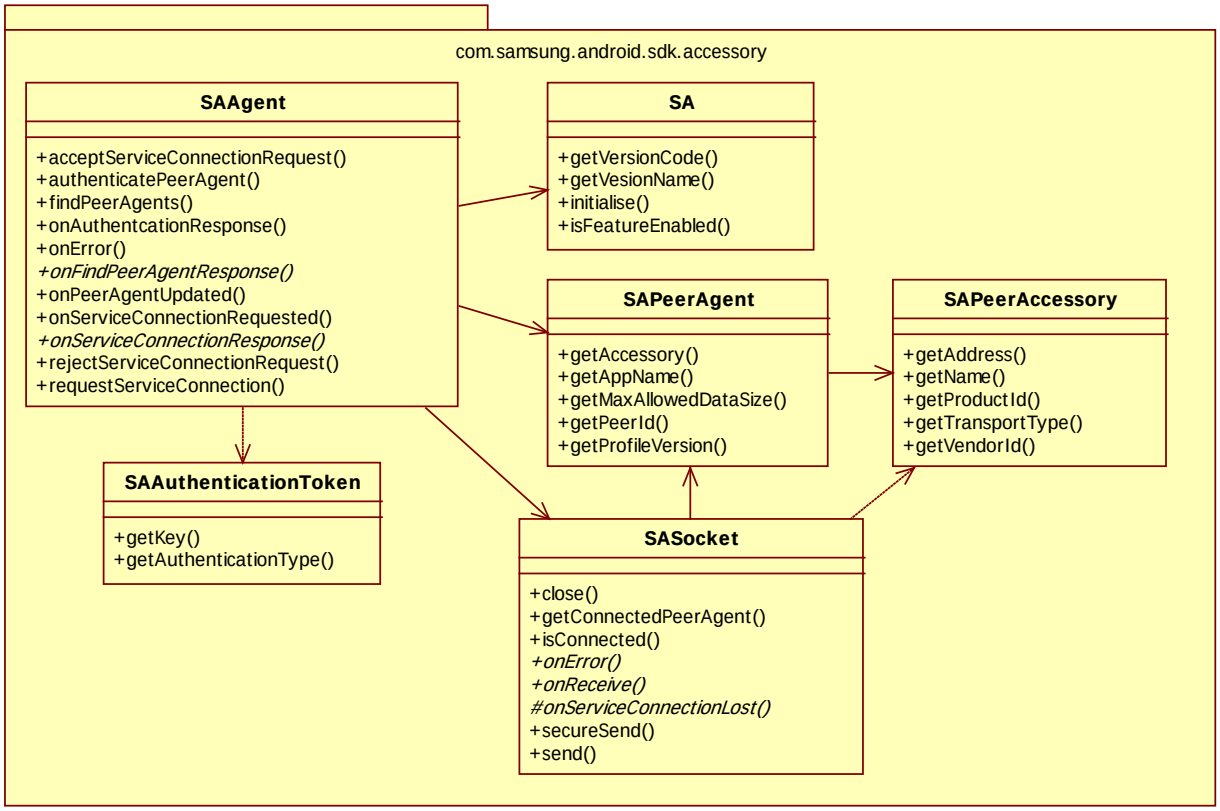

Figure 6: Accessory classes and interfaces

The Accessory classes and interfaces include:

- **SA:** Initializes Accessory.
- **SAAgent:** Represents an **Accessory Peer Agent**. Both Service Provider and Service Consumer implementations are expected to extend this class for each Accessory Service Profile instance they implement. This class exposes request methods creating outgoing Service Connections with matching remote Accessory Peer Agents. In case your Accessory Peer Agent sends an outgoing Service Connection request, your application is notified when the request result becomes available (with Service Connection establishment, through a rejection by the remote Accessory Peer Agent, or due to a failure). Remote Accessory Peer Agents can also initiate Service Connect requests with your Accessory Peer Agent.

  Your application is expected to implement the method handling for incoming Service Connection requests, deciding to accept or reject incoming Service Connection requests (trigger UI activities if needed). If a Service Connection is successfully established, both Accessory Peer Agents (Service Provider and Service Consumer at both ends of the Service Connection) are notified with a callback with an instance of the SASocket object passed by the Samsung Accessory Service Framework.
- **SASocket:** Represents a Service Connection between a Service Provider and a Service Consumer. This class handles Service Connection related events. Both the Service Consumer and Service Provider implementations extend this class to receive data on established Service Channels and send data according to the Accessory Service Profile specification.
- **SAPeerAgent:** Represents a **remote Accessory Peer Agent**. This is a passive entity that encapsulates the information of the remote Accessory Peer Agent. The remote Accessory Peer Agent information

includes such as the version of the Accessory Service Profile specification that the Accessory Peer Agent implements or follows, the application name, and the Accessory Device.

- **SAPeerAccessory:** Represents a remote Accessory Device. It is a component of SAPeerAgent. SAPeerAccessory is a passive entity encapsulating the information of an Accessory Device. SAPeerAccessory information includes such as the vendor ID, product ID, device name, and address.
- **SAAuthenticationToken:** Stores the type of authentication (Currently, it only supports X.509 certificate), and the key corresponding to the authentication type.

  Note: The Authentication may not be working properly depending on the firmware version of accessory device. It is recommended to upgrade accessory device firmware if possible.

For more information on the Accessory classes, see the Accessory API Reference.

1.4. Supported Platforms

Android 4.3 or above supports Accessory.

1.5. Supported Features

Samsung works with domain experts within and outside Samsung to define Accessory Service Profiles. The Accessory Service Profiles define the application-level state machine and application-level protocol to implement domain-specific functionality. For example, the Notification Accessory Service Profile defines an application-level protocol to convey phone notifications to connected Accessory Devices.

Accessory supports the following features:

- Accessory Peer Agent:
  - Getting the list of Accessory Peer Devices.
  - Getting the list of services offered by the Accessory Peer Devices.
  - Identifying the available services between Peer Devices.
- Service Connection:
  - Creating and storing the Service Connection between Accessory Peer Devices.
  - Initiating a Service Connection request
  - Processing Service Connection requests from Peer Devices to provide or consume a service.
  - Closing a Service Connection.

