Accessory

Programming Guide

Version 2.5.3

Table of Contents

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/

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.
- Support Accessory Message.

A glossary for the Accessory SDK is listed up in the following table.

Term	Description
Accessory Service Profile	An Accessory Service Profile defines the roles of Service Provider and Service Consumer. It also specifies the formats for application-level protocol messages and message sequences between Service Consumers and Service Providers. For example, the Notification Accessory Service Profile 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 application 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. For example, a notification Service Provider implemented on a Smart Device provides notifications from that Smart Device to interested Service Consumers on connected Accessory Devices.
Service Consumer	A Service Consumer is an application 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. For example, a notification Service Consumer implemented on an Accessory Device 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 application 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 a 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.

Accessory Message	An Accessory Message is provided to send or receive a message without establishing a Service Connection between a Service Provider and a Service Consumer. Both Service Provider and Service Consumer don't have to worry about the Service Connection and Service Channel. They only have to send a message to the desired Accessory Peer Agent.

Table 1: Glossary

1.1. Background

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 such as Samsung GEAR Manager. The Smart Devices and Accessory Devices described in this document have the Samsung Accessory Service Framework preloaded.

1.2. 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 functionalities. For example, the Notification Accessory Service Profile defines an application-level protocol to convey phone notifications to the connected Accessory Devices.

The Accessory provides the following features:

- Accessory Peer Agent
  - Getting the list of Peer Devices.
  - Getting the list of services offered by Peer Devices.
  - Identifying the available services between Peer Devices.
- Service Connection
  - Creating and storing the Service Connection between Peer Devices.
  - Initiating a Service Connection request.
  - Processing Service Connection requests from Peer Devices to provide or consume a service.
  - Closing a Service Connection.
- Accessory Message
  - Sending a message to a known peer device without Service Connection.
  - Receiving a message from a known peer device without Service Connection.
  - Acquiring the success acknowledgement or proper error codes in case of a failure.

The Accessory File Transfer uses the File Transfer Service to transfer files between devices. The file is transferred on a separate service connection.

The Accessory File Transfer provides the following features:

- Sends files to a known peer device.
- Queues file transfer requests from multiple applications.
- Receives incoming file transfer request notifications.
- Receives file transfer progress and completion updates.
- Receives proper error codes in case of a file transfer failure.
- Cancels an ongoing or scheduled file transfer.

1.3.Architecture

Applications such as Calendar Provider and Camera Consumer use Accessory as a facade. Accessory API communicates with the Samsung Accessory Service Framework that is 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.

The following figure shows the architecture of Accessory.

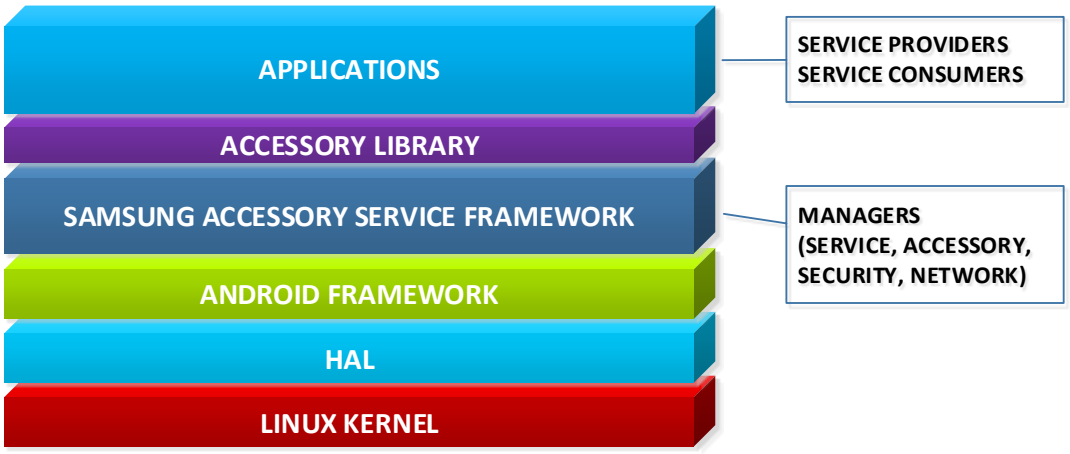


Figure 2: Accessory Architecture

Accessory Peer Agents like 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.

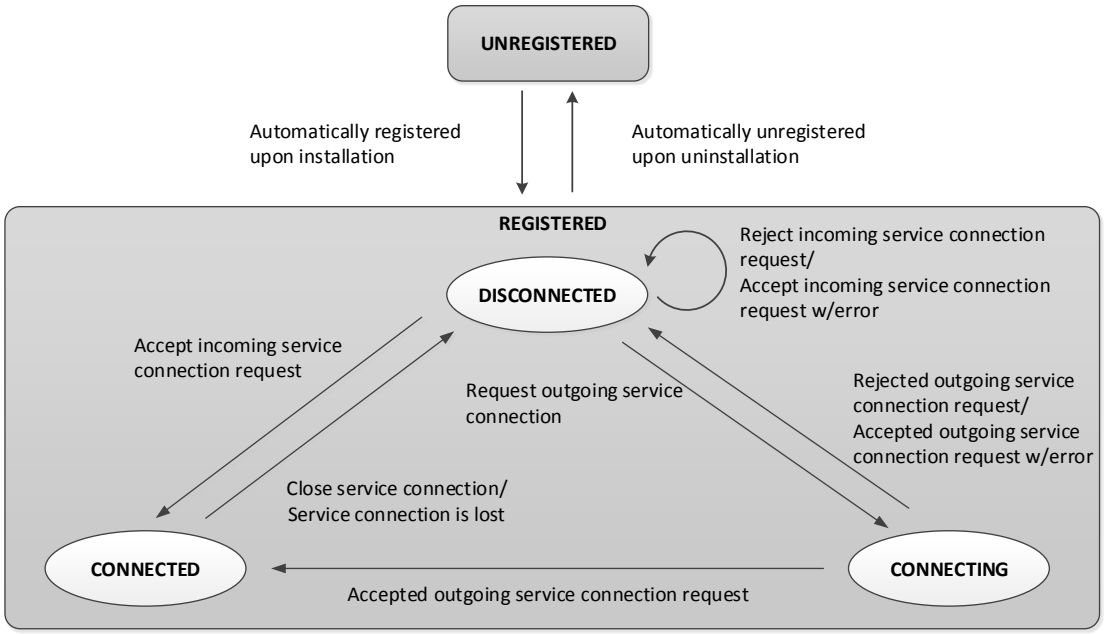


Figure 3: State Machine of Accessory Peer Agent

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 "*DISCONNECTED*" 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 has been 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.

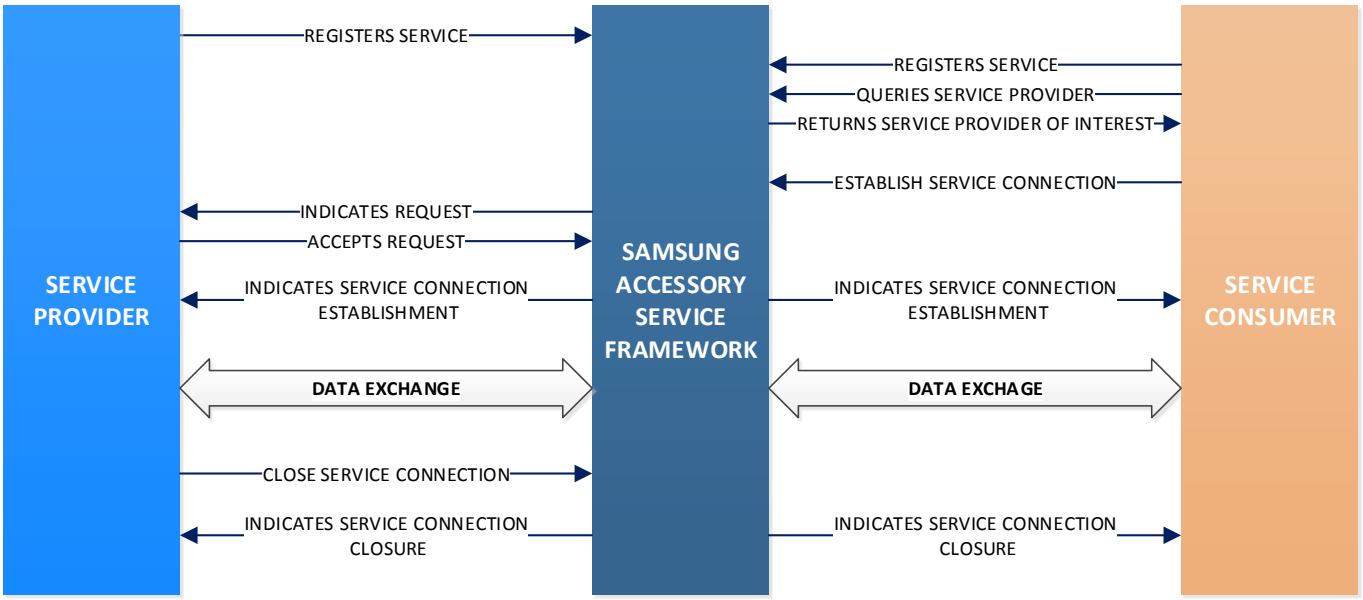


Figure 4: Sequence flow of Accessory Peer Agent

The above figure illustrates the following flow of Accessory Peer Agent:

The Service Provider and Service Consumer applications register their service capabilities with the Samsung Accessor y Service Framework. The Samsung Accessory Service Framework advertises and exchanges the capabilities of the reg istered 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 al so try to establish Service Connections with Service Consumers.

The Service Provider decides to accept or reject the Service Connection request. If the Service Provider attempts to e stablish a connection, the Service Consumer decides to accept or reject the Service Connection request.

The Service Connection is established once all the Service Channels defined by the associated Accessory Service Profil e are created. The Service Consumer and Service Provider use the established Service Connection to read and write d ata following the associated Accessory Service Profile specification on the Service Channels.

The following figure shows the relationship between classes and interfaces of the `com.samsung.android.sdk.accessory`. Detailed information for each class and interface can be found in the API Reference under `Accessory\Docs\API Reference` folder.

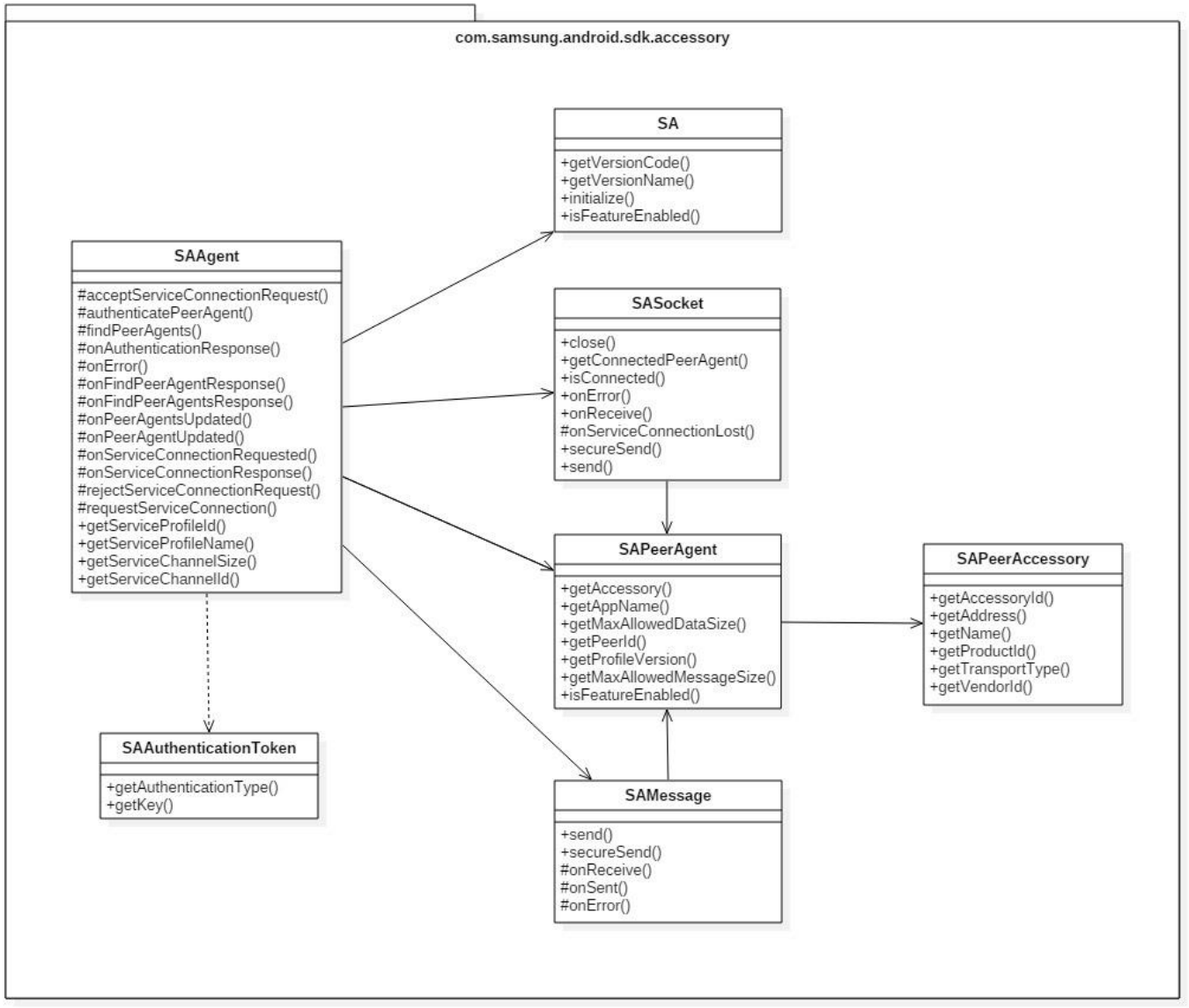


Figure 5: Accessory Class diagram

Its interfaces and classes are described in the following table.

Interface / Class	Description
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 Accessory Peer Agent sends an outgoing Service Connection request, your application is notified when the request result becomes available (with Service Connection establishment, a rejection by the remote Accessory Peer Agent, or due to a failure). Remote Accessory Peer Agents can also initiate Service Connection requests with Accessory Peer Agent. The application is expected to implement the method handling for incoming Service Connection requests and decide 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 send and receive data on established Service Channels according to the Accessory Service Profile specification.
SAMessage	Represents an instance for a Message Service between a Service Provider and a Service Consumer. Both the Service Consumer and Service Provider implementations extend this class to send and receive a message without established Service Channels by both applications.
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 includes information 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. It includes information 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.

Table 2: Accessory Interfaces and classes

The following figure shows the relationship between classes and interfaces of the `com.samsung.android.sdk.accessoryfiletransfer`. Detailed information for each class and interface can be found in the API Reference under the `Accessory\Docs\API Reference` folder.