1.6. Components

- Components
  - accessory-v*x.y.z*.jar : Samsung Accessory SDK Library
  - sdk-v*x.y.z*.jar : Samsung SDK Library
  - Samsung Accessory Service Framework (preloaded on Samsung Smart Devices and Accessory Devices)

- Package to be imported:
    - com.samsung.android.sdk.accessory

1.7. Importing Libraries

To import Accessory libraries to the application project:

Add the following files to the libs folder of your Service Provider or Service Consumer application in Eclipse:

- accessory-v2.1.11.jar
- sdk-v1.0.0.jar

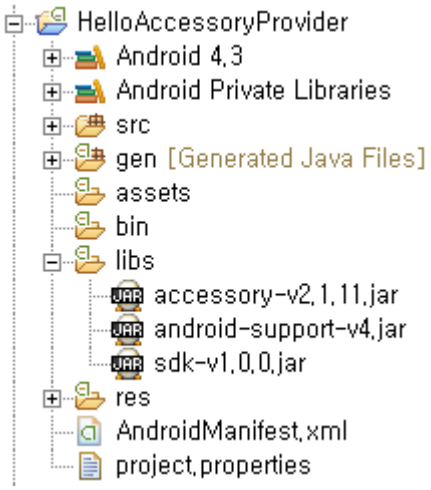

Figure 7: libs folder in Eclipse

The following permission has to be specified in the AndroidManifest.xml file to initialize Accessory.

```
<uses-permission android:name=
"com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY"/>
```

If you don't add the permission,

- Android 4.4.2 (KitKat) and above: SecurityException is thrown and your application doesn't work.
- Prior to Android 4.4.2 (KitKat): No exception. And the application works properly.

2. Hello Accessory

Hello Accessory consists of two applications: Hello Provider and Hello Consumer. Hello Provider can run on a Smart Device and Hello Consumer on an Accessory Device, or Hello Provider on an Accessory Device and Hello Consumer on a Smart Device.

Hello Accessory applications form a simple program that:

- Both Hello Accessory Provider and Hello Accessory Consumer initialize Accessory.
- Both Hello Accessory Provider and Hello Accessory Consumer register their capability with the Samsung Accessory Service Framework.
- Hello Accessory Consumer searches for Hello Accessory Provider, and initiates a Service Connection.
- Hello Accessory Provider sends a “Hello” string data to Hello Accessory Consumer.

2.1. Hello Accessory Provider

HelloAccessoryProviderService.java:

```
public class HelloAccessoryProviderService extends SAAgent {
    public static final String TAG = "HelloAccessoryProviderService";
    public static final int SERVICE_CONNECTION_RESULT_OK = 0;
    public static final int HELLOACCESSORY_CHANNEL_ID = 104;

    HashMap<Integer, HelloAccessoryProviderConnection> mConnectionsMap = null;

    private final IBinder mBinder = new LocalBinder();
    public class LocalBinder extends Binder {
        public HelloAccessoryProviderService getService() {
            return HelloAccessoryProviderService.this;
        }
    }

    public HelloAccessoryProviderService() {
        super(TAG, HelloAccessoryProviderConnection.class);
    }

    public class HelloAccessoryProviderConnection extends SASocket {
        private int mConnectionId;

        public HelloAccessoryProviderConnection() {
            super(HelloAccessoryProviderConnection.class.getName());
        }

        @Override
        public void onError(int channelId, String errorString, int error) {
            Log.e(TAG, "onError: " + error);
        }

        @Override
        public void onReceive(int channelId, byte[] data) {
            Time time = new Time();
            time.set(System.currentTimeMillis());
            String timeStr = " " + String.valueOf(time.minute) + ":" + String.valueOf(time.second);
            String strToUpdateUI = new String(data);
            String message = strToUpdateUI.concat(timeStr);

            HelloAccessoryProviderConnection uHandler = mConnectionsMap.get(Integer
                    .parseInt(String.valueOf(mConnectionId)));

            try {
                uHandler.send(HELLOACCESSORY_CHANNEL_ID, message.getBytes());
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        @Override
        protected void onServiceConnectionLost(int errorCode) {
            if (mConnectionsMap != null) {
                mConnectionsMap.remove(mConnectionId);
            }
        }
    }

    @Override
    public void onCreate() {
        super.onCreate();

        SA mAccessory = new SA();
        try {
            mAccessory.initialize(this);
            boolean isFeatureEnabled = mAccessory.isFeatureEnabled(SA.DEVICE_ACCESSORY);
        } catch (SsdkUnsupportedException e) {
            // Error Handling
        } catch (Exception e1) {
            e1.printStackTrace();
            /*
             * Your application can not use Accessory.
             * You application should work smoothly without using this SDK,
             * or you may want to notify user and close your app gracefully
             * (release resources, stop Service threads, close UI thread, etc.)
             */
            stopSelf();
        }
    }

    @Override
    protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
        acceptServiceConnectionRequest(peerAgent);
    }

    @Override
    protected void onFindPeerAgentResponse(SAPeerAgent arg0, int arg1) {
        // TODO Auto-generated method stub
    }

    @Override
    protected void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket thisConnection,
            int result) {
        if (result == CONNECTION_SUCCESS) {
            if (thisConnection != null) {
                HelloAccessoryProviderConnection myConnection
                                                = (HelloAccessoryProviderConnection) thisConnection;

                if (mConnectionsMap == null) {
                    mConnectionsMap = new HashMap<Integer, HelloAccessoryProviderConnection>();
                }

                myConnection.mConnectionId = (int) (System.currentTimeMillis() & 255);
                mConnectionsMap.put(myConnection.mConnectionId, myConnection);
                Toast.makeText(getBaseContext(),
                        R.string.ConnectionEstablishedMsg, Toast.LENGTH_LONG).show();
            } else {
                Log.e(TAG, "SASocket object is null");
            }
        } else if (result == CONNECTION_ALREADY_EXIST) {
            Log.e(TAG, "onServiceConnectionResponse, CONNECTION_ALREADY_EXIST");
        } else {
            Log.e(TAG, "onServiceConnectionResponse result error =" + result);
        }
    }

    @Override
    public IBinder onBind(Intent arg0) {
```

```
        return mBinder;
    }
}
```

2.2. Hello Accessory Consumer

HelloAccessoryConsumerService.java:

```
public class HelloAccessoryConsumerService extends SAAgent {
    public static final String TAG = "HelloAccessoryConsumerService";
    public static final int HELLOACCESSORY_CHANNEL_ID = 104;

    @Override
    protected void onError(SAPeerAgent peerAgent, String errorMessage, int errorCode) {
        super.onError(errorMessage, errorCode);
    }

    private final IBinder mBinder = new LocalBinder();
    Handler mHandler = new Handler();

    public HelloAccessoryConsumerService() {
        super("HelloAccessoryConsumerService", HelloAccessoryConsumerConnection.class);
    }

    public class LocalBinder extends Binder {
        public HelloAccessoryConsumerService getService() {
            return HelloAccessoryConsumerService.this;
        }
    }

    private SASocket mConnectionHandler;

    public class HelloAccessoryConsumerConnection extends SASocket {
        public HelloAccessoryConsumerConnection() {
            super(HelloAccessoryConsumerConnection.class.getName());
        }

        @Override
        public void onError(int channelId, String errorMessage, int errorCode) {
        }

        @Override
        public void onReceive(int channelId, byte[] data) {
             final String strToUpdateUI = new String(data);
             mHandler.post(new Runnable() {
                @Override
                public void run() {
                    HelloAccessoryActivity.mTextView.setText(strToUpdateUI);
                }
            });
        }

        @Override
        protected void onServiceConnectionLost(int reason) {
             closeConnection();
        }
    }

    public void findPeers() {
         findPeerAgents();
    }

    @Override
    public void onCreate() {
        super.onCreate();

        SA mAccessory = new SA();
        try {
```

```
            mAccessory.initialize(this);
            boolean isFeatureEnabled = mAccessory.isFeatureEnabled(SA.DEVICE_ACCESSORY);
        } catch (SsdkUnsupportedException e) {
            // Error Handling
        } catch (Exception e1) {
            Log.e(TAG, "Cannot initialize Accessory.");
            e1.printStackTrace();
            /*
             * Your application can not use Accessory.
             * You application should work smoothly without using this SDK,
             * or you may want to notify user and close your app gracefully
             * (release resources, stop Service threads, close UI thread, etc.)
             */
            stopSelf();
        }
    }

    @Override
    protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
        acceptServiceConnectionRequest(peerAgent);
    }

    @Override
    protected void onFindPeerAgentResponse(SAPeerAgent remoteAgent, int result) {
       if (result == PEER_AGENT_FOUND) {
            onPeerFound(remoteAgent);
       }
    }

    @Override
    protected void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket thisConnection,
            int connResult) {
        if (connResult == CONNECTION_SUCCESS) {
             this.mConnectionHandler = thisConnection;
             Toast.makeText(getBaseContext(), R.string.ConnectionEstablishedMsg, Toast.LENGTH_LONG).show();
        } else if (connResult == CONNECTION_ALREADY_EXIST) {
            Toast.makeText(getBaseContext(), R.string.ConnectionAlreadyExist, Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(getBaseContext(), R.string.ConnectionFailure, Toast.LENGTH_LONG).show();
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
         return mBinder;
    }

    public boolean sendHelloAccessory() {
        boolean retvalue = false;
        String jsonStringToSend = "Hello Accessory!";

        if (mConnectionHandler != null) {
            try {
                mConnectionHandler.send(HELLOACCESSORY_CHANNEL_ID, jsonStringToSend.getBytes());
                retvalue = true;
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            Log.d(TAG, "Requested when no connection");
        }
        return retvalue;
    }

    public void onPeerFound(SAPeerAgent remoteAgent) {
         if (remoteAgent != null) {
             establishConnection(remoteAgent);
        } else {
             Toast.makeText(getApplicationContext(), R.string.NoPeersFound, Toast.LENGTH_LONG).show();
        }
    }

    public boolean closeConnection() {
```



```
        if (mConnectionHandler != null) {
            mConnectionHandler.close();
            mConnectionHandler = null;
        } else {
            Log.d(TAG, "closeConnection: Called when no connection");
        }
        return true;
    }

    public boolean establishConnection(SAPeerAgent peerAgent) {
        if (peerAgent != null) {
            requestServiceConnection(peerAgent);
            return true;
        }
        return false;
    }
}
```

HelloAccessoryActivity.java:

```
public class HelloAccessoryActivity extends Activity {
    public static final String TAG = "HelloAccessoryActivity";
    private HelloAccessoryConsumerService mConsumerService = null;
    private boolean mIsBound = false;
    public static TextView mTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
        mTextView = (TextView) findViewById(R.id.text1);
        doBindService();
    }

    @Override
    protected void onDestroy() {
        closeConnection();
        doUnbindService();
        super.onDestroy();
    }

    void doBindService() {
        mIsBound = bindService(new Intent(HelloAccessoryActivity.this, HelloAccessoryConsumerService.class),
                mConnection, Context.BIND_AUTO_CREATE);
    }

    void doUnbindService() {
        if (mIsBound) {
            unbindService(mConnection);
            mIsBound = false;
        }
    }

    public void mOnClick(View v) {
        switch (v.getId()) {
            case R.id.button1: {
                startConnection();
                break;
            }
            case R.id.button2: {
                closeConnection();
                break;
            }
            case R.id.button3: {
                sendHelloAccessory();
                break;
            }
        }
    }
```



```
    private void startConnection() {
        if (mIsBound == true && mConsumerService != null) {
            mTextView.setText("startConnection");
            mConsumerService.findPeers();
        }
    }

    private void closeConnection() {
         if (mIsBound == true && mConsumerService != null) {
            mTextView.setText("closeConnection");
            mConsumerService.closeConnection();
        }
    }

    private void sendHelloAccessory() {
        if (mIsBound == true && mConsumerService != null) {
            mTextView.setText("sending HelloGear!");
            mConsumerService.sendHelloAccessory();
        }
    }

    private final ServiceConnection mConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName className, IBinder service) {
           mConsumerService = ((HelloAccessoryConsumerService.LocalBinder) service).getService();
           mConsumerService.findPeers();
           mTextView.setText("onServiceConnected");
        }

        @Override
        public void onServiceDisconnected(ComponentName className) {
            mConsumerService = null;
            mIsBound = false;
            mTextView.setText("onServiceDisconnected");
        }
    };
}
```

3. Using the SA Class

The SA class provides the following methods:

- `initialize()` initializes Accessory. You need to initialize Accessory before you can use it. If the device does not support Accessory, SsdkUnsupportedException is thrown.
- `getVersionCode()` gets the Accessory library version number as an integer.
- `getVersionName()` gets the Accessory library version name as a string.
- `isFeatureEnabled()` checks if the Accessory feature is available on the device.

```
SA SAPackage = new SA();

try{
        SAPackage.initialize (applicationContext) {
        boolean isFeatureEnabled = mAccessory.isFeatureEnabled(SA.DEVICE_ACCESSORY);
} catch (final SsdkUnsupportedException e) {
        //try to handle SsdkUnsupportedException
        if( processUnsupportedException(e) == true) {
            return;
        }
} catch (Exception e1) {
    /* Your application cannot use Accessory .
     * You application should work smoothly without using Accessory,
     * or you may want to notify the user and close your app gracefully
     * (release resources, stop Service threads, close UI thread, etc.)
     */
}
```

3.1. Using the initialize() Method

The `SA.initialize()` method:

- Initializes Accessory.
- Checks if the device is a Samsung device.
- Checks if the device supports Accessory.
- Checks if the Accessory libraries are installed on the device.

```
void initialize(Context context) throws SsdkUnsupportedException
```

If Accessory fails to initialize, the `initialize()` method throws an `SsdkUnsupportedException` exception. To find out the reason for the exception, check the exception message.

3.2. Handling SsdkUnsupportedException

If an SsdkUnsupportedException exception is thrown, check the exception message type using `SsdkUnsupportedException.getType()`.