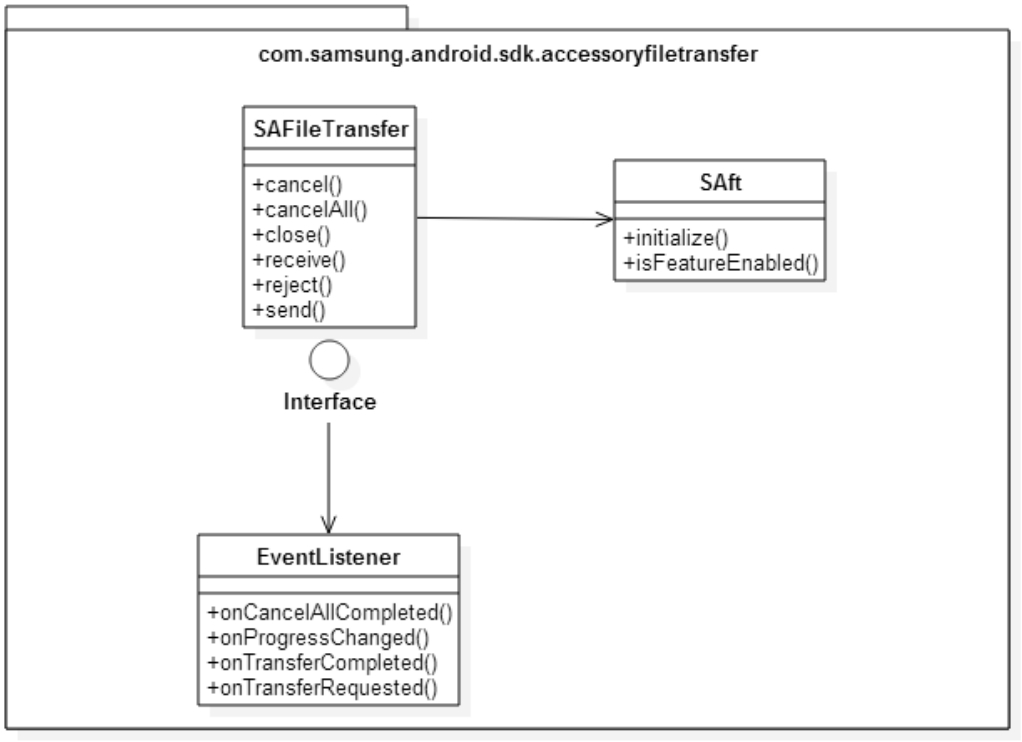


Figure 6: Accessory File Transfer Class diagram

Its interfaces and classes are described in the following table.

Interface / Class	Description
SAft	Initializes Accessory File Transfer.
SAFileTransfer	Provides the file transfer methods. Sending and receiving applications need to use the Accessory File Transfer class. Each SAAgent implementation can make their own SAFileTransfer object and call each method on it. This class also registers the SAAgent implementation using Accessory File Transfer and the SAFileTransfer.EventListener implementation where file transfer updates are notified.
SAFileTransfer.EventListener	Listens to file transfer update notifications.

Table 3: Accessory File Transfer Interfaces and classes

2. Development Environment

To develop applications with the Accessory SDK, check prerequisites for the SDK first and download the SDK.

2.1. Prerequisites

Before downloading the Accessory SDK and starting to develop an application, refer to the following information.

Android Version

Android 4.3 (API 18) or above

Available Devices

Android 4.3 or above devices

Limitations

Samsung Accessory Service Framework should be installed on the device before launching an application.

2.2. Downloading the Accessory SDK

The Accessory SDK can be downloaded in the Samsung developer site. If the downloaded SDK is unzipped, you can check the following content for the application development.

Folder in SDK	Description
Docs	API Reference with Javadoc Programming guide
Libs	**accessory-v2.5.3.jar** Provides Samsung Accessory SDK Library **sdk-v1.0.0.jar** Provides Samsung SDK Library
Samples	**Samples(Native)** Shows interactions between Smart device and Remote device(Native Application) using Accessory SDK **Samples(Web)** Shows interactions between Smart device and Remote device(Web Application) using Accessory SDK
Tools	**Application for Emulator** Helps to develop SAMSUNG GEAR application using GEAR IDE without actual devices

NOTE. **The android-support-v4 Library is necessary for using Accessory File Transfer.**

Table 4: SDK content

Emulator APK

This is used to develop SAMSUNG GEAR application using GEAR IDE without actual devices.

2.3.Using the Library

After adding the Accessory library under `libs` folder in your created application project, import the package in your code as below:

```
import com.samsung.android.sdk.accessory.*;
```

When using the Samsung Accessory File Transfer feature, import the necessary packages in your code as shown below:

```
import com.samsung.android.sdk.accessoryfiletransfer.*;
import com.samsung.android.sdk.accessoryfiletransfer.SAFileTransfer.*;
```

2.4.Permission

To use Accessory, it needs the permission below. If it is not added in the `AndroidManifest.xml` file, the initialization will fail with `SecurityException`.

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

If you don’t add the permission,

- For Samsung device,
    - Android 4.4.2 (KitKat) and above: `SecurityException` is thrown and your application won’t work.
    - Prior to Android 4.4.2 (KitKat): No exception and the application will work properly.
- For other companies,
    - No exception and the application will work properly.

The following permissions have to be specified in the `AndroidManifest.xml` file to use Samsung Accessory Service:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="com.samsung.accessory.permission.ACCESSORY_FRAMEWORK" />
```

2.5.Tools

The Accessory SDK provides tools for its application development.

Emulator

This is used to develop SAMSUNG GEAR applications using GEAR IDE without actual devices.

3. Accessory

3.1. Hello Accessory

`Hello Accessory` is a pseudo code example to show how to:

- Initialize Accessory.
- Connect with the remote Accessory Peer Agent.
- Send and receive JSON messages between Accessory Peer Agents.

For more information about sample applications, please visit
http://developer.samsung.com/gear/develop/samples/companion

`Hello Accessory` is composed of two parts: Consumer and Provider.

3.1.1. Consumer Application

Consumer application has the functionalities below

- Initiates service connection request and sends command to Service Provider.
- Shows a received response to user.

```
class HelloAccessoryConsumer extends SAAgent {
...
    void onCreate() {
        Create SA;
        try {
            Initialize SA;
        } catch (Exception e) {
            // Error Handling
        }
    }

    void onStart() {
        // Find Peer Agent
        FindPeerAgent();
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Store found Peer Agent if success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
            RequestServiceConnection(peerAgent);
        }
    }

    void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket socket, int result) {
        // if result is successful, cache socket for using on sending message
        Cache(socket);
        Create WorkerThread(
```

```
            try {
                message = composeMessage();
                Send(channel id, message);
            } catch (Exception e) {
                // Error handling
            }
        );
    }

    class ServiceConnection extends SASocket {
        public void onReceive(int channelId, byte[] data) {
            // Check received data
            Parse(data);
            // Create a worker thread and show message to user
            Create WorkerThread(
                Show(message);
            );
        }

        void onServiceConnectionLost(int errorCode) {
            // Reset cached peer agent and close service connection
            ResetCache();
            Close();
        }

        void onError(int channelId, String errorString, int error) {
            // Error handling
        }
    }
...
}
```

3.1.2.Provider Application

Provider application has the functionalities below

- Accepts a received service connection request from Service Consumer.
- Replies to a received command from Service Consumer with current time stamp.

```
class HelloAccessoryProvider extends SAAgent {
...
    void onCreate() {
        Create SA;
        try {
            Initialize SA;
        } catch (Exception e) {
            // Error Handling
        }
    }

    void onStart() {
        // Find Peer Agent
        FindPeerAgent();
```

```
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Store found Peer Agent if success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
        }
    }

    void onServiceConnectionRequested(SAPeerAgent peerAgent) {
        // Received service connection request from remote, decide whether to accept or to reject.
        Accept(peerAgent);
    }

    void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket socket, int result) {
        // if result is successful, cache socket for using on sending message
        Cache(socket);
    }

    class ServiceConnection extends SASocket {
        void onReceive(int channelId, byte[] data) {
            // Check received data
            Parse(data);
            // Create a worker thread and send message to Consumer
            Create WorkerThread(
                message = composeMessage();
                CachedSocket.Send(channel id, message);
            );
        }

        void onServiceConnectionLost(int errorCode) {
            // Reset cached peer agent and close service connection
            ResetCache();
            Close();
        }

        void onError(int channelId, String errorString, int error) {
            // Error handling
        }
    }
...
}
```

3.2. 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 sa = new SA();
try {
    sa.itialize(applicationContext) {
    boolean isFeatureEnabled = sa.isFeatureEnabled(SA.DEVICE_ACCESSORY);
} catch (final SsdkUnsupportedException e) {
   // try to handle SsdkUnsupportedException
   if (e.getType() == SsdkUnsupportedException.LIBRARY_NOT_INSTALLED) {
       // You should install service application first.
   }
} catch (Exception e1) {
    // Your application cannot use Accessory. Your application should work smoothly without
    // using Accessory, or you may want to notify the user and close your application
    // gracefully (release resources, stop Service threads, close UI thread, etc.)
   return;
}

int versionCode = sa.getVersionCode();
String versionName = sa.getVersionName();
```

3.2.1.Using initialize()

The `SA.initialize()` method:

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

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

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

3.2.2.Handling SsdkUnsupportedException

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

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

- `LIBRARY_NOT_INSTALLED`: The Samsung Accessory Service Framework is not installed on the device.

3.2.3.Checking the Availability of Accessory

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

The following type is defined in the SA class:

- DEVICE_ACCESSORY

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

3.3.Using Accessory

The following chapter describes how to use Accessory.

3.3.1.Declaring Broadcast Receiver

Communicating with the remote Peer Agent needs the broadcast receiver below. If it is not added in the AndroidManifest.xml file, any intent handled by Samsung Accessory Service Framework is not delivered to the developer's created application.

```
<application>
  ...
  <receiver
  android:name="com.samsung.android.sdk.accessory.ServiceConnectionIndicationBroadcastReceiver">
    <intent-filter>
        <action android:name="com.samsung.accessory.action.SERVICE_CONNECTION_REQUESTED" />
    </intent-filter>
  </receiver>
  <receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver">
    <intent-filter>
        <action android:name="com.samsung.accessory.action.REGISTER_AGENT" />
    </intent-filter>
  </receiver>
  ...
</application>
```

***NOTE*.** Name of actions are changed from 2.3.0. The old actions are not anymore supported in the latest Accessory SDK.

Communicating with the remote Peer Agent needs the declaration of a service in the AndroidManifest.xml. This ensures that the application is derived from the class SAAgent.

```
<application>
  ...
  <service android:name="the class name that extends SAAgent" />
```

```
  ...
</application>
```

NOTE. 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 the developer's created application main thread.

3.3.2.Defining Accessory Service Profile

Communicating with remote Peer Agent needs the declaration of descriptions about Accessory Service Profile. This is declared in a separate file in /res/xml folder in the developer's application project. The path of the actual XML file can be added in the application's AndroidManifest.xml.

For example, /res/xml/<profileName>.xml:

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

Its elements and attributes are described in the following table.

Element	Attribute	Description
application	name	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. ***NOTE.*** This attribute allows up to 30 characters.
serviceProfile	serviceImpl	The subclass that extends SAAgent.
	role	The direction to serve an associated service to Accessory Peer Agents. ***NOTE.*** The value must be either "provider" or "consumer"
	name	The name of your Service Provider or Service Consumer. ***NOTE.*** This attribute allows up to 30 characters.
	id	The Service Profile ID of the Service Provider or Service Consumer. ***NOTE.*** It is necessary to start with '/'. It then allows [0-9], [a-z], '_' and '/' (as delimiter). This attribute allows up to 30 characters.
	version	The Service Profile specification version that your Service Provider or Service Consumer application supports. This attribute is represented as a two-part string with the following format: <major>.<minor>. ***NOTE.*** The major version and minor version have a maximum value of 255.
	serviceLimit*	The number of Accessory Peer Agents that 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 : Can have Service Connections to multiple Accessory Peer Agents on an Accessory Device **any** : Supports multiple Accessory Peer Agents on multiple Accessory Devices ***NOTE.*** If you do not set the value, "***any***" is applied by default.
	serviceTimeout*	The timeout in seconds 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 more time to make a decision to accept or reject incoming Service Connection requests. If it is needed, e.g., in cases when it needs to connect to a cloud server, show a UI prompting the user to either accept or reject the request. On the other hand, if 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. ***NOTE.*** This attribute allows up to 300 seconds.
serviceProfile supportedTransports	transport	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 TRANSPORT_BT and TRANSPORT_WIFI (only for Emulator). Other types will be supported soon.
serviceChannel	dataRate	The throughput at which data traffic originated from the Accessory Agent. ***NOTE.*** The value must be either "low" or "high".
	priority	The basis on which the application prioritizes transmissions of messages to Accessory Peer Agent. ***NOTE.*** The value must be either "low", "medium", or "high".
	reliability	The basis on which the application can have a reliable transfer or not. In case of a packet drop, a reliable transfer re-transmits the packet but also creates additional overhead. ***NOTE.*** The value must be either "enable" or "disable".

NOTE. Optional attributes are denoted with an asterisk (*).

Table 5: Defining Accessory Service Profile

An example of Accessory Service Profile XML:

```
<resources>
    <application name="ProviderExample">
        <serviceProfile
            serviceImpl="com.samsung.accessory.example.providerServiceImpl"
            role="provider"
            name="ExampleService"
            id="/app/example"
            version="1.0"
            serviceLimit="any"
            serviceTimeout="10">
            <supportedTransports>
                <transport type="TRANSPORT_BT"/>
            </supportedTransports>
            <serviceChannel
                id="910"
                dataRate="low"
                priority="high"
                reliability="enable"/>
        </serviceProfile>
    </application>
</resources>
```

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. To define the Accessory Service Profile, refer to **A.2.**

3.3.3.Finding Accessory Peer Agents

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”.

The application searches for matching Peer Agents by calling `SAAgent.findPeerAgents()`. If matching Peer Agents are found, it is notified by the function called `SAAgent.onFindPeerAgentResponse()`. If a matching Peer Agent is not found, it is notified with the same callback. The result will have a null Peer Agent and the reason why there’s no match found.

```
@Override
protected void onFindPeerAgentResponse(SAPeerAgent peerAgents[], int result) {
    ...
    switch(result) {
        case PEER_AGENT_FOUND:
            // Peer Agent is found
            ...
            break;
        case FINDPEER_DEVICE_NOT_CONNECTED:
            // Peer Agents are not found, no accessory device connected
            ...
            break;
```

```
        case FINDPEER_SERVICE_NOT_FOUND:
            // No matching service on connected accessory
            ...
            break;
        ...
    }
}
```

3.3.4.Setting up Service Connection

If the 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 with. The application can initiate a Service Connection with an Accessory Peer Agent by calling SAAgent.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.

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 called 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 SAAgent.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.

The remote Accessory Peer Agent either accepts or rejects your Service Connection request. Your application is notified with the SAAgent.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 onFindPeerAgentResponse(SAPeerAgent peerAgents[], int result) {
    ...
    switch(result) {
        case PEER_AGENT_FOUND:
            // Peer Agent is found
            requestServiceConnection(peerAgent);
            break;
        case FINDPEER_DEVICE_NOT_CONNECTED:
            // Peer Agents are not found, no accessory device connected
            ...
            break;
        case FINDPEER_SERVICE_NOT_FOUND:
            // No matching service on connected accessory
            ...
            break;
        ...
    }
}

SASocket mSocket = null;
```

```
@Override
protected void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket socket, int result) {
    if(result == CONNECTION_SUCCESS) {
        // It is passed when a Service Connection has been established.
        mSocket = socket;
    }
    ...
}
```

NOTE. If setting up Service Connection failed continuously, it is your application's responsibility to call `SAAgent.findPeerAgents()` to try to re-find the remote Accessory Peer Agent and `SAAgent.requestServiceConnection()` to make Service Connection request again.

3.3.5.Handling Setup Service Connection Request

The Service Provider or Consumer application is notified with the `SAAgent.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()` methods, respectively. The default implementation of the `SAAgent.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 `SAAgent.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.

If your application accepts the Service Connection request, your application is notified through the `SAAgent.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 `SAAgent.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.