The following types of exception messages are defined in the SA class:

- DEVICE_NOT_SUPPORTED: The device does not support Accessory**.**
- LIBRARY_NOT_INSTALLED**:** The device does not have the Accessory library installed.
- LIBRARY_UPDATE_IS_REQUIRED**:** The device must update the Accessory library.
- LIBRARY_UPDATE_IS_RECOMMENDED**:** The device is recommended to update the Accessory library.

3.3. Checking the Availability of Accessory Features

You can check if the Accessory feature is supported on the device with the `isFeatureEnabled()` method. The feature types are defined in the SA class. The feature type is passed as a parameter when calling the `isFeatureEnabled()` method. The method returns a Boolean value that indicates the support for the feature on the device.

```
boolean isFeatureEnabled(int type)
```

The flowing type is defined in the SA class:

- DEVICE_ACCESSORY

4. Using Accessory

The following chapter describes how to use Accessory.

4.1. Configuring Your Application

When you create your application, you must add the following to your Android manifest file and to your project:

Add permissions for your Service Provider or Service Consumer application to your Android manifest file to use the Samsung Accessory Service Framework.

```
<uses-permission android:name=”com.samsung.accessory.permission.ACCESSORY_FRAMEWORK”/>
<uses-permission
    android:name="com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY"/>
```

Add your Service Provider or Service Consumer broadcast receivers inside the <application> element for the intents handled by Accessory package in your Android manifest file.

```
<receiver android:name ="com.samsung.android.sdk.accessory.ServiceConnectionIndicationBroadcastReceiver">
    <intent-filter>
    <action android:name="android.accessory.service.action.ACCESSORY_SERVICE_CONNECTION_IND"/>
</intent-filter>
</receiver>
<receiver android:name = "com.samsung.android.sdk.accessory.RegisterUponInstallReceiver">
<intent-filter>
    <action android:name="android.accessory.device.action.REGISTER_AFTER_INSTALL"/>
</intent-filter>
</receiver>
```

Add the path of your Service Profile XML file to your Android manifest file.
(for example, /res/xml/<*profileName*>.xml)

```
<meta-data android:name="AccessoryServicesLocation" android:value="/res/xml/<profileName>.xml"/>
```

Declare your Service Provider or Service Consumer class derived from `SAAgent` as a service in your Android manifest file. The `SAAgent` class extends the Android service and handles asynchronous Accessory-related intents. Its implementation executes all of its activities in a worker thread, which means it does not overload your application’s main thread.

```
<service android:name="com.samsung.accessory.alarmProvider.backend.SAAlarmProviderImpl"/>
```

4.2. Registering Your Service Provider or Service Consumer

Your Accessory Peer Agent implementations, both the Service Provider and Service Consumer applications, have to extend the `SAAgent` and `SASocket` base classes. These base classes allow you to focus on implementing your functionality and free you from managing the details of Accessory.

Register the Service Provider or Service Consumer with the Samsung Accessory Service Framework by specifying the Accessory Service Profile description. The Samsung Accessory Service Framework enters this in the local capability database. The capability exchange module advertises the services registered with the connected Accessory Devices.

The registration process expects the Accessory Service Profile description in the Service Profile XML file to be located in the */res/xml* folder of your Android project. If your application implements multiple Service Providers or Service Consumers, you must declare multiple Accessory Service Profile descriptions in the Service Profile XML file. For more information, see the following Service Profile XML file example.

```
<resources>
<application name="ProviderExample ">
<serviceProfile
        serviceImpl="com.samsung.accessory.example.providerServiceImpl"
        role="Provider"
        name="ProviderService"
        id="/app/example"
        version="1.0"
        serviceLimit="any"
        serviceTimeout="10">
        <supportedTransports>
            <transport type="TRANSPORT_BT"/>
            <transport type="TRANSPORT_WIFI"/>
        </supportedTransports>
        <serviceChannel
            id="910"
            dataRate="low"
            priority="high"
            reliability="enable"/>
</serviceProfile>
<serviceProfile
        serviceImpl="com.samsung.accessory.example.extproviderServiceImpl”
        role="Provider"
        name="ExtProviderService"
        id="/app/extexample"
        version="1.0"
        serviceLimit="any">
        <supportedTransports>
            <transport type="TRANSPORT_BT"/>
            <transport type="TRANSPORT_WIFI"/>
        </supportedTransports>
        <serviceChannel
            id="902"
            dataRate="low"
            priority="low"
            reliability="disable"/>
</serviceProfile>
</application>
</resources>
```

- "***application name***": This attribute is the name that you want the Samsung Accessory Service Framework to advertise in the Accessory eco-system. Usually the application's Android AppName is used. You can implement multiple Service Providers and Service Consumers in one application. In that case, declare multiple <*serviceProfile*> elements inside the <*application*> element.
- In each <***serviceProfile***> element:
    - "***serviceImpl***" attribute is your subclass that extends `SAAgent`.
    - "***role***" attribute is either "Provider" or "Consumer".
    - "***name***" attribute is the friendly name of your Service Provider or Service Consumer.
    - "***id***" attribute is the Service Profile ID of the Service Provider or Service Consumer.

- "***version***" attribute specifies the Service Profile specification version that your Service Provider or Service Consumer application supports.
- "***serviceLimit***" attribute sets how many Accessory Peer Agents you want to connect with concurrently. If an Accessory Peer Agent requests a Service Connection with your application after you have reached the limit, the Samsung Accessory Service Framework rejects the Service Connection request. The attribute can be one of the following values:

  - *one_ peeragent*: Supports only one Accessory Peer Agent.
  - *one_ accessory* : Supports only one Accessory Device but can have Service Connections to multiple Accessory Peer Agents on that Accessory Device.
  - *any*: Supports multiple Accessory Peer Agents on multiple Accessory Devices.

- "***serviceTimeout***" attribute controls the timeout for handling incoming Service Connection requests. This attribute is optional. If you do not set the value, the default timeout is applied. Use the default timeout unless your application needs prolonged time to make the decision to accept or reject incoming Service Connection requests. If it needs, e.g., to connect to a cloud server, show a UI prompting the user to make a decision whether to accept or reject. If, on the other hand, it needs to do authentication, set the attribute value for the timeout of the decision. If the timeout has exceeded, the requesting Accessory Peer Agent gets the response that Service Connection failed because your application did not respond.
- ***<supportedTransports>*** element declares the transports on which the Service Provider or Service Consumer is able to operate. The Samsung Accessory Service Framework supports the TRANSPORT_WIFI, TRANSPORT_BT, TRANSPORT_BLE, and TRANSPORT_USB transport types. If your Service Provider or Service Consumer supports multiple transport types, declare multiple <*transports*> elements.

  Note: The current version of the Samsung Accessory Service Framework supports only TRANSPORT_BT . Other types will be supported soon.

- In each **<serviceChannel>** element:

  - "*dataRate*" attribute must be either "*low*" or "*high*".
  - "*priority*" attribute must be "*low*", "*medium*", or "*high*".
  - "*reliability*" attribute must be "*enable*" or "*disable*". The attribute defines whether you want a reliable transfer or not. In case of a packet drop, a reliable transfer re-transmits the packet but also creates additional overhead.

When the application is installed, the Samsung Accessory Service Framework automatically registers its Accessory Peer Agents using the information specified in your Service profile XML file. Similarly, the Accessory Peer Agents are deregistered when the application is uninstalled. An error log is dumped if the registration process fails to register the Accessory Service Profile implementation.

4.2.1. Validating the Service Profile XML

The Samsung Accessory Service Framework Document Type Definition (DTD) schema validation significantly lowers the chances of registration failure. The following code snippet shows the Accessory Service Profile XML file DTD.

```
<!DOCTYPE resources [
```

```
<!ELEMENT resources (application)>
<!ELEMENT application (serviceProfile)+>
<!ATTLIST application name CDATA #REQUIRED>
<!ELEMENT serviceProfile (supportedTransports, serviceChannel+) >
<!ATTLIST application xmlns:android CDATA #IMPLIED>
<!ATTLIST serviceProfile xmlns:android CDATA #IMPLIED>
<!ATTLIST serviceProfile serviceImpl CDATA #REQUIRED>
<!ATTLIST serviceProfile role (PROVIDER | CONSUMER | provider | consumer) #REQUIRED>
<!ATTLIST serviceProfile name CDATA #REQUIRED>
<!ATTLIST serviceProfile id CDATA #REQUIRED>
<!ATTLIST serviceProfile version CDATA #REQUIRED>
<!ATTLIST serviceProfile serviceLimit
(ANY | ONE_ACCESSORY | ONE_PEERAGENT | any | one_peeragent | one_accessory) #IMPLIED>
<!ATTLIST serviceProfile serviceTimeout CDATA #IMPLIED>
<!ELEMENT supportedTransports (transport)+>
<!ATTLIST supportedTransports xmlns:android CDATA #IMPLIED>
<!ELEMENT transport EMPTY>
<!ATTLIST transport xmlns:android CDATA #IMPLIED>
<!ATTLIST transport type (TRANSPORT_WIFI | TRANSPORT_BT | TRANSPORT_BLE | TRANSPORT_USB | transport_wifi |
transport_bt | transport_ble | transport_usb) #REQUIRED>
<!ELEMENT serviceChannel EMPTY>
<!ATTLIST serviceChannel xmlns:android CDATA #IMPLIED>
<!ATTLIST serviceChannel id CDATA #REQUIRED>
<!ATTLIST serviceChannel dataRate (LOW | HIGH | low | high) #REQUIRED>
<!ATTLIST serviceChannel priority (LOW | MEDIUM | HIGH | low | medium | high) #REQUIRED>
<!ATTLIST serviceChannel reliability (ENABLE | DISABLE | enable | disable ) #REQUIRED>
]>
```

To include the DTD validation rules at the very top of your Accessory Service Profile XML file:

- In the Eclipse IDE, under Eclipse setting:

  Window > Preferences > XML > XML Files > Validation**,** select **Enable markup validation**.
- Set **No grammar specified**.
- Set **Missing root element** to **Ignore**.

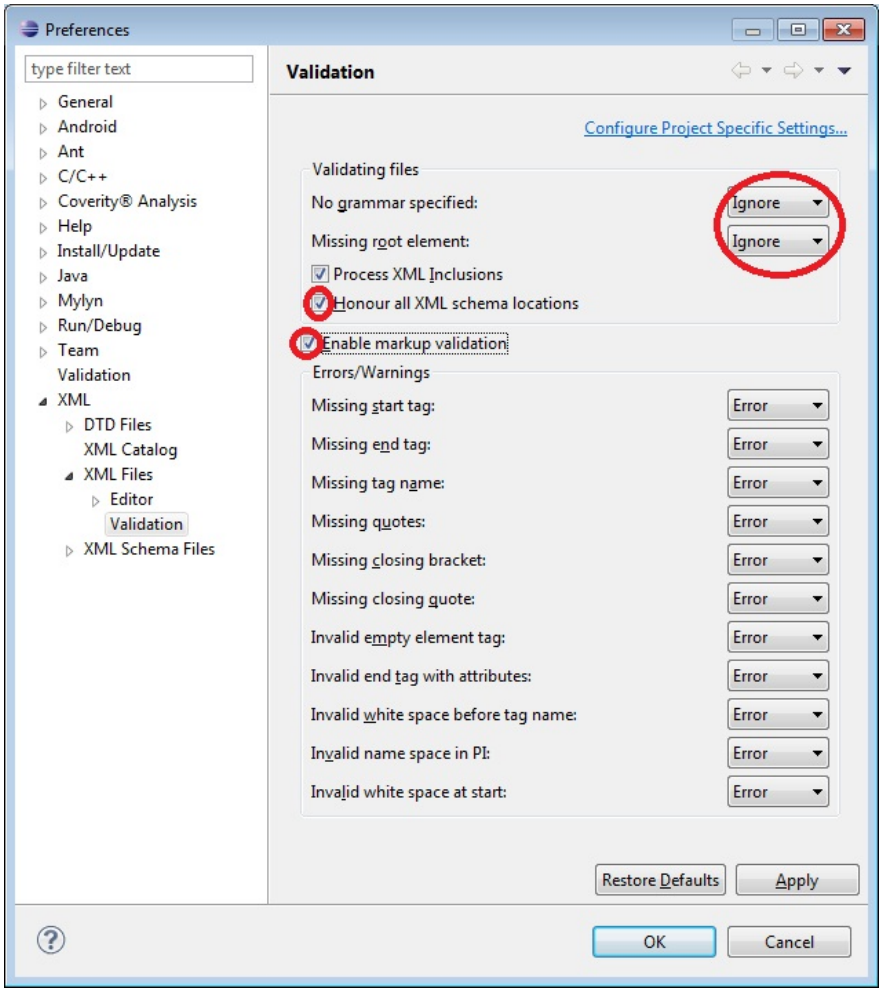

Figure 8: Eclipse IDE XML validation settings

Eclipse validates the Accessory XML file when you build your application to check whether the XML file follows the Samsung Accessory Service Framework DTD.

You can also validate the XML any time by right-clicking on the XML file and selecting **Validate**.