```
@Override
protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
    // Makes a decision after checking the validation of given information.
    String vendorId = peerAgent.getAccessory().getVendorId();
    String productId = peerAgent.getAccessory().getVendorId();
    if (vendorId.equals("SAMSUNG ELECTRONICS") && productId.equals("SAMSUNG GEAR")) {
        // If connected accessory is the right device
        acceptServiceConnectionRequest(peerAgent);
    } else {
        // If connected accessory is not the right device
        rejectServiceConnectionRequest(peerAgent);
    }
    ...
```

```
}

SASocket mSocket = null;

@Override
protected void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket socket, int result)
{
    ...
    switch(result) {
        case CONNECTION_SUCCESS:
            // Peer Agent is found, store it for sending data
            mSocket = socket;
            break;
        case CONNECTION_FAILURE_NETWORK:
            // Try to request service connection again after a while
            ...
            break;
        case CONNECTION_ALREADY_EXIST:
            // Previous Service Connection is alive. Reuse it
            ...
            break;
        case CONNECTION_FAILURE_PEER_AGENT_REJECTED:
            // Peer Agent Rejected. Try to request service connection again after a while
            ...
            break;
        case CONNECTION_FAILURE_PEER_AGENT_NO_RESPONSE:
            // Peer Agent no response. Try to request service connection again after a while
            ...
            break;
        case CONNECTION_FAILURE_DEVICE_UNREACHABLE:
            // Accessory Device not reachable, may already be disconnected
            ...
            break;
        default:
            // Service Connection Fail, non-recoverable error
            ...
            break;
        ...
    }
    ...
}
```

3.3.6.Exchanging Data with Accessory Peer Agent

Call the `SASocket.send()` method of the `SASocket` object passed with the `SAAgent.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. You can get the `Channel ID` from SAAgent.getServiceChannelId().

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

```
try {
    mSocket.send(channel id, mJsonStringToSend.getBytes());
} catch (IOException e) {
    // Handle exception
...
}
```

NOTE. `SASocket.send()` and `SASocket.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 in the main thread of the application.

If you want your data encrypted, call `SASocket.secureSend()` instead of `SASocket.send()`.

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

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

3.3.7.Disconnecting Service Connection

Call the `SASocket.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 `SASocket.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 `SASocket.close()` to terminate the Service Connection, your application is notified with the same callback.

```
public boolean closeConnection() {
    if (mSocket != null) {
        mSocket.close();
        mSocket = null;
    }
    return true;
}

@Override
public void onServiceConnectionLost(int reason) {
    // This function is called when Service Connection is broken or lost
    // or there is a peer disconnection.
    switch (reason) {
        case CONNECTION_LOST_DEVICE_DETACHED:
            // If the Peer Agent is killed because of LMK OOM, call SAAgent.findPeerAgents()
            // and request Service Connection. Accessory will invoke Peer Agent in your method
            // implementation. You should follow the procedures in “Finding Peer Agents” and
            // “Setting Up 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:
            // Though it rarely happens, the error may be recoverable or not.
            // You may want to call SAAgent.findPeerAgents().
            // If found, you may want to re-connect in your method implementation.
            // you should follow the procedures in “Finding Peer Agent” and
            // “Setting Up Service Connection”.
            ...
            break;
    }
    ...
}
```

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 `SASocket.onServiceConnectionLost()` callback. However, it is not necessary to close in the `SASocket.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 `SAAgent.findPeerAgents()` to try to re-find the remote Accessory Peer Agent and `SAAgent.requestServiceConnection()` to make Service Connection request again.

3.3.8.Handling Errors

Application is notified with the `SAAgent.onError()` callback about errors related with Service Channels, Accessory Peer Agents and Samsung Accessory Service Framework. For detailed error types, see the API reference.

```
@Override
public void onError(SAPeerAgent peerAgent, String errorMessage, int errorCode) {
switch (errorCode){
    case ERROR_CONNECTION_INVALID_PARAM:
        // Data cleared by user(in Settings-> Application Manager-> Clear data)
        // or data lost for other reasons except run-time recoverable errors and reboot is
        // needed, you may want to exit the application.
        break;
    case ERROR_FATAL:
        // Samsung Accessory Service Framework died or binding failure
        // Fatal error, you need to stop using Accessory
        break;
    case ERROR_PERMISSION_DENIED:
        // Required permission missed, check the AndroidManifest.xml
        break;
    case ERROR_PERMISSION_FAILED:
        // Permission failure when application is installed before Samsung Accessory Service
        // Framework is installed. Reinstallation of the application might be needed
        break;
     case ERROR_SDK_NOT_INITIALIZED:
       // Samsung Accessory SDK is not initialized
```

```
        // It's necessary to install Samsung Accessory Service Framework and call SA.initialize()
        break;
    }
    ...
}
```

Below are some transient errors due to Android environment:

- 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 `SASocket.onServiceConnectionLost()` callback. Your application or peer applications should create Service Connection again upon restart.
- Application crash or `onDestroy()`
  - If the application crashed from whatever reason, all Service Connections will be terminated. Upon restart, 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.
  - If your application has cashed references of `SAPeerAgent` object, they can be cleared in using `SAAgent.onDestory()`. Your application can refresh `SAPeerAgent` object with a fresh call to `SAAgent.findPeerAgents()` when you application is restarted.
- SAMSUNG ACCESSORY SERVICE be killed
  - If *SAMSUNG ACCESSORY SERVICE* is killed on a local device, application will be notified with an `ERROR_FATAL` callback error code. In this case, your application needs to stop using Accessory. After restoring *SAMSUNG ACCESSORY SERVICE*, it will automatically make application ready to set up a service connection via the broadcast receiver.
- 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 *SAMSUNG ACCESSORY SERVICE* and both sides will receive the `SASocket.onServiceConnectionLost()` callback. Your application or peer application should find the remote Accessory Peer Agent and create Service Connection again upon restart.

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

3.3.9.Indicating the status of Accessory Peer Agent

After you call `SAAgent.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 `SAAgent.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 `SAAgent.findPeerAgents()`, the `SAAgent.onFindPeerAgentResponse()` callback gets a failure code. When it becomes available, you can get the `PEER_AGENT_AVAILABLE` from SAAgent.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) {
        // Peer Agent no longer available
    }
    ...
}
```

3.3.10.Authenticating Accessory Peer Agent

The `SAAgent.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 whether to accept or reject its Service Connection request.

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

```
@Override
protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
    // Check Peer Agent's basic info
    if(peerAgent.getAccessory().getVendorId().equals("SAMSUNG ELECTRONICS")
            && peerAgent.getAccessory().getProductId().equals("SAMSUNG GEAR")){
        // Authenticate Peer Agent for enhanced security
        authenticatePeerAgent(peerAgent);
    } else {
        rejectServiceConnectionRequest(peerAgent);
    }
}

@Override
protected void onAuthenticationResponse(SAPeerAgent peerAgent, SAAuthenticationToken authToken, int
code) {
    ...
    // Get the certificate from context
    byte[] myAppKey = getApplicationCertificate(mContext);

    // Compare it to certificate received from remote peer.
    if (authToken.getKey().length != myAppKey.length) {
        matched = false;
    } else {
        for (int i = 0; i < authToken.getKey().length; i++) {
            if (authToken.getKey()[i] != myAppKey[i]) {
                matched = false;
            }
        }
    }
    // if identical, do further work like accept service connection request
```

```
    ...
}
```

NOTE. Due to platform difference, it's necessary that Gear App creates author certificate using Android keystore. Please refer to ***Appendix D. Creating Gear Author Certificate Using Android Keystore*****.** It's also necessary to sign Android App with Android Keystore and Gear App with author certificate created from Android Keystore.

4. Accessory Message

4.1. Hello Message

Hello Message is a pseudo code example to show how to:

- Initialize SA.
- Create a SAMessage.
- Send and receive messages between Accessory Peer Agents more simply.

For more information about sample applications, please visit
http://developer.samsung.com/gear/develop/samples/companion

`Hello Message` is composed of two parts: Consumer and Provider.

> ***NOTICE.*** Please note that the previous Gear Series before Gear S3 do not support Accessory Message feature. In addition, it needs to use the legacy Service Connection if the application intends to send or receive heavy messages since this feature is applicable to simple message delivery.

4.1.1. Consumer Application

Consumer application has the functionalities below:

- Creates a SAMessage in your subclass extending SAAgent.
- Sends a message to Service Provider.
- Receives success or failure result of sending a messge.

```
class HelloMessageConsumer extends SAAgent {
...
    void onCreate() {
        Create SA;
        try {
            Initialize SA;
        } catch (Exception e) {
            // Error Handling
        }
        Create SAMessage(this) {
            @Override
            protected void onSent(SAPeerAgent peerAgent, int id) {
                // Success to send a message
            }

            @Override
            protected void onError(SAPeerAgent peerAgent, int id, int errorCode) {
                // Failure to send a message
            }
```

```
            @Override
            protected void onReceive(SAPeerAgent peerAgent, byte[] message) {
                // Receive a message
                Parse(message);
            }
        };

    }

    void onStart() {
        // Find Peer Agent
        FindPeerAgent();
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Store found Peer Agent if success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
        }
    }

    public int sendData(SAPeerAgent peerAgent, String value) {
        // Send a message
        Create WorkerThread(
            try {
                return SAMessage.send(peerAgent, value.getBytes());
            } catch (Exception e) {
                // Error Handling
                return -1;
            }
        );
    }
...
}
```

4.1.2.Provider Application

Provider application has the functionalities below:

- Creates a SAMessage in your subclass extending SAAgent.
- Receives a message from Service Provider.

```
class HelloMessageProvider extends SAAgent {
...
    void onCreate() {
        Create SA;
        try {
            Initialize SA;
        } catch (Exception e) {
            // Error Handling
        }
        Create SAMessage(this) {
            @Override
            protected void onSent(SAPeerAgent peerAgent, int id) {
                // Success to send a message
            }
```

```
            @Override
            protected void onError(SAPeerAgent peerAgent, int id, int errorCode) {
                // Failure to send a message
            }

            @Override
            protected void onReceive(SAPeerAgent peerAgent, byte[] message) {
                // Receive a message
                // Check received message
                 Parse(message);
                 // Create a worker thread and show message to user
                 Create WorkerThread(
                     Show(message);
                 );

            }
        };
    }

    void onStart() {
        // Find Peer Agent
        FindPeerAgent();
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Store found Peer Agent if success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
        }
    }
...
}
```

4.2.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 sa = new SA();
try {
   sa.itialize(applicationContext) {
   boolean isFeatureEnabled = sa.isFeatureEnabled(SA.DEVICE_ACCESSORY);
} catch (final SsdkUnsupportedException e) {
   // try to handle SsdkUnsupportedException
   if (e.getType() == SsdkUnsupportedException.LIBRARY_NOT_INSTALLED) {
       // You should install service application first.
```

```
    }
 } catch (Exception e1) {
    // Your application cannot use Accessory. Your application should work smoothly without
    // using Accessory, or you may want to notify the user and close your application
    // gracefully (release resources, stop Service threads, close UI thread, etc.)
    return;
}

 int versionCode = sa.getVersionCode();
String versionName = sa.getVersionName();
boolean isMessageEnabled = sa.isFeatureEnabled(SA.SERVICE_MESSAGE);
```

4.2.1. Using initialize()

The SA.initialize() method:

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

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

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

4.2.2. Handling SsdkUnsupportedException

If an SsdkUnsupportedException exception is shown, check the exception message type using SsdkUnsupportedExcpetion.getType().

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

- LIBRARY_NOT_INSTALLED: The Samsung Accessory Service Framework is not installed on the device.

4.2.3. Checking the Availability of Accessory Message

The application can check if the Accessory Message feature is supported on own device with the SA.isFeatureEnabled() method and on the remote Peer Agent with the SAPeerAgent.isFeatureEnabled(). The feature types are defined in the SA class. The feature type is passed as a parameter when calling the SA.isFeatureEnabled() method or calling the SAPeerAgent.isFeatureEnabled(). The method returns a Boolean value that indicates the support for the feature on the device.

The following type and method are defined in the SA and SAPeerAgent class:

- SERVICE_MESSAGE

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

4.3.Using Accessory Message

The following chapter describes how to use Accessory Message.

4.3.1. Declaring Broadcast Receiver

Communicating with the remote Peer Agent needs the broadcast receiver below. If it is not declared in the `AndroidManifest.xml` file, any intent handled by Samsung Accessory Service Framework is not delivered to the developer's created application.

```
<application>
  ...
  <receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver">
    <intent-filter>
        <action android:name="com.samsung.accessory.action.REGISTER_AGENT" />
    </intent-filter>
  </receiver>
  <receiver android:name="com.samsung.android.sdk.accessory.MessageReceiver">
    <intent-filter>
       <action android:name="com.samsung.accessory.action.MESSAGE_RECEIVED" />
    </intent-filter>
  </receiver>

  ...
</application>
```

Communicating with the remote Peer Agent needs the declaration of a service in the `AndroidManifest.xml`. This ensures that the application is derived from the class `SAAgent`.

```
<application>
  ...
  <service android:name="the class name that extends SAAgent" />
  ...
</application>
```

NOTE. 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 the developer's created application main thread.

4.3.2. Defining Accessory Service Profile

Communicating with remote Peer Agent needs the declaration of descriptions about Accessory Service Profile. This is declared in a separate file in /res/xml folder in the developer's application project. The path of the actual XML file can be added in the application's `AndroidManifest.xml`.

For example, /res/xml/<profileName>.xml:

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

Its elements and attributes are described in the following table:

Element	Attribute	Description
application	name	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. ***NOTE.*** This attribute allows up to 30 characters.
serviceProfile	serviceImpl	The subclass that extends SAAgent.
	role	The direction to serve an associated service to Accessory Peer Agents. ***NOTE.*** The value must be either “provider” or “consumer”
	name	The name of your Service Provider or Service Consumer. ***NOTE.*** This attribute allows up to 30 characters.
	id	The Service Profile ID of the Service Provider or Service Consumer. ***NOTE.*** It is necessary to start with ‘/’. It then allows [0-9], [a-z], ‘_’ and ‘/’ (as delimiter). This attribute allows up to 30 characters.
	version	The Service Profile specification version that your Service Provider or Service Consumer application supports. This attribute is represented as a two-part string with the following format: <major>.<minor>. ***NOTE.*** The major version and minor version have a maximum value of 255.
serviceProfile supportedTransports	transport	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 TRANSPORT_BT and TRANSPORT_WIFI (only for Emulator). Other types will be supported soon.
serviceProfile supportedFeatures	feature	The features on which the Service Provider or Service Consumer is able to operate. The Samsung Accessory Service Framework supports the message type. ***NOTE.*** The current version of the Samsung Accessory Service Framework supports only message feature. Other types will be supported in the future.

Table 6: Defining Accessory Service Profile

An example of Accessory Service Profile XML:

```
<resources>
    <application name="ProviderExample">
        <serviceProfile
            serviceImpl="com.samsung.accessory.example.providerServiceImpl"
            role="provider"
            name="ExampleService"
            id="/app/example"
            version="1.0">
            <supportedTransports>
                <transport type="TRANSPORT_BT"/>
            </supportedTransports>
            <supportedFeatures>
                <feature type="message"/>
            </supportedFeatures>
        </serviceProfile>
    </application>
</resources>
```

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.

4.3.3. Finding Accessory Peer Agents

Please refer to 3.3.3.

4.3.4. Creating Message Sender

Creating a SAMessage instance

The application must have a `SAAgent` subclass and a `SAMessage` instance using `SAAgent` subclass and the instance must include all implemented methods. The following example shows this implementation.

```
SAMessage mSAMessage = new SAMessage(this) {

    @Override
    protected void onSent(SAPeerAgent peerAgent, int id) {
        // Success to send a message
    }

    @Override
    protected void onError(SAPeerAgent peerAgent, int id, int errorCode) {
        // Failure to send a message
    }

    @Override
    protected void onReceive(SAPeerAgent peerAgent, byte[] message) {
        // Receive a message
    }
};
```

Sending a Message

To send a message, the application must decide who is the desired Accessory Peer Agent.

The Accessory Peer Agent can be obtained either by calling `SAAgent.findPeerAgents()` or by using the connected peer. The application must then call `SAMessage.send()` on the `SAMessage` object. `SAMessage.send()` will return integer value if it has succeeded to enqueue the message to Accessory Framework.

```
int id = mSAMessage.send(peerAgent, message);
```

NOTE. `SAMessage.send()` and `SAMessage.secureSend()` methods have to be 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 in the main thread of the application.

If you want your data encrypted, call `SAMessage.secureSend()` instead of `SAMessage.send()`.

Handling Success or Failure

Application is notified with the `SAMessage.onSent()` callback with the corresponding Accessory Peer Agent and the ID which was returned when calling `SAMessage.send()` or `SAMessage.secureSend()` if it has succeeded on sending a message. Otherwise, it is notified with the `SAMessage.onError()` callback about errors related with Accessory Peer Agents and Samsung Accessory Service Framework. For detailed error types, see the API reference.

```
@Override
protected void onSent(SAPeerAgent peerAgent, int id) {
    // Success to send a message
}

@Override
protected void onError(SAPeerAgent peerAgent, int id, int errorCode) {
    switch (errorCode){
    case ERROR_PEER_AGENT_UNREACHABLE:
        // The remote Accessory Peer Agent is not
        // reachable or not alive to receive any messages.
        break;
    case ERROR_PEER_AGENT_NO_RESPONSE:
        // The remote Accessory Peer Agent does not give any response
        // within the timeout period.
        break;
    case ERROR_PEER_AGENT_NOT_SUPPORTED:
        // The remote Accessory Peer Agent does not support message feature.
        // It needs to establish legacy service connection.
        break;
    case ERROR_PEER_SERVICE_NOT_SUPPORTED:
        // Samsung Accessory Service Framework on remote device does not support message feature.
        // It needs to establish legacy service connection.
        break;
    case ERROR_SERVICE_NOT_SUPPORTED:
        // Samsung Accessory Service Framework on this device does not support message feature.
        // It needs to establish legacy service connection.
        break;
    case ERROR_UNKNOWN:
```

```
        // The remote Accessory Peer Agent has some internal error which occurred
        // on the remote device.
        break;
    }
    ...
}
```

NOTE. `SAMessage.send()` and `SAMessage.secureSend()` methods have to be 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 in the main thread of the application.

If you want your data encrypted, call `SAMessage.secureSend()` instead of `SAMessage.send()`.

4.3.5. Creating Message Receiver

Creating a SAMessage instance

The application must have a `SAAgent` subclass and a `SAMessage` instance using `SAAgent` subclass and the instance must include all implemented methods. The following example shows this implementation.

```
SAMessage mSAMessage = new SAMessage(this) {

    @Override
    protected void onSent(SAPeerAgent peerAgent, int id) {
        // Success to send a message
    }

    @Override
    protected void onError(SAPeerAgent peerAgent, int id, int errorCode) {
        // Failure to send a message
    }

    @Override
    protected void onReceive(SAPeerAgent peerAgent, byte[] message) {
        // Receive a message
    }
};
```

Receiving a Message

When your application receives a message from a remote Accessory Peer Agent, it is notified with the `SAMessage.onReceive()` callback. If your application cannot receive any message, please check if there is a receiver filter of `"com.samsung.accessory.action.MESSAGE_RECEIVED"` in the `AndroidManifest.xml` file.

```
@Override
protected void onReceive(SAPeerAgent peerAgent, byte[] message) {
    // Receive a message
    String str = new String(message);
    ...
}
```

5. Accessory File Transfer

5.1. Hello AccessoryFileTransfer

`Hello Accessory File Transfer` is a pseudo code example to show how to:

- Initialize Accessory File Transfer.
- Create `SAFileTransfer`.
- Send a file from `Sender` to `Receiver`.

For more information about sample applications, please visit
http://developer.samsung.com/gear/develop/samples/companion

`Hello Accessory File Transfer` is composed of two parts: Sender and Receiver.

5.1.1. Sender Application

Sender application has the functionalities below:

- Sends a file to Receiver.

```
class HelloFileTransferSender extends SAAgent {
    ...
    void onCreate() {
        Create SAft;
        try {
            Initialize SAft;
        } catch (Exception e) {
            // Error Handling
        }
    }

    void onStart() {
        // Find Peer Agent
        FindPeerAgent();
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Store found Peer Agent if success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
            // Send a file to found Peer Agent
            Create SAFileTransfer(EventListner);
            transId = Send(peerAgent, filename);
        }
    }

    class EventListener() {

        void onProgressChanged(int transId, int progress) {
            // Show the progress
            Show(progress);
        }
```

```
        void onTransferCompleted(int transId, String fileName, int errorCode) {
        // Clean up resources if no more usage
        cleanup();
        }
    };
    ...
}
```

5.1.2.Receiver Application

Receiver application has the functionalities below:

- Receives a file from Sender.

```
public class HelloFileTransferReceiver extends SAAgent {
    ...
    void onCreate() {
        Create SAft;
        try {
            Initialize SAft;
        } catch (Exception e) {
            // Error Handling
        }
    }

    void onStart() {
        // Find Peer Agent
        FindPeerAgent();
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Store found Peer Agent if success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
            // Send a file to found Peer Agent
            Create SAFileTransfer(EventListner);
        }
    }

    class EventListener() {

        void onTransferRequested(int transId, String fileName) {
            // Receive a file from found Peer Agent
            receive(transId, fileName);
        };

        void onProgressChanged(int transId, int progress) {
            // Show the progress
            Show(progress);
        }

        void onTransferCompleted(int transId, String fileName, int errorCode) {
            // Clean up resources if no more usage
            cleanup();
        }
    };
    ...
}
```

5.2.Using the SAft Class

The SAft class provides the following methods:

- `initialize()` initializes Accessory File Transfer. You need to initialize Accessory File Transfer before you can use it. If the device does not support Accessory, SsdkUnsupportedException is thrown.
- `isFeatureEnabled()` checks if the Accessory File Transfer feature is available on the device.

```
SAft saft = new SAft();

try {
    saft.initialize(this) {
    boolean isFeatureEnabled = saft.isFeatureEnabled(SAft.DEVICE_ACCESSORY);
} catch (final SsdkUnsupportedException e) {
    // try to handle SsdkUnsupportedException
    if (e.getType() == SsdkUnsupportedException.LIBRARY_NOT_INSTALLED) {
        // You should install service application first
    }
} catch (Exception e1) {
    /* Your application cannot use Accessory File Transfer
     * Your application should work smoothly without using Accessory File Transfer,
     * or you may want to notify the user and close your application gracefully
     * (release resources, stop Service threads, close UI thread, etc.)
     */
    return;
}
```

5.2.1. Using initialize()

The `SAft.initialize()` method:

- Initializes Accessory File Transfer.
- Checks if the device supports Accessory File Transfer.
- Checks if the Accessory libraries are installed on the device.

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

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

5.2.2. Handling SsdkUnsupportedException

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

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

- `LIBRARY_NOT_INSTALLED`: The Samsung Accessory Service Framework is not installed on the device.

5.2.3. Checking the Availability of Accessory File Transfer

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

The flowing type is defined in the `SAft` class:

- DEVICE_ACCESSORY

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

5.3.Using Accessory File Transfer

The user's application does not need an existing service connection to use Accessory File Transfer. Both the sending and receiving application needs an interface implemented to get the peer, and an interface implemented to receive file transfer event updates (progress and completion). The sending application must know the peer to which it wants to send the file. Then, the receiving application must create an Accessory File Transfer object in order to receive the incoming file transfer request notifications. The sending application is usually considered a 'file provider' and the receiving application is considered the 'file consumer'.

The following chapter describes how to use Accessory File Transfer

5.3.1.Creating a Sender

Setting An Event Listener

The application must have a `SAAgent` subclass, an implementation of the `SAFileTransfer.EventListener` interface, and a `SAFileTransfer` instance to bind the application to Accessory File Transfer. The following example shows this implementation.

```
EventListener mCallback = new EventListener() {

    @Override
    public void onProgressChanged(int transId, int progress) {
        // Indicates the progress of transfer
    }

    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
      // Indicates that transfer has been finished
    }

    @Override
    public void onTransferRequested(int id, String fileName) {
      // No use in case of a file sender
    }
```

```
    @Override
    public void onCancelAllCompleted (int errorCode) {
     // No use in case of a file receiver
    }
};

SAFileTransfer mFileTransfer = new SAFileTransfer(this, mCallback);
```

Sending Files

To send a file, the application must know where the Accessory Peer Agent wants to send the file.

The Accessory Peer Agent can be obtained either by calling `SAAgent.findPeerAgents()` or by using the connected peer. The application must then call `SAFileTransfer.send()` on the `SAFileTransfer` object.

```
int tx = mFileTransfer.send(mPeerAgent, fileName);
```

In case that the file is external,

The file name provided must have a fully qualified path for the file. The data must be stored in a publicly-visible location, for example, on /mnt/sdcard. A unique transaction ID is returned to the application, which the application can retain for future reference.

In case that the file is internal,

Accessory File Transfer will convert the paths to URI using `FileProvider`. The application that will send a file from an internal path should implement these changes. Otherwise, an `IllegalArgumentException` will occur while trying to send files from an internal folder.

- Add the provider details in AndroidManafiest.xml for using content URI.
  A. Use package name as the authority if `FileProvider` is the only `ContentProvider` used in the application.
     - `android:authorities`="[application's package name]"
     - `android:resources`="[path of xml file having path information about file to be sent]"

```
<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.sender"
    android:exported="false"
    android:grantUriPermissions="true" >
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/accessoryservices" />
</provider>
```

  B. Use the unique string appending the unique string to package name as each authority if there are multiple `ContentProvider`s used in the application.
     - `android:authorities`="[application's package name.unique string]"
     - `android:resources`="[path of xml file having path information about file to be sent]"

```
<provider
    android:name="com.samsung.android.sdk.accessory.example.filetransfer.sender.YourProvider1"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.sender"
```

```
    ...
</provider>

<provider
    android:name=" com.samsung.android.sdk.accessory.example.filetransfer.sender.YourProvider2"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.sender.xxx"
    ...
</provider>

<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.sender.yyy"
    android:exported="false"
    android:grantUriPermissions="true" >
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/accessoryservices" />
</provider>
```

- `FileProvider` can only generate a context URI for files in directories that you specified beforehand. To specify a directory, specify the storage area and path in xml using child elements of the `<paths>` element. The xml file should be in the location mentioned in `android:resources` under the provider tag in `AndroidManifest.xml` similar to what is shown above. The following is a sample xml file for *paths*.

```
<paths xmlns:android="http://schemas.androidcom/apk/res/android">
    <files-path name="my_images" path="." />
    <caches-path name="my_cache" path="." />
</paths>
```

NOTE. The application can send files from the internal storage directories: /data/data/<application package>/files/ and /data/data/<application package>/cache/.

NOTICE. It should not be used to transfer sensitive or private information, since this method does not support any security features. If the application would like to transfer sensitive or private information, it needs to implement the encryption and decryption for security in its own application

Checking the Sending Progress

During the file transfer, progress updates are notified with the `SAFileTransfer.EventListener.onProgressChanged()` callback. Applications can update a progress bar based on the progress value received in the callback. When the file transfer is completed (successfully or not), the `onTransferCompleted()` callback is called with the appropriate error values. Applications can match the error codes with the error fields declared in the `SAFileTransfer` class.

Cancelling the Sending File

Applications can cancel the file transfer at any time by calling `SAFileTransfer.cancel()`. If a file transfer is cancelled, the `SAFileTransfer.EventListener.onTransferCompleted()` callback is called with a proper error code.

```
mFileTransfer.cancel(transId);
...
```

```
EventListener mCallback = new EventListener() {
    ...
    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
        ...
        if (errorCode == SAFileTransfer.ERROR_PEER_AGENT_REJECTED) {
            // Indicates that transfer has been cancelled or rejected.
        } else {
            ...
        }
    }
    ...
}
```

Sender application can cancel all file transfer request at any time by calling `SAFileTransfer.cancelAll()`. If a file tr ansfer is cancelled, the `onCancelAllCompleted()` is called with a proper error code.

```
mFileTransfer.cancelAll();
...
EventListener mCallback = new EventListener() {
    ...
    @Override
    public void onCancelAllCompleted(int errorCode) {
        ...
        if (errorCode == SAFileTransfer.ERROR_NONE) {
           // Indicates that transfer has been cancelled.
        } else {
            ...
        }
    }
    ...
}
```

NOTE. `SAFileTransfer.cancelAll()` and `SAFileTransfer.EventListener.onCancelAllCompleted()` are newly added from 2.3.0. Unless you override those methods, you can get a compilation error (unimplemented methods). It's recommended to add those methods to your implementation.

5.3.2.Creating a Receiver

Declaring Broadcast Receiver

Communicating with the sender needs the broadcast receiver below. This receiver is triggered when receiving a file transfer request. If it is not added in the `AndroidManifest.xml` file, intents will not be delivered to the developer's created application.

```
<application>
  ...
  <receiver
android:name="com.samsung.android.sdk.accessoryfiletransfer.SAFileTransferIncomingRequestReceiver">
      <intent-filter>
          <action android:name="com.samsung.accessory.ftconnection"/>
      </intent-filter>
  </receiver>
  ...
</application>
```

Setting An Event Listener

The application must have a `SAAgent` subclass and an implementation of the `SAFileTransfer.EventListener` interface and create a `SAFileTransfer` instance to bind the application to Accessory File Transfer. The following example shows this implementation.

```
EventListener mCallback = new EventListener() {

    @Override
    public void onProgressChanged(int transId, int progress)
        // Indicates the progress of transfer
    }

    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
        // Indicates that transfer has been finished
    }

    @Override
    public void onTransferRequested(int id, String fileName) {
        // No use in case of a file sender
    }

    @Override
    public void onCancelAllCompleted (int errorCode) {
        // No use in case of a file receiver
    }
};

SAFileTransfer mFileTransfer = new SAFileTransfer(this, mCallback);
```

Receiving Files

The `EventListener` instance and the `SAFileTransfer` object are needed to enable the receiving application to receive incoming file transfer requests. The Accessory File Transfer Service notifies the receiving application about the incoming request with the `SAFileTransfer.EventListener.onTransferRequested()` callback.

The application can inform the user through a notification or pop-up about the incoming file transfer and then ask for permission to accept or reject the incoming file transfer request.

The application must call `SAFileTransfer.receive()` on the `SAFileTransfer` object to receive the file. For exampl e, file path to be stored is `/storage/emulated/0/.`

```
mFileTransfer.receive(transId, "/storage/emulated/0/RecevicedFile.ext");
```

In case that file is external,

The destination file path where the received file is stored must be a publicly available location and also a fully qualified path. You can leave the parameter blank, in which case the file is stored in an external storage directory under a generated file name, for example, ReceivedFile<timestamp>.ext. An `IllegalArgumentException` occurs if an invalid file path or an invalid transaction ID is used.

In case that file is internal,

For receiving file in internal storage directory, the application must implement below changes or else an `IllegalArgumentException` will occur while trying to receive files from an internal directory.

- Add the provider details in AndroidManafiest.xml for using content URI.

  A. Use package name as the authority if `FileProvider` is the only `ContentProvider` used in the application.

  - `android:authorities`="[application's package name]"
  - `android:resources`="[path of xml file having path information about file to be received]"

```
<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.receiver"
    android:exported="false"
    android:grantUriPermissions="true" >
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/accessoryservices" />
</provider>
```

  B. Use the unique string appending the unique string to package name as each authority if there are multiple `ContentProvider`s used in the application.

  - `android:authorities`="[application's package name.unique string]"
  - `android:resources`="[path of xml file having path information about file to be received]"

```
<provider
    android:name="com.samsung.android.sdk.accessory.example.filetransfer.receiver.YourProvider1"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.receiver"
    ...
</provider>

<provider
    android:name=" com.samsung.android.sdk.accessory.example.filetransfer.receiver.YourProvider2"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.receiver.xxx"
    ...
</provider>

<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.samsung.android.sdk.accessory.example.filetransfer.receiver.yyy"
    android:exported="false"
    android:grantUriPermissions="true" >
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/accessoryservices" />
</provider>
```

- `FileProvider` can only generate a context URI for files in directories that you specified beforehand. To specify a directory, specify the storage area and path in xml using child elements of the `<paths>` element. The xml file should be in the location mentioned in `android:resources` under the provider tag in `AndroidManifest.xml` similar to what is shown above. The following is a sample xml file for *paths*.