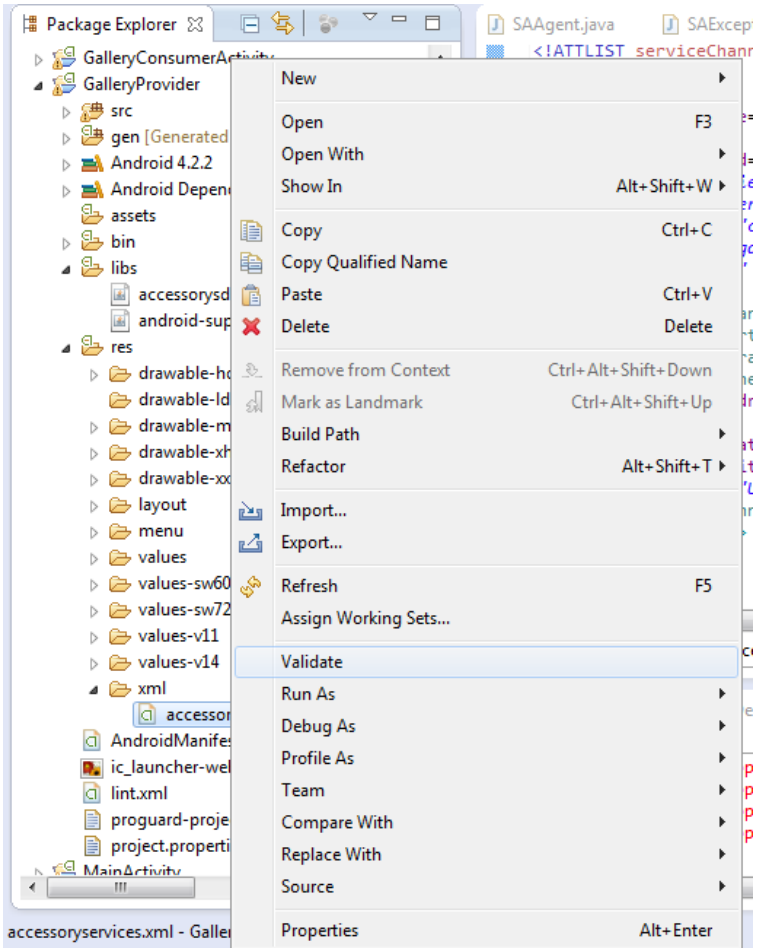

Figure 4: Eclipse IDE XML Validation

4.3. Finding a Matching Accessory Peer Agent and Initiating a Service Connection

Your Service Provider or Service Consumer application can search for matching Accessory Peer Agents by calling the `SAAgent.findPeerAgents()` method. Matching Accessory Peer Agents have the same Accessory Service Profile, i.e., Notification Service or Weather Service, and have a complementary provider or consumer relationship with the calling Accessory Peer Agent. Accessory Peer Agents with different Accessory Service Profiles for Service Providers or Service Consumers do not "match" and cannot be connected with each other. If two Accessory Peer Agents have the same Accessory Service Profile with different versions, however, they are still considered to "match". For example, Notification Service Consumer that implements the Notification Service Profile version 2.0 and a Notification Service Provider that implements the Notification Service Profile version 1.0, "match" .

If a matching Accessory Peer Agent is found, the calling Accessory Peer Agent is notified with the `onFindPeerAgentResponse()` callback method. If multiple matching Accessory Peer Agents are found, the callback occurs multiple times, one for each matching Accessory Peer Agent. If no Accessory Peer Agent is found, the calling Accessory Peer Agent is notified with the same callback method, but the peerAgent parameter is null and the result parameter includes the reason why there is no match found.

If your application wants to establish a Service Connection with only one Accessory Peer Agent, check the first callback. You can also check the identity or properties of the discovered Accessory Peer Agents by calling the methods provided by the SAPeerAgent class to decide which Accessory Peer Agent you want to form a Service Connection.

You can initiate a Service Connection with an Accessory Peer Agent by calling requestServiceConnection(). This method is called from a worker thread. If you need to do any heavy lifting or long latency work in this callback, spawn a separate thread.

```
@Override
protected void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
    if(result == PEER_AGENT_FOUND) {
        requestServiceConnection(peerAgent);
    } else if(result == FINDPEER_DEVICE_NOT_CONNECTED){
        Log.i(TAG, "Peer Agents are not found, no accessory device connected.");
    } else if(result == FINDPEER_SERVICE_NOT_FOUND ) {
        Log.i(TAG, "No matching service on connected accessory.");
    }
}
```

If a Service Provider connects only with a specific Service Consumer, or a Service Consumer with a specific Service Provider, the Service Provider and Consumer are known as ”companion apps”. When you only want to connect to a companion Service Provider or Service Consumer, call the methods provided by the SAPeerAgent class for specific information, such as model number or vendor information, before calling requestServiceConnection(). For example, when a photo printer Service Provider on an Accessory Device from a company only wants to connect to a photo printer Service Consumer on a Smart Device from the same company, they are companion apps.

After you call findPeerAgents(), the Samsung Accessory Service Framework keeps track of any changes in the availability of the matching Accessory Peer Agents for your application. If a change occurs, your application is notified with the onPeerAgentUpdated() callback. This happens especially when an Accessory Device with a matching Accessory Peer Agent is connected or disconnected, or a matching Accessory Peer Agent is installed or uninstalled on a remote Accessory Device. If a matching Accessory Peer Agent is not found when calling findPeerAgents(), the onFindPeerAgentResponse() callback gets a failure code. When it becomes available, you can get the PEER_AGENT_AVAILABLE with the onPeerAgentUpdated() callback. Your application can check the identity or properties of the new Accessory Peer Agent by using the APIs in the SAPeerAgent object, and decide whether to request a Service Connection with that Accessory Peer Agent

```
@Override
protected void onPeerAgentUpdated(SAPeerAgent peerAgent, int result) {
    if(result == PEER_AGENT_AVAILABLE) {
        requestServiceConnection(peerAgent);
    } else if (result == PEER_AGENT_UNAVAILABLE) {
        Log.i(TAG,"Peer Agent no longer available:" + peerAgent.getAppName());
    }
}
```

The remote Accessory Peer Agent accepts or rejects your Service Connection request. Your application is notified with the onServiceConnectionResponse() callback. The request can either be accepted and a Service Connection is established, rejected, or failed to establish Service Connection for other reasons.

When a Service Connection is successfully established, the requesting Accessory Peer Agent gets an instance of the SASocket object, which is used to handle Service Connection events and to send data or receive it from Accessory Peer Agents.

```
@Override
protected void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket socket, int result) {
    if(result == CONNECTION_SUCCESS) {
        /* HelloAccessoryConsusmerConnection is SASocket derived object for this sample.
         * This is passed when Service Connection is established.
         */
        mSocket = (HelloAccessoryConsumerConnection)socket;
    } else {
        Log.e(TAG,"Service Connection establishment failed" + result);
    }
}
```

4.4. Handling Service Connection Requests

The Service Provider or Consumer application is notified with the onServiceConnectionRequested() callback when remote Accessory Peer Agents want to create a Service Connection with it. The Accessory Peer Agent implementation can accept or reject Service Connection requests by calling the acceptServiceConnectionRequest() or rejectServiceConnectionRequest() method. The default implementation of the onServiceConnectionRequested() callback method is to accept every incoming Service Connection request from any remote Accessory Peer Agent. Your Accessory Peer Agent implementation can override this method, usually to check the identity and properties of the requesting remote Accessory Peer Agent before accepting or rejecting incoming Service Connection requests.