```
<paths xmlns:android="http://schemas.androidcom/apk/res/android">
    <files-path name="my_images" path="." />
    <caches-path name="my_cache" path="." />
</paths>
```

Checking the Receiving Progress

The sender starts sending data only after `SAFileTransfer.receive()` is called.

During the file transfer, progress updates are notified with the `SAFileTransfer.EventListener.onProgressChanged()` callback. The application can update a progress bar based on the progress value received. When the file transfer is completed (successfully or not), the `SAFileTransfer.EventListener.onTransferCompleted()` callback is called with the requisite error code. The application can match the error code received with those defined in `SAFileTransfer` to find the exact reason for the error.

Rejecting the Receiving File

Applications can reject the file transfer receiving the incoming request with the `SAFileTransfer.EventListener.onTransferRequested()` callback by calling `SAFileTransfer.reject()`. If a file transfer is rejected, the `SAFileTransfer.EventListener.onTransferCompleted()` is called with the `SAFileTransfer.ERROR_PEER_AGENT_REJECTED` code.

```
mFileTransfer.reject(transId);
...
EventListener mCallback = new EventListener() {
...
    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
        ...
        if (errorCode == SAFileTransfer.ERROR_PEER_AGENT_REJECTED) {
            // Indicates that transfer has been cancelled or rejected.
        } else {
            ...
        }
    }
    ...
};
```

Cancelling the receiving File

Applications can cancel the file transfer any time by calling `SAFileTransfer.cancel()`.

If a file transfer is cancelled, the `SAFileTransfer.EventListener.onTransferCompleted()` is called with the `SAFileTransfer.ERROR_PEER_AGENT_REJECTED` code.