The onServiceConnectionRequested() callback can check for Accessory Peer Agent specific information before accepting Service Connection requests. You can use the SAPeerAgent object methods for checking specific information, such as application name or vendor ID. In addition, you can optionally authenticate the Peer Agent by checking its key and then decide to accept or reject its Service Connection request.

Example of checking the Accessory Peer Agent information:

```
@Override
protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
    // Make a decision after checking the validation of given information.
    if (peerAgent.getAccessory().getVendorId.equals("Star Wars series")
       && peerAgent.getAccessory().getProductId.equals("Death Star")) {
        acceptServiceConnectionRequest(peerAgent);
    } else {
        rejectServiceConnectionRequest(peerAgent);
    }
}
```

Example of authenticating the Accessory Peer Agent:

```
@Override
protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
    // Check Peer Agent's basic info
    if(peerAgent.getAccessory().getVendorId.equals("Star Wars series")
       && peerAgent.getAccessory().getProductId.equals("Death Star")){
```

```
        // Authenticate Peer Agent for enhanced security
        authenticatePeerAgent(SAPeerAgent uPeerAgent);
    } else {
        rejectServiceConnectionRequest(peerAgent);
    }
}

@Override
onAuthenticationResponse(SAPeerAgent peerAgent,SAAuthenticationToken authToken,int code){
     // callback for calling authenticatePeerAgent, check Accessory Peer Agent's key
     if(code == AUTHENTICATION_SUCCESS)
        && peerAgent.getAccessory().getVendorId.equals("Star Wars series")
        && peerAgent.getAccessory().getProductId.equals("Death Star")){

        if(authToken.getAuthenticationType() == AUTHENTICATION_TYPE_CERTIFICATE_X509
            && Arrays.equals(authToken.getKey(), JediWarriorKey)){
                acceptServiceConnectionRequest(peerAgent);
                return;
        }
        rejectServiceConnectionRequest(peerAgent);
     } else {
        rejectServiceConnectionRequest(peerAgent);
     }
}
```

Note: The authenticating Accessory Peer Agent may not be working properly depending on the firmware version of accessory device. It is recommended to upgrade accessory device firmware if possible.

If your application accepts the Service Connection request, your application is notified with the `onServiceConnectionResponse()` callback when the Service Connection is established or a failure occurs. On success, a `SASocket` object is passed with the callback. If you want to implement a Service Provider application that can serve multiple Service Consumer applications at the same time, keep a repository of the `SASocket` objects for all active Service Connections and give an identifier for each `SASocket` object. The `onServiceConnectionResponse()` callback is called from a worker thread. If you need to do any heavy lifting or long latency work in this callback, spawn a separate thread.

```
HashMap<Integer, HelloAccessoryProviderConnection> mConnectionsMap = null;

@Override
protected void onServiceConnectionResponse (SAPeerAgent peerAgent, SASocket socket, int result) {
    if(result == CONNECTION_SUCCESS) {
        // Keep a repository of connected SASocket. This sample uses HashMap
        if(mConnectionsMap == null) {
            mConnectionsMap = HashMap<Integer, HelloAccessoryProviderConnection>();
        }
        // HelloAccessoryProviderConnection extends SASocket
        HelloAccessoryProviderConnection providerConnection =
            (HelloAccessoryProviderConnection) socket;
        // Assign unique identifier to each SASocket(HelloAccessoryProviderConnection) object
        providerConnection.mSocketId = mSocketCounter++;
        mConnectionsMap.put(String.valueOf(providerConnection.mSocketId), providerConnection);
    } else if(result == CONNECTION_FAILURE_NETWORK) {
        for(;i<=5;i++){
        try {
            wait(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        requestServiceConnection(peerAgent);
    } else if (result == CONNECTION_FAILURE_PEER_AGENT_REJECTED) {
        Log.i(TAG, "Peer Agent Rejected");
    } else if (result == CONNECTION_FAILURE_PEER_AGENT_NO_RESPONSE) {
        Log.i(TAG, "Peer Agent no response");
    } else if (result == CONNECTION_FAILURE_DEVICE_UNREACHABLE) {
```

```
            Log.e(TAG, "Accessory Device not reachable, may already be disconnected");
        } else {
            Log.e(TAG, "Service Connection Fail, no-recoverable error");
        }
    }
```

4.5. Exchanging Data with Accessory Peer Agents

Both Service Provider and Service Consumer applications implement a subclass of `SASocket` to send and receive data over an established Service Connection. Register your `SASocket` implementation with `SAAgent` by passing the name and the derived concrete `SASocket` subclass as parameters to the `SAAgent` constructor for Java Reflection construction.

```
public HelloAccessoryConsumerService() {
    // HelloAccessoryConsumerService extends SAAgent
    // HelloAccessoryConsumerConnection extends SASocket
    super("HelloAccessoryConsumerService", HelloAccessoryConsumerConnection.class);
}
```

Both the Service Provider and Service Consumer application also need to implement the `SASocket` subclass constructor for Java Reflection construction. The following example illustrates the implementation.

```
public HelloAccessoryConsumerConnection() {
    // HelloAccessoryConsumerConnection extends SASocket
    // name of the subclass extends SASocket
    super(HelloAccessoryConsumerConnection.class.getName());
}
```

When you build your application in release mode, you must add **-keep** lines in the proguard.cfg file of your application, to prevent ProGuard from renaming your subclasses of `SAAgent` and `SASocket`. ProGuard is a tool integrated into the Android build system that obfuscates the code by renaming classes and methods. For more information about how to configure ProGuard, please visit the ProGuard page of Android Development Site and see section 5.

To send data to your connected Accessory Peer Agent, call the `send()` method of the SASocket object passed with the `onServiceConnectionResponse()` callback to send data on the selected Service Channel inside an established Service Connection. The Samsung Accessory Service Framework provides a datagram service. Either all the data is sent or nothing is sent. The Service Connection encapsulates all Service Channels as defined by the Accessory Service Profile specification.

Do not send byte array bigger than `SAPeerAgent.getMaxAllowedDataSize()`, which returns the limit of size that you can send to the remote Accessory Peer Agent. The limit is a variable depends on transport type and memory size of the remote Accessory Device.

```
try {
    mSocket.send(GALLERY_CHANNEL_ID, mJsonStringToSend.getBytes());
} catch (IOException e) {
    e.printStackTrace();
}
```

If you want your data encrypted, call `secureSend()`.

```
try {
     mSocket.secureSend(GALLERY_CHANNEL_ID, mJsonStringToSend.getBytes());
} catch (IOException e) {
     e.printStackTrace();
}
```

Note: send() and secureSend() methods are called from a worker thread. If you need to do any heavy lifting or long latency work in this callback, spawn a separate thread. Do not invoke this method on main thread of the application.