```
mFileTransfer.cancel(transId);
...
EventListener mCallback = new EventListener() {
...
    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
        ...
        if (errorCode == SAFileTransfer.ERROR_PEER_AGENT_REJECTED) {
            // Indicates that transfer has been cancelled or rejected.
        } else {
            ...
        }
    }
    ...
};
```

6. Samples

There are a few sample applications provide in order to help the developers understand the use of Accessory SDK. These sample applications can be also downloaded in the Samsung developer site.

- Hello Accessory (WEB / NATIVE)
  - Shows simple text interactions between Smart device and Remote device using Accessory SDK.
- Gallery (WEB / NATIVE)
  - Shows simple image exchange interactions between Smart device and Remote device using Accessory SDK.
- File Transfer (WEB / NATIVE)
  - Shows simple file transfer between Smart device and Remote device using Accessory SDK.
- Security Enabled (WEB / NATIVE)
  - Shows simple encrypted text interactions between Smart device and Remote device using secured APIs of Accessory SDK.
- Multiplicity (WEB / NATIVE)
  - Shows how to communicate a Provider application with multiple Consumer applications using Accessory SDK.
- Weather (HYBRID-WEB+NATIVE WIDGET)
  - Shows how to communicate Provider applications with Consumer Web application and Widget application using Accessory SDK.
- Hello Message (WEB /NATIVE)
  - Shows simple text interactions between Smart device and Remote device using Accessory Message of Accessory SDK.

6.1.Hello Accessory

The Hello Accessory sample application displays simple text interactions between Smart device and Remote device using Accessory SDK. This sample application is following a Gear companion type application and is provided in two types according to location of provider and consumer application.

Provider (Android) and Consumer (Gear)

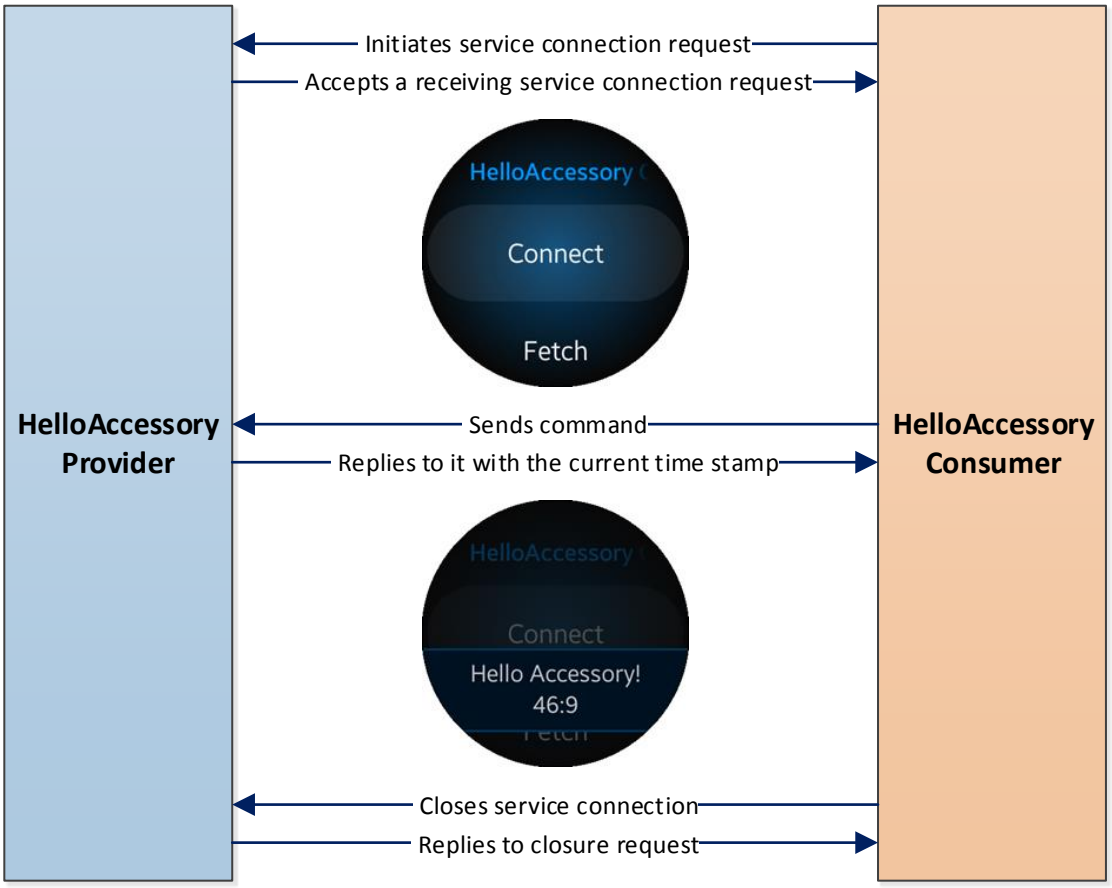


Figure 7: Hello Accessory - Provider (Android) and Consumer (Gear)

This type of sample application has two parts:

- Provider application
  - Works in Smart device, but has no UI.
  - Accepts a receiving a service connection request from remote Accessory Peer Agent in Remote device (Gear).
  - Replies to a receiving command from remote Accessory Peer Agent in Remote device (Gear) with current time stamp.
- Consumer application
  - Works in Remote device (Gear) and has UI.
  - Initiates service connection request and sends command to peer Accessory Agent in Smart device.
  - Shows a received response to user.

Provider (Gear) and Consumer (Android)

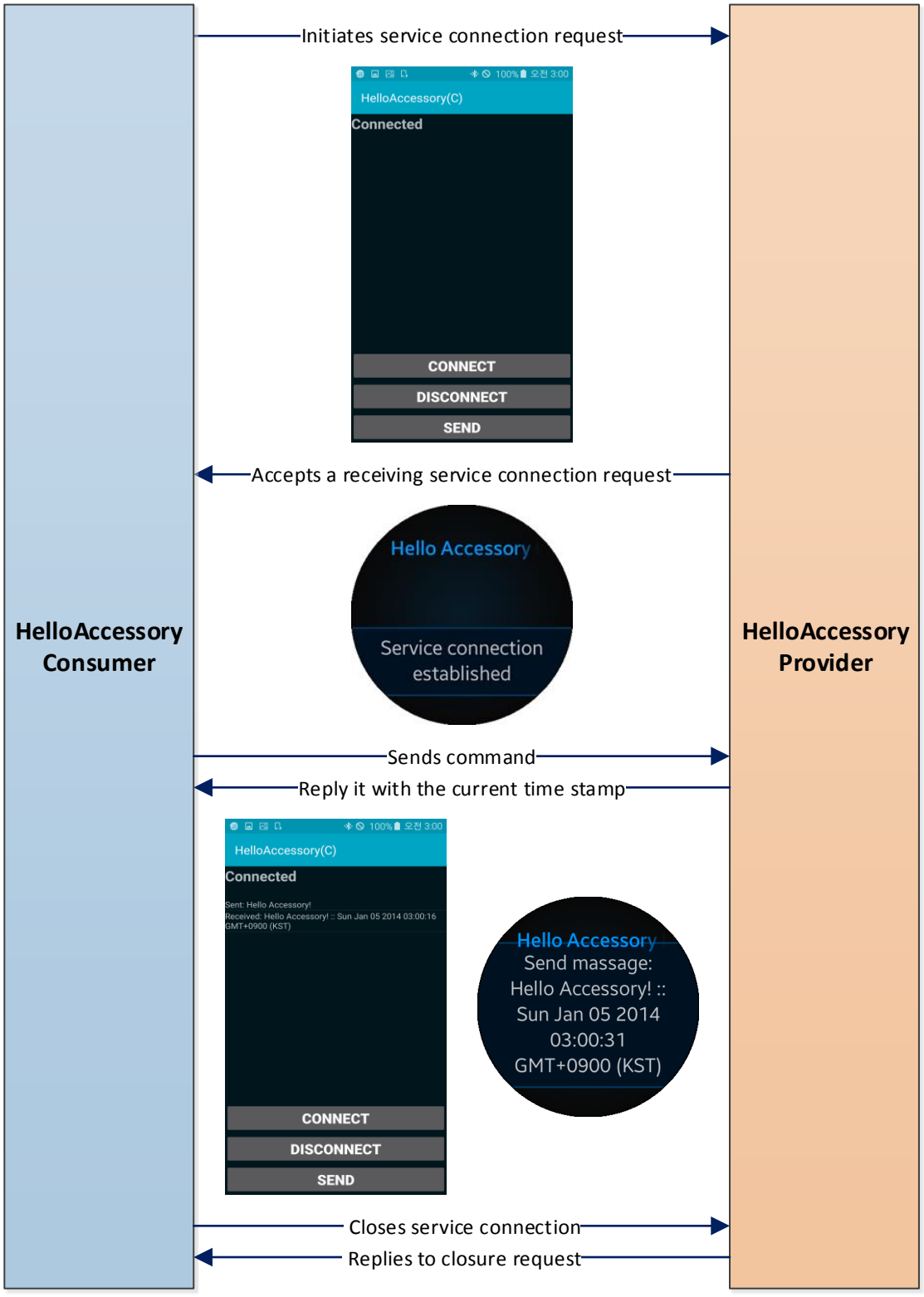


Figure 8: Hello Accessory - Provider (Gear) and Consumer (Android)

This sample application has two parts:

- Provider application

  - Works in Remote device (Gear) and has UI.
  - Accepts a receiving a service connection request from remote Accessory Peer Agent in Smart device.
  - Replies to a receiving command from remote Accessory Peer Agent in Smart device with current time stamp.

- Consumer application
  - Works in Smart device and has UI.
  - Initiates service connection request and sends command to remote Accessory Peer Agent in Remote device (Gear).
  - Shows a received response to user.

6.2.Gallery

Gallery sample application displays simple image exchange interactions between Smart device and Remote device using Accessory SDK. This sample application is following a Gear companion type application and is provided in two types according to location of provider and consumer application.

Provider (Android) and Consumer (Gear)

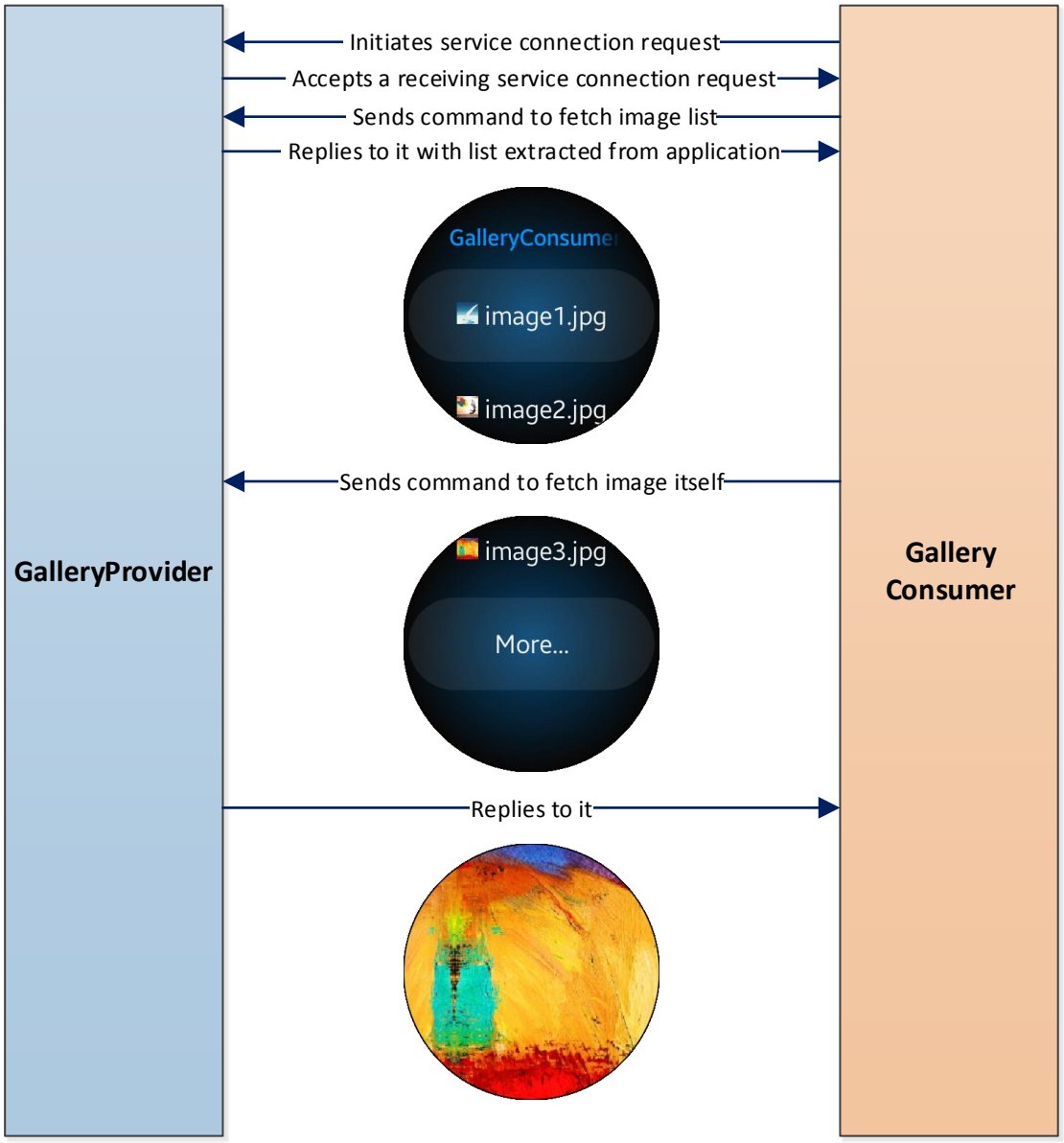


Figure 9: Gallery - Provider (Android) and Consumer (Gear)

This sample application has two parts:

- Provider application
  - Works in Smart device and has no UI.
  - Accepts a receiving a service connection request from remote Accessory Peer Agent in Remote device (Gear).

- Replies to a receiving command to fetch image list from remote Accessory Peer Agent in Remote device (Gear) with list extracted from application.
- Replies to a receiving command to fetch images from remote Accessory Peer Agent in Remote device (Gear) with actual images after encoding it as BASE64.

- Consumer application
  - Works in Remote device (Gear) and has UI.
  - Initiates service connection request and send commands to remote Accessory Peer Agent in Smart device.
  - When receiving a response, decodes BASE64 encoded images and shows it to user.

Provider (Gear) and Consumer (Android)

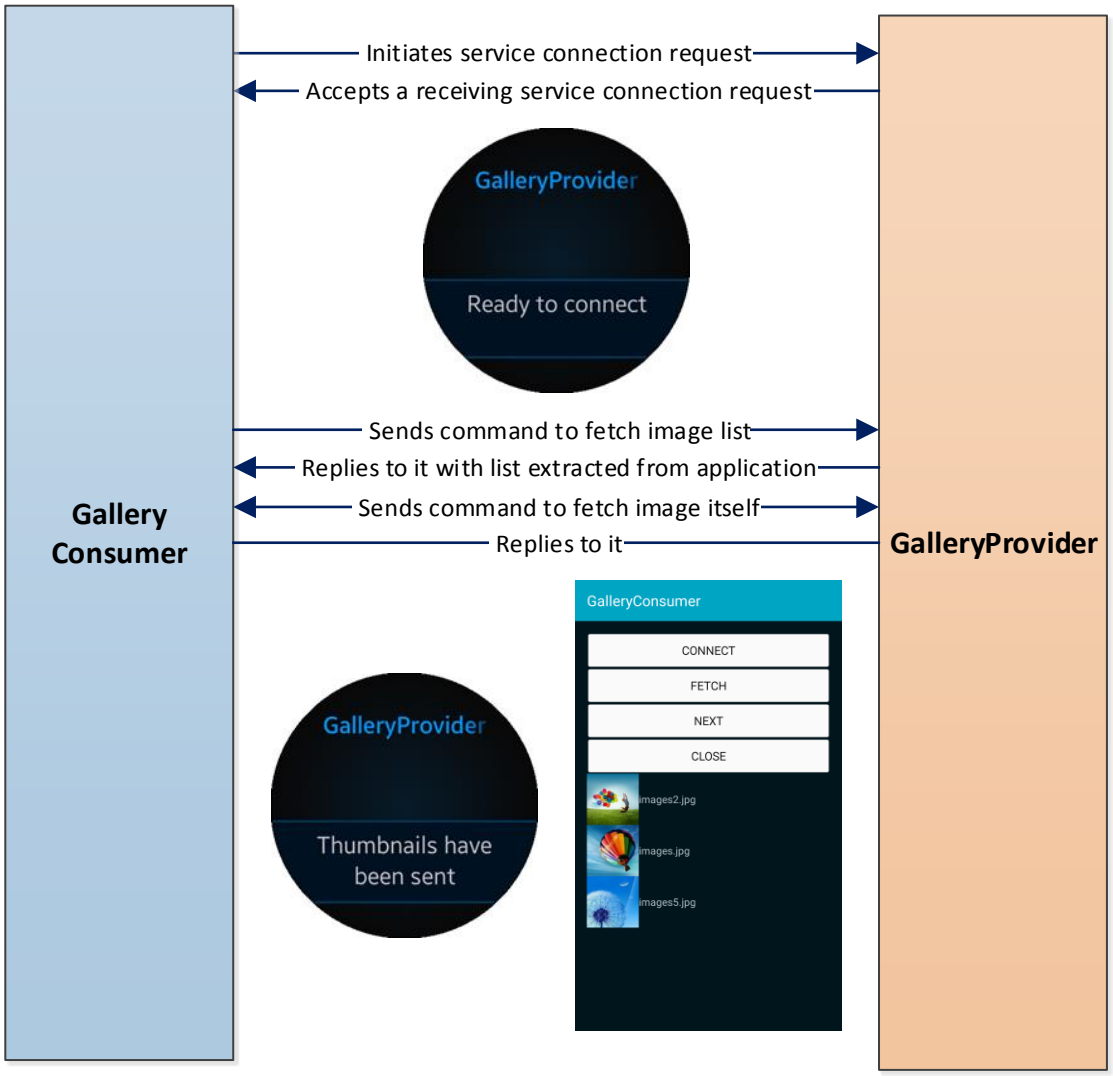


Figure 10: Gallery - Provider (Gear) and Consumer (Android)

This sample application has two parts:

- Provider application

  - Works in Remote device (Gear) and has UI.
  - Accepts a receiving a service connection request from remote Accessory Peer Agent in Smart device.
  - Replies to a receiving command to fetch image list from remote Accessory Peer Agent in Smart device with list extracted from Gear application.
  - Replies to a receiving command to fetch images from remote Accessory Peer Agent in Smart device with actual images after encoding it as BASE64.

- Consumer application
  - Works in Smart device and has UI.
  - Initiates service connection request and send commands to remote Accessory Peer Agent in Remote device (Gear).
  - When receiving a response, decodes BASE64 encoded images and shows it to user.

6.3.File Transfer

File Transfer sample application displays file exchange interactions between the Smart device and the Remote device using Accessory File Transfer SDK. This sample application is following a Gear companion type application and is provided in two types according to location of sender and receiver application.

Sender (Android) and Receiver (Gear)

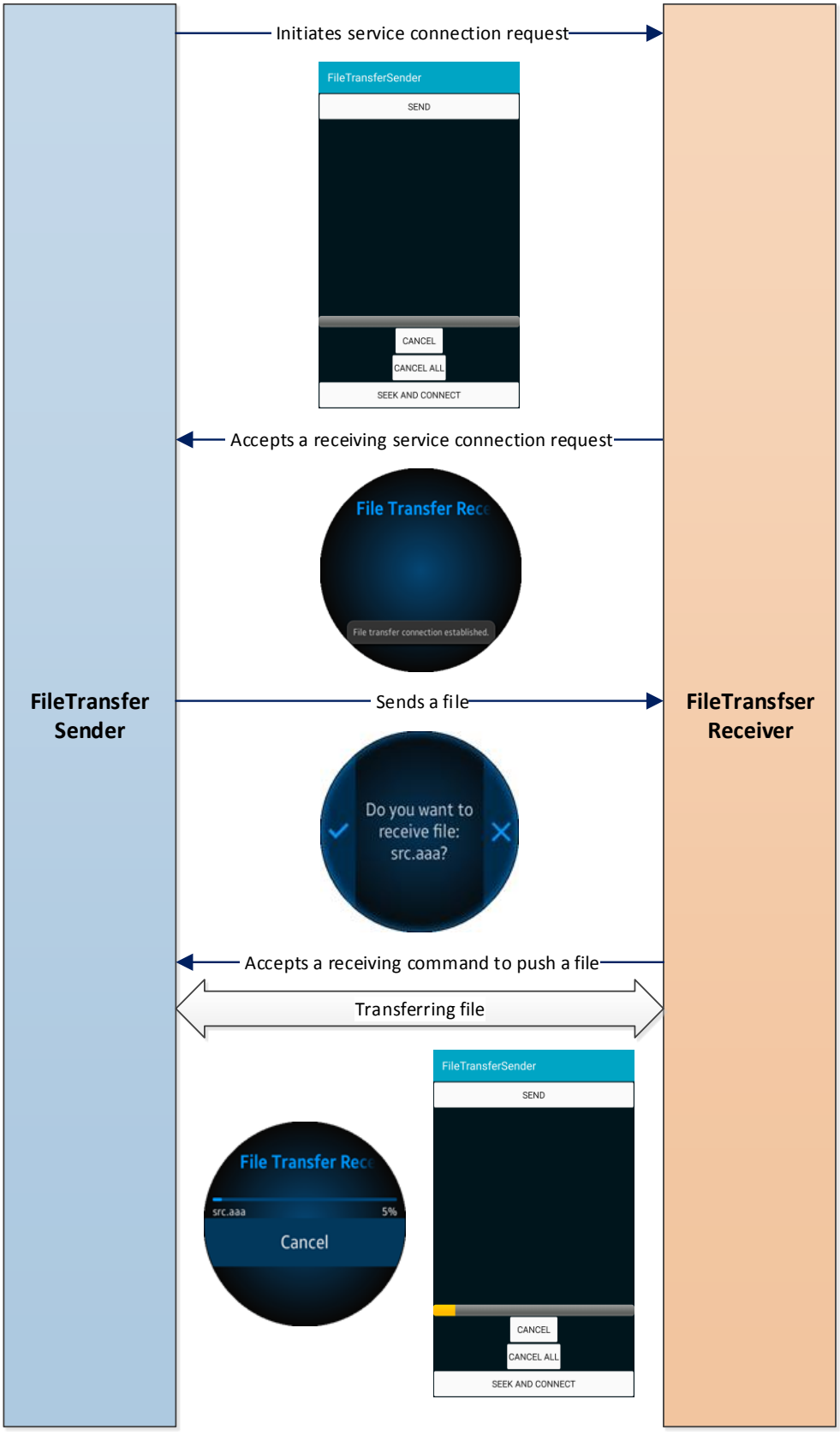


Figure 11: File Transfer - Sender (Android) and Receiver (Gear)

This sample application has two parts:

- Sender application
  - Works in Smart device and has UI.
  - Sends files to remote Accessory Peer Agent in Remote device (Gear).
- Receiver application
  - Works in Remote device (Gear) and has UI.
  - Accepts or rejects a receiving command to push file from remote Accessory Peer Agent in Smart device.
  - Receiving file from remote Accessory Peer Agent in Smart device.

NOTE. Provider application does not have any file to be sent. It's necessary to push file named src.aaa into Smart device before clicking Send button

Sender (Gear) and Receiver (Android)

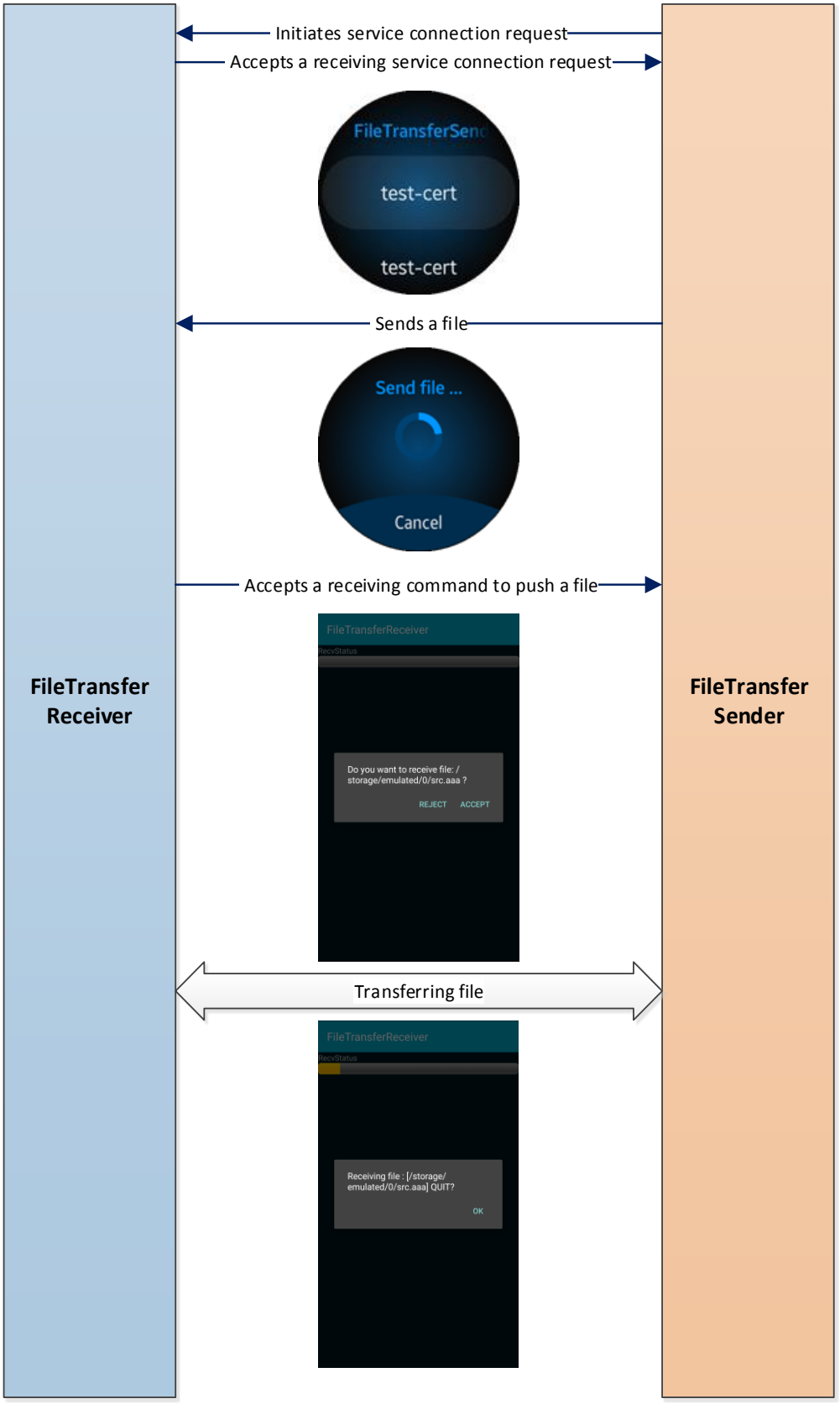


Figure 12: File Transfer - Sender (Gear) and Receiver (Android)

This sample application has two parts:

- Sender application
  - Works in Remote device (Gear) and has UI.
  - Sends files to remote Accessory Peer Agent in Smart device.
- Receiver application
  - Works in Smart device and has UI.
  - Accepts or rejects a receiving command to push file from remote Accessory Peer Agent in Remote device (Gear).
  - Receiving file from remote Accessory Peer Agent in Remote device (Gear).

6.4.Security Enabled

Security enabled sample application displays simple text interactions between Smart device and Remote device using secured APIs of Accessory SDK. This sample application is following a Gear companion type application and is provided in two types according to location of provider and consumer application.

Provider (Android) and Consumer (Gear)

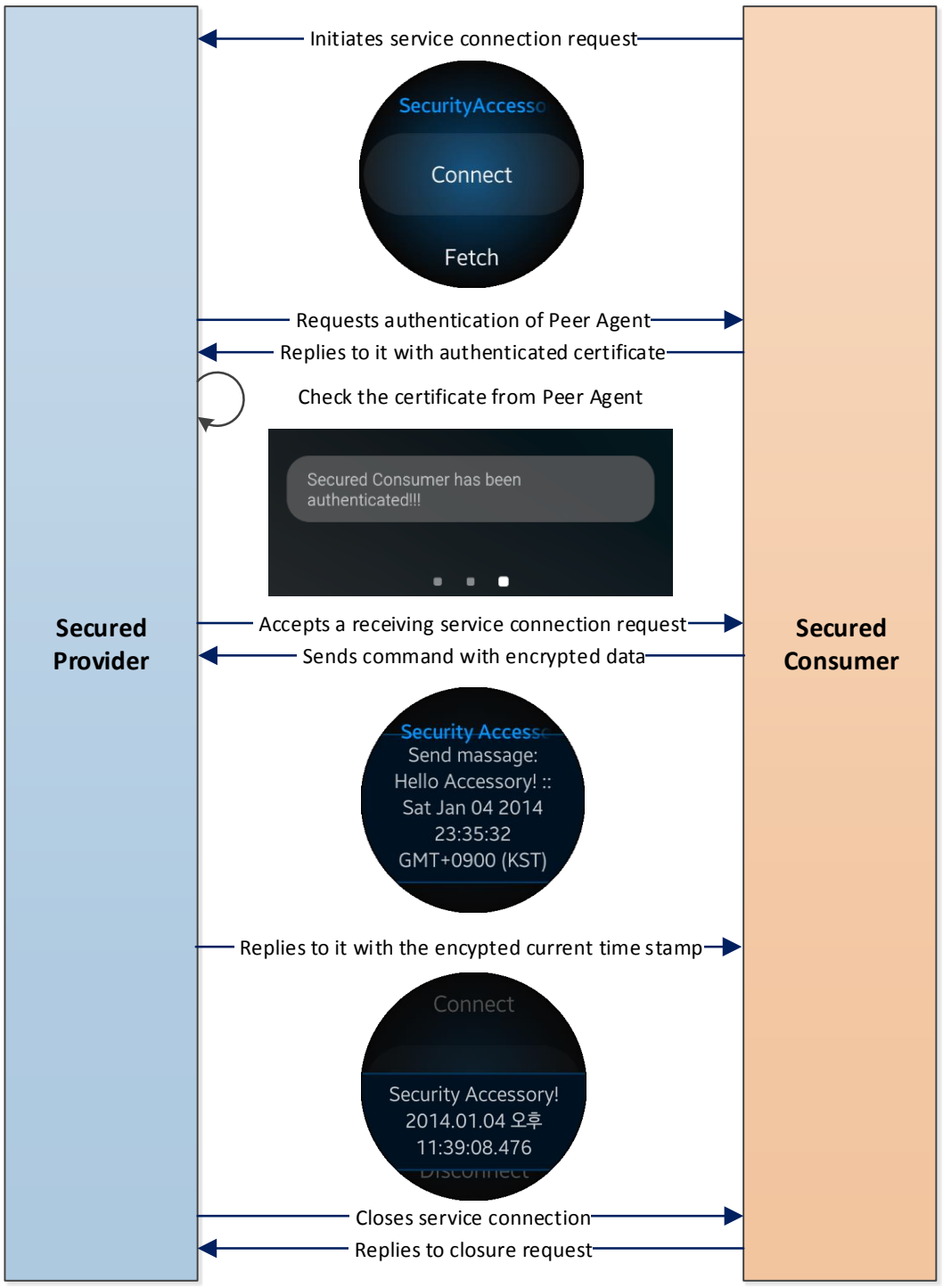


Figure 13: Security Enabled - Provider (Android) and Consumer (Gear)

This type of sample application has two parts:

- Provider application
  - Works in Smart device, but has no UI.
  - Authenticates a remote Accessory Peer Agent in Remote device (Gear).
  - Accepts a receiving a service connection request from remote Accessory Peer Agent in Remote device (Gear).
  - Replies to a receiving command from remote Accessory Peer Agent in Remote device (Gear) with encrypted current time stamp.
- Consumer application
  - Works in Remote device (Gear) and has UI.
  - Initiates service connection request and sends command to peer Accessory Agent in Smart device.
  - Shows a received response to user.

Provider (Gear) and Consumer (Android)

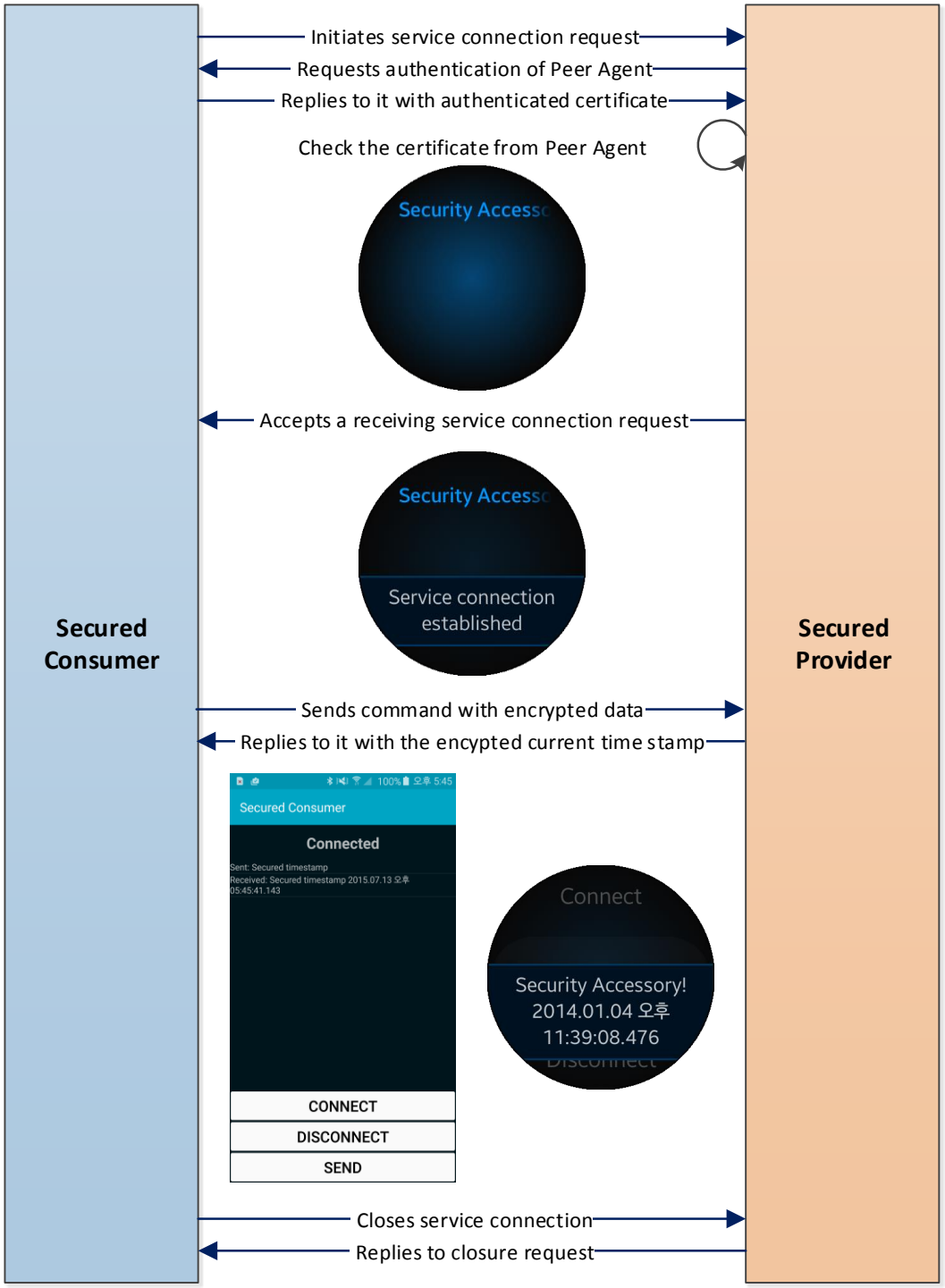


Figure 14: Security Enabled - Provider (Gear) and Consumer (Android)

This sample application has two parts:

- Provider application
  - Works in Remote device (Gear) and has UI.

  - Authenticates a remote Accessory Peer Agent in Smart device.
  - Accepts a receiving a service connection request from remote Accessory Peer Agent in Smart device.
  - Replies to a receiving command from remote Accessory Peer Agent in Smart device with encrypted current time stamp.

- Consumer application
  - Works in Smart device and has UI.
  - Initiates service connection request and sends command to remote Accessory Peer Agent in Remote device (Gear).
  - Shows a received response to user.

NOTE. Due to platform difference, it's necessary that Gear App creates author certificate using Android keystore. Please refer to ***Appendix D. Creating Gear Author Certificate Using Android Keystore*****.** It's also necessary to sign Android App with Android Keystore and Gear App with author certificate created from Android Keystore.

6.5.Multiplicity

Multiplicity sample application shows how to communicate a Provider application with multiple Consumer applications using Accessory SDK as one of possible multiplicity combinations. This sample application is following a Gear companion type application and is provided in two types according to location of provider and consumer application.

Provider (Android) and Consumer (Gear)

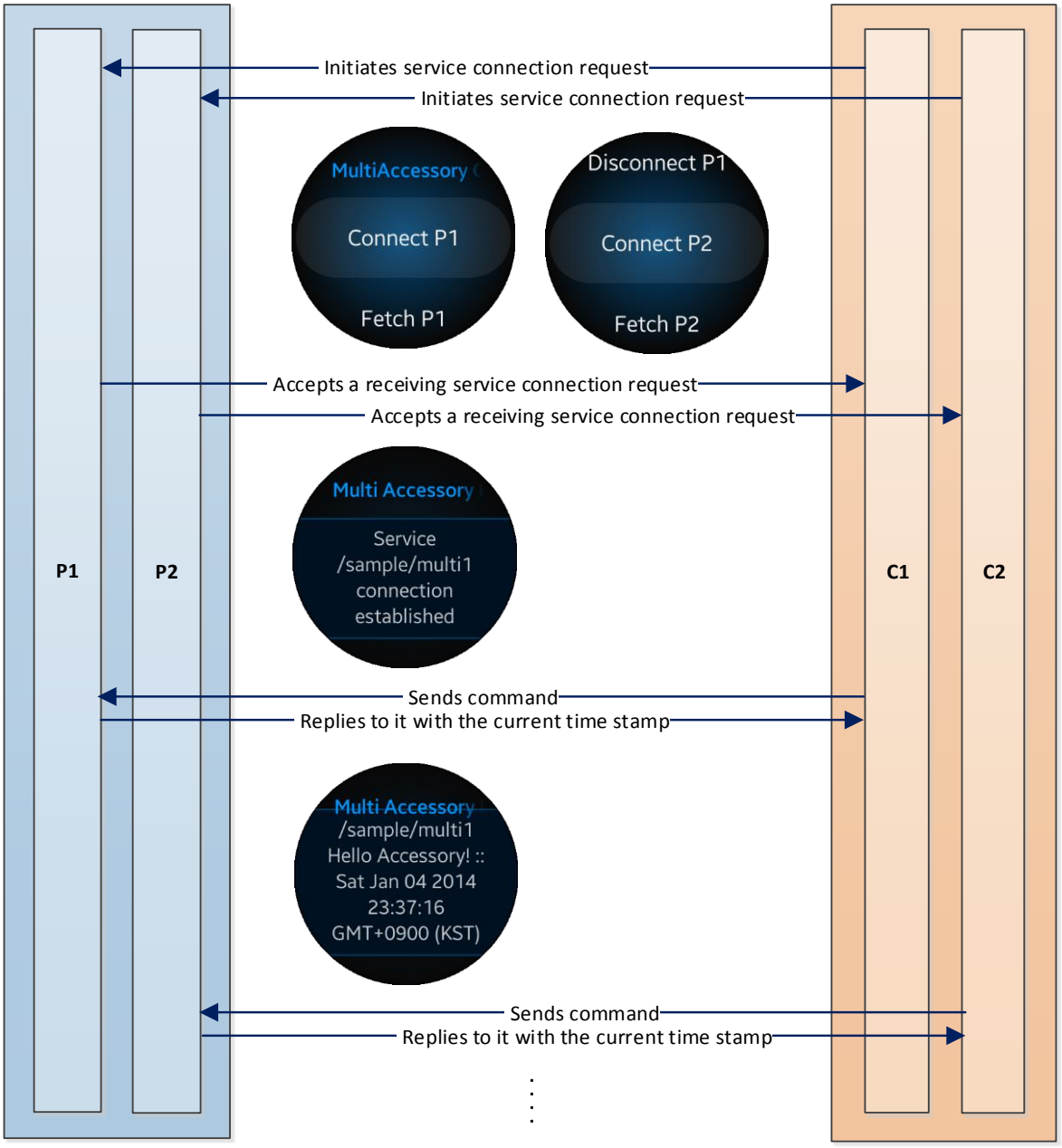


Figure 15: Multiplicity - Provider (Android) and Consumer (Gear)

This type of sample application has two parts:

- Provider application

  - Works in Smart device, but has no UI.