When your application receives data from a remote Accessory Peer Agent, it is notified with the `SASocket.onReceive()` callback. Implement the `onReceive()` method to handle the data.

```
public class HelloAccessoryConsumerServiceConnection extends SASocket{
    @Override
    public class onReceive(int channelId, byte[] data) {
        String str = new String(data);
    }
}
```

4.6. Disconnecting and Error Handling

Call the `close()` method in the SASocket object to terminate the Service Connection with the remote Accessory Peer Agent. The remote Accessory Peer Agent is notified with the `onServiceConnectionLost()` callback and the Samsung Accessory Service Framework closes all the established Service Channels of the Service Connection. If a remote Accessory Peer Agent calls `close()` to terminate the Service Connection, your application is notified with the same callback.

If a Service Connection is lost, for instance, due to a network failure or devices leaving the wireless connectivity range, the Accessory Peer Agents are notified with the `onServiceConnectionLost()` callback. However, it is not necessary to call close in the onServiceConnectionLost callback, since the Service connection is already closed and cleaned up. You can handle these events by implementing the method illustrated in the following example.

Note: If you want to restore Service Connection, it is your application's responsibility to call `findPeerAgents()` to try to re-find the remote Accessory Peer Agent and `requestServiceConnection()` to make Service Connection request again.

```
@Override
public void onServiceConnectionLost(int reason) {
    /**
     * This function is called when Service Connection is broken or lost
     *  or peer disconnection
     */
     activityBoundToTheService.handleConnectionLost(reason); //application logic
     switch(reason){
         case CONNECTION_LOST_DEVICE_DETACHED:
             //if the Peer Agent is killed because of LMK OOM,
             //better to call findPeerAgents() and request Service Connection
             //especially for Service Consumer, Accessory will invoke Peer Agent
             yourSAAgentImpl.tryConnect();
             // in your implementation of that method
             // you should follow the procedures talked in Section 4.3
```

```
            // “Find Matching Peer Agent and Initiate Service Connection”
            break;
        case CONNECTION_LOST_PEER_DISCONNECTED:
            // if device is out of range,
            // or connectivity(BT, Wi-Fi, and etc.) is turned off
            break;
        case CONNECTION_LOST_UNKNOWN_REASON:
            // this rarely happens, the error may be recoverable or not
            // you may want to call SAAgent.findPeerAgents(),
            // if found, you may want to re-connect
            yourSAAgentImpl.tryConnect();
            // in your implementation of that method
            // you should follow the procedures talked in Section 4.3
            // “Find Matching Peer Agent and Initiate Service Connection”
            break;
    }
}
```

Service Consumers and Service Providers are notified with the `SASocket.onError()` callback about errors related with Service Channels.

```
@Override
public void onError(int channelId, String errorMessage, int errorCode) {
    if(errorCode == ERROR_CONNECTION_CLOSED){
        Log.e(TAG, "Data not sent, Service Connection closed");
    }
}
```

Your applications are notified with the `SAAgent.onError()` callback about most other errors, such as errors related to the Samsung Accessory Service Framework and Accessory Peer Agents. For detailed error types, see the Accessory API Reference.

```
@Override
public void onError(SAPeerAgent peerAgent, String errorMessage, int errorCode) {
    switch(errorCode){
        case ERROR_SDK_NOT_INITIALIZED:
        SA accessory = new SA();
        try {
            accessory.initialize(this);
        } catch (SsdkUnsupportedException e) {
        e.printStackTrace();
        }
        break;
    case ERROR_FATAL:
           // Samsung Accessory Service Framework died or binding failure
           // Fatal error, you need to stop using Accessory
           break;
    case ERROR_CONNECTION_INVALID_PARAM:
           // data cleared by user(in Settings-> Application Manager-> Clear data)
           // or data lost for other reasons
           // not run-time recoverable errors, reboot needed,
           // you may want to exit the application
           break;
    }
}
```

Below are some transient errors due to Android environment:

1) Low memory

It is recommended to close all Service Connection in the `onLowMemory` callback of your `SAAgent` implementation (`onLowMemory` is an inherited method from Service.), to release caches.

If your application process is killed by Android Low Memory Killer (LMK), it will notify the `onServiceConnectionLost` callback. Your application or peer application should create Service Connection again upon restart.

2) Application crash or onDestory

If the application crashed from whatever reason, all Service Connections will be terminated. Upon restart of your application, it is your application's responsibility to restore the Service Connection.

When the `SAAgent` implementation is being removed by Android (will get `SAAgent.onDestroy()`), all Service Connections with the Accessory Peer Agent will be terminated.

3) Samsung Accessory Service Framework be killed

If Samsung Accessory Service Framework is killed on a local device, application will be notified with the ERROR_FATAL callback error code. You need to stop using Accessory.

4) Application stopSelf

It is strongly recommended to close Service Connections before the application stops itself. Calling stopSelf() notifies the Accessory Peer Agent in a graceful way. If stopSelf() is not called, all Service Connections will be terminated by Samsung Accessory Service Framework and then, both sides will get but the `onServiceConnectionLost` callback. Your application or peer application should create Service Connection again upon restart.

5. Guide for ProGuard

If application developer need to run ProGuard, please see the followings:

1. Upgrade the version of Eclipse in Android ADT to latest version.
2. The changes in project.properties file

   [Before]
   #proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt

   [After]
   proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt

3. Add the lines below *to proguard-project.txt* to exclude Accessory SDK for the proguard.

   - keepclassmembers class com.samsung.** { *; }
   - keep class com.samsung.** { *; }
   - dontwarn com.samsung.**

4. If you extend `SASocket`, create a new java file for creating class to extend `SASocket`. Do not use inner class to extend `SASocket`.

   - To avoid modifications in the inner class, ProGaurd option must be used by application developers following:

     - keepattributes InnerClasses

Copyright

Though every care has been taken to ensure the accuracy of this document, Samsung Electronics Co., Ltd. cannot accept responsibility for any errors or omissions or for any loss occurred to any person, whether legal or natural, from acting, or refraining from action, as a result of the information contained herein. Information in this document is subject to change at any time without obligation to notify any person of such changes.

Samsung Electronics Co. Ltd. may have patents or patent pending applications, trademarks copyrights or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give the recipient or reader any license to these patents, trademarks copyrights or other intellectual property rights.

No part of this document may be communicated, distributed, reproduced or transmitted in any form or by any means, electronic or mechanical or otherwise, for any purpose, without the prior written permission of Samsung Electronics Co. Ltd.

The document is subject to revision without further notice.

All brand names and product names mentioned in this document are trademarks or registered trademarks of their respective owners.

For more information, please visit http://developer.samsung.com/