  - Includes two Providers having different Accessory Service Profile.
  - Accepts a receiving a service connection request from remote Accessory Peer Agents in Remote device (Gear) independently.
  - Replies to a receiving command from remote Accessory Peer Agents in Remote device (Gear) independently.

- Consumer application
  - Works in Remote device (Gear) and has UI.
  - Initiates service connection request and sends command to peer Accessory Agents in Smart device independently.
  - Shows a received response to user independently.

Provider (Gear) and Consumer (Android)

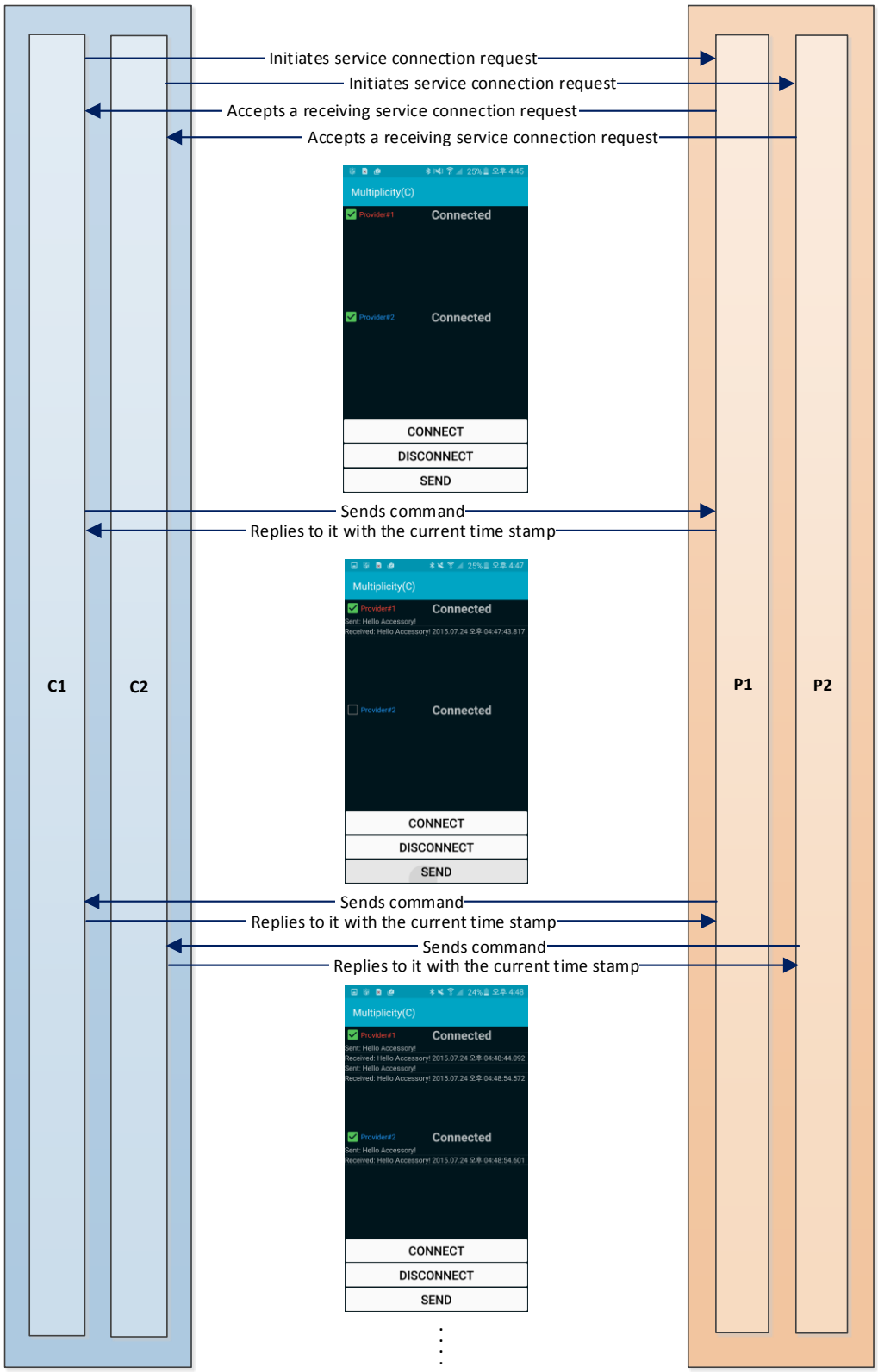


Figure 16: Multiplicity - Provider (Gear) and Consumer (Android)

This sample application has two parts:

- Provider application
  - Works in Remote device (Gear), but has UI.
  - Includes two Providers having different Accessory Service Profile.
  - Accepts a receiving a service connection request from remote Accessory Peer Agents in Smart device independently.
  - Replies to a receiving command from remote Accessory Peer Agents in Smart device independently.
- Consumer application
  - Works in Smart device and has UI.
  - Initiates service connection request and sends command to peer Accessory Agents in Remote device (Gear) independently.
  - Shows a received response to user independently.

6.6.Weather

The Weather sample application is a companion type application that shows how multiple Provider applications can communicate with multiple Consumer applications (WebApp and Widget) using the Accessory SDK

Provider (Android) and Consumer (Gear)

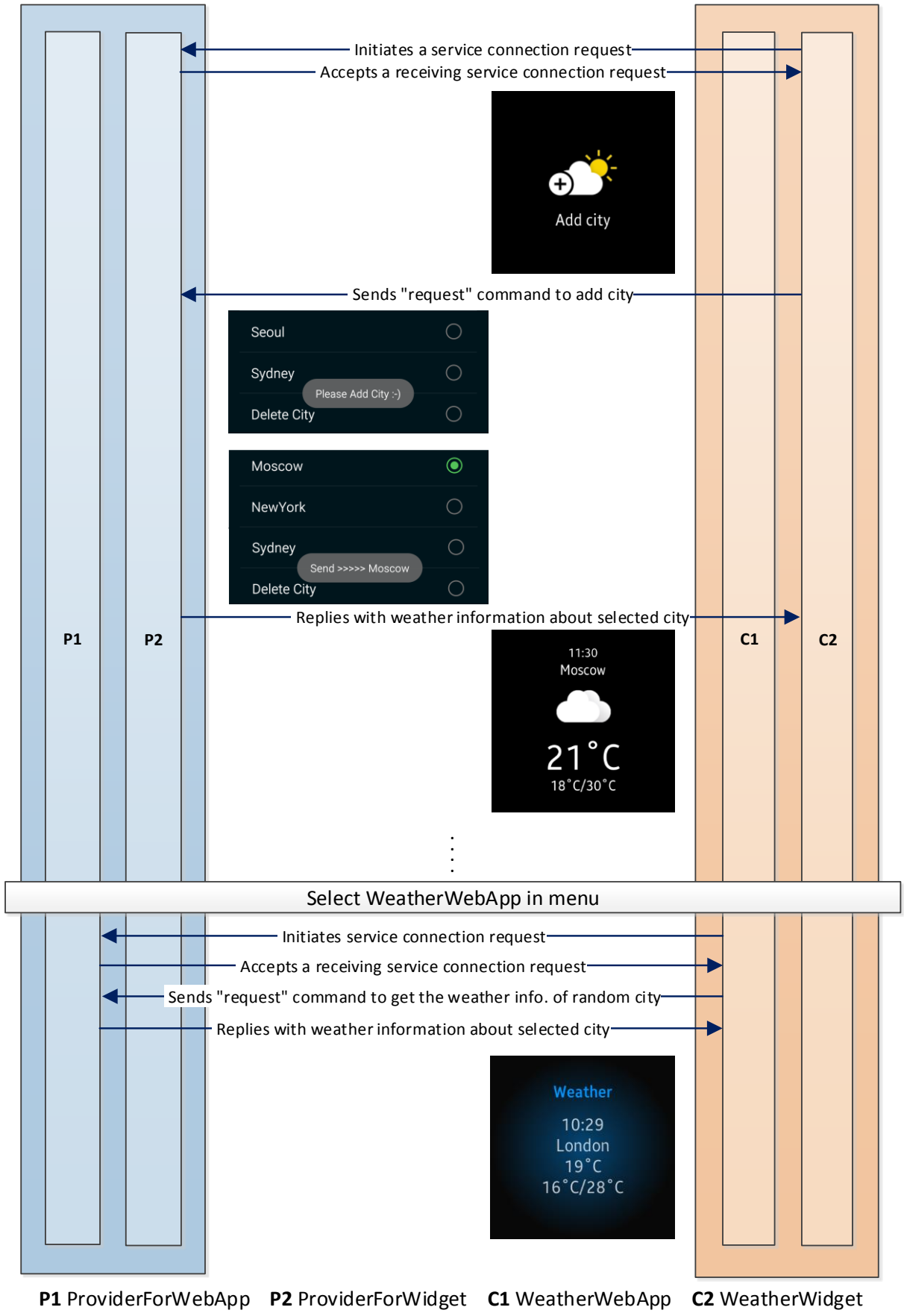


Figure 17: Weather - Provider (Gear) and Consumer (Android)

This sample application has three parts:

- WeatherProvider
  - Works in Host device and has UI
  - Includes two Providers having different Accessory Service Profiles
  - Accepts a receiving service connection request from Gear device independently
  - Replies to a receiving command from Gear device independently
- WeatherWebApp
  - Works in Gear device and has UI
  - Initiates a service connection request and sends commands to Host device independently
  - Shows a received response to user independently
  - Shows the weather information of random city among stored cities
- WeatherWidget
  - Works in Gear device and has UI
  - Initiates a service connection request and sends commands to Host device independently
  - Shows a received response to the user independently
  - Shows the weather information of selected cities
  - Shows to user the screen to add the city

> ***NOTE.*** Please refer to the link below to package a hybrid application.
> *https://developer.tizen.org/community/tip-tech/packaging-hybrid-application*

6.7.Hello Message

The Hello Message sample application displays simple text interactions between Smart device and Remote device using Accessory SDK. This sample application is following a Gear companion type application and is provided in two types according to location of provider and consumer application.

Provider (Android) and Consumer (Gear)

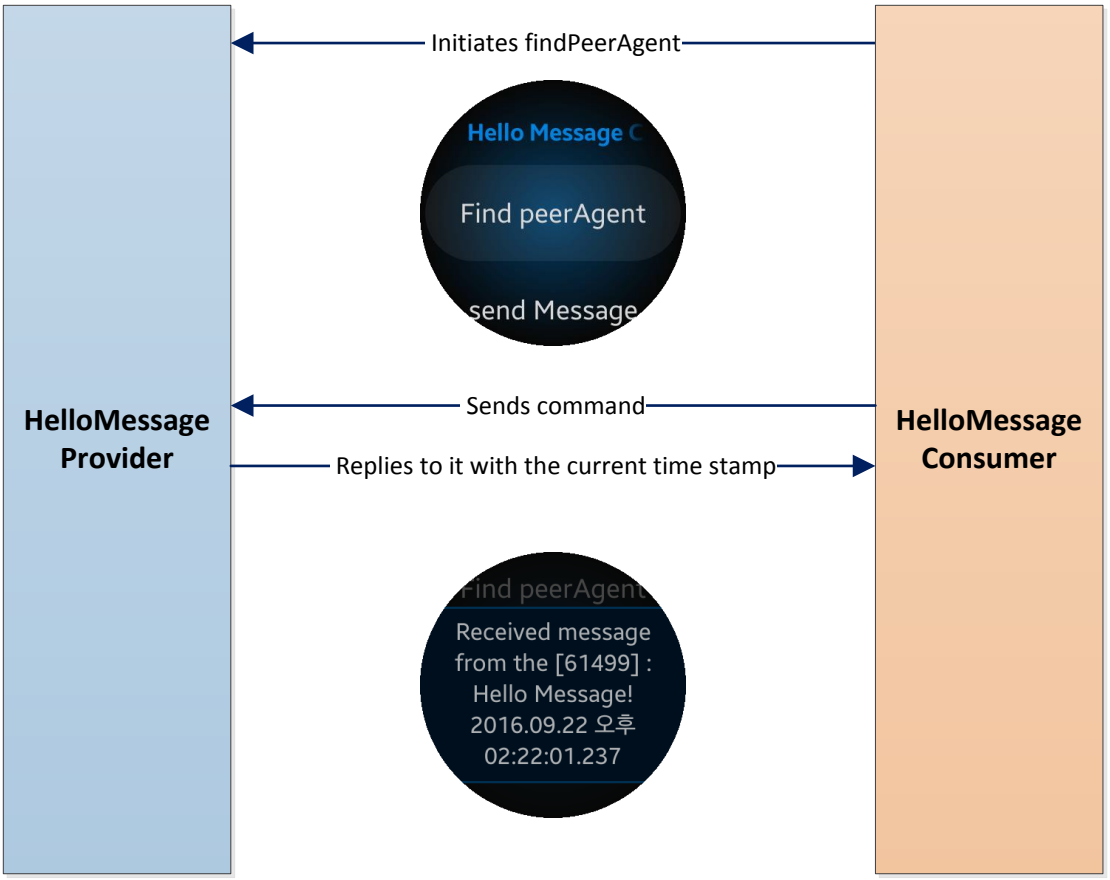


Figure 18: Hello Message - Provider (Android) and Consumer (Gear)

This type of sample application has two parts:

- Provider application
  - Works in Smart device, but has no UI.
  - Replies to a receiving command from remote Accessory Peer Agent in Remote device (Gear) with current time stamp.
- Consumer application
  - Works in Remote device (Gear) and has UI.
  - Sends command to peer Accessory Agent in Smart device.
  - Shows a received response to user.

Provider (Gear) and Consumer (Android)

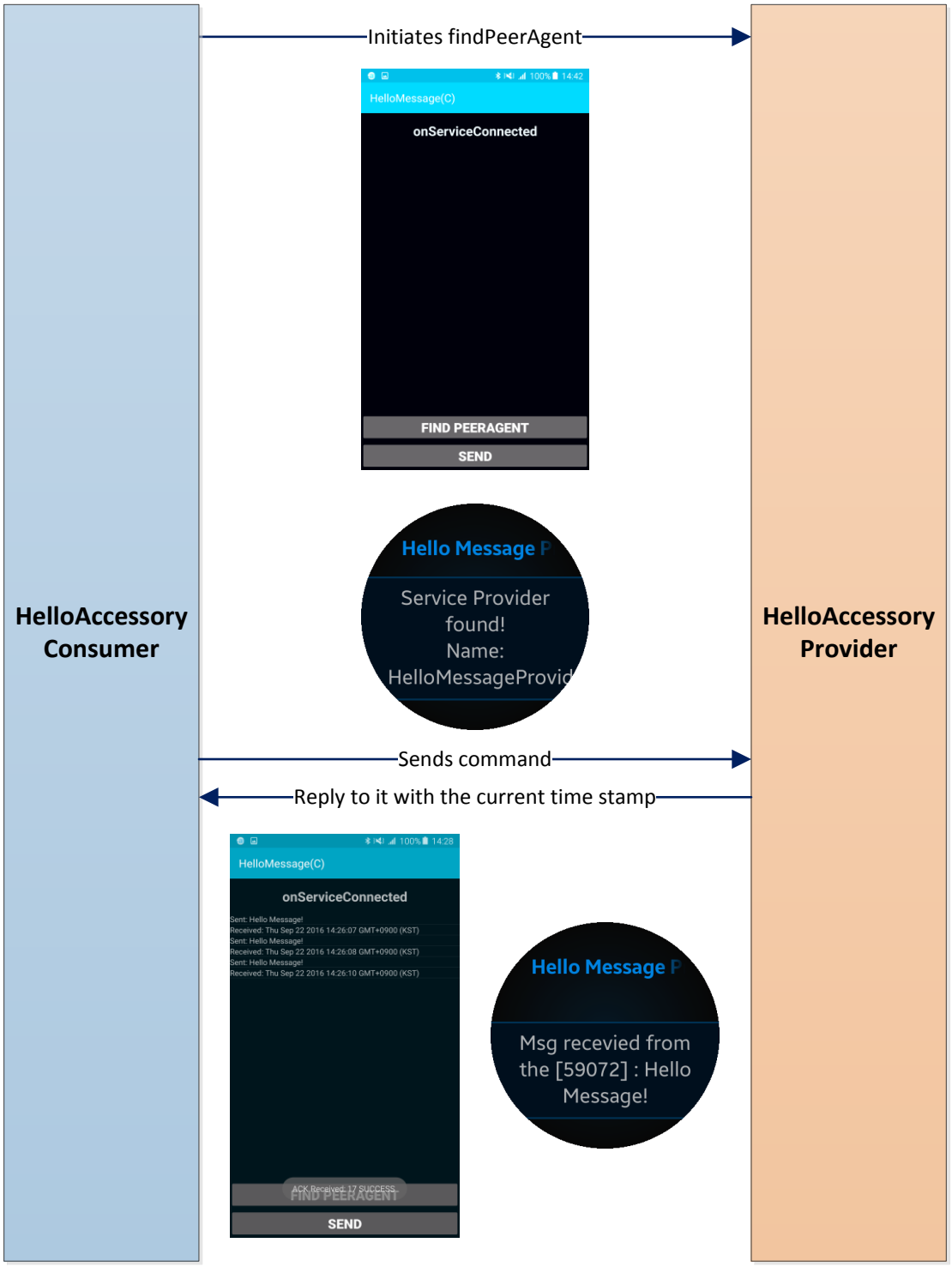


Figure 19: Hello Message - Provider (Gear) and Consumer (Android)

This sample application has two parts:

- Provider application
  - Works in Remote device (Gear) and has UI.

  - Replies to a receiving command from remote Accessory Peer Agent in Smart device with current time stamp.

- Consumer application
  - Works in Smart device and has UI.
  - Sends command to remote Accessory Peer Agent in Remote device (Gear).
  - Shows a received response to user.

7. Tools

The Accessory SDK provides tools for its application development.

7.1. Emulator

This is used to develop SAMSUNG GEAR applications using GEAR IDE without actual devices

Setting up the Test Environment

For preparing the Accessory test environment with GEAR IDE, **download and install Samsung GEAR Manager** into your Smart device **at first**. After that, find *Application_for_Emulator.zip*, which includes the necessary file in `tools` folder in SDK. After extracting zipped file, you can find the following file: *SAccessoryService_Emul.apk* . And then install it on your Smart device.

```
adb install -r SAccessoryService_Emul.apk
```

NOTE. Even if Samsung Accessory Service was already installed, install this file. Below figure shows the correct installation status for emulator.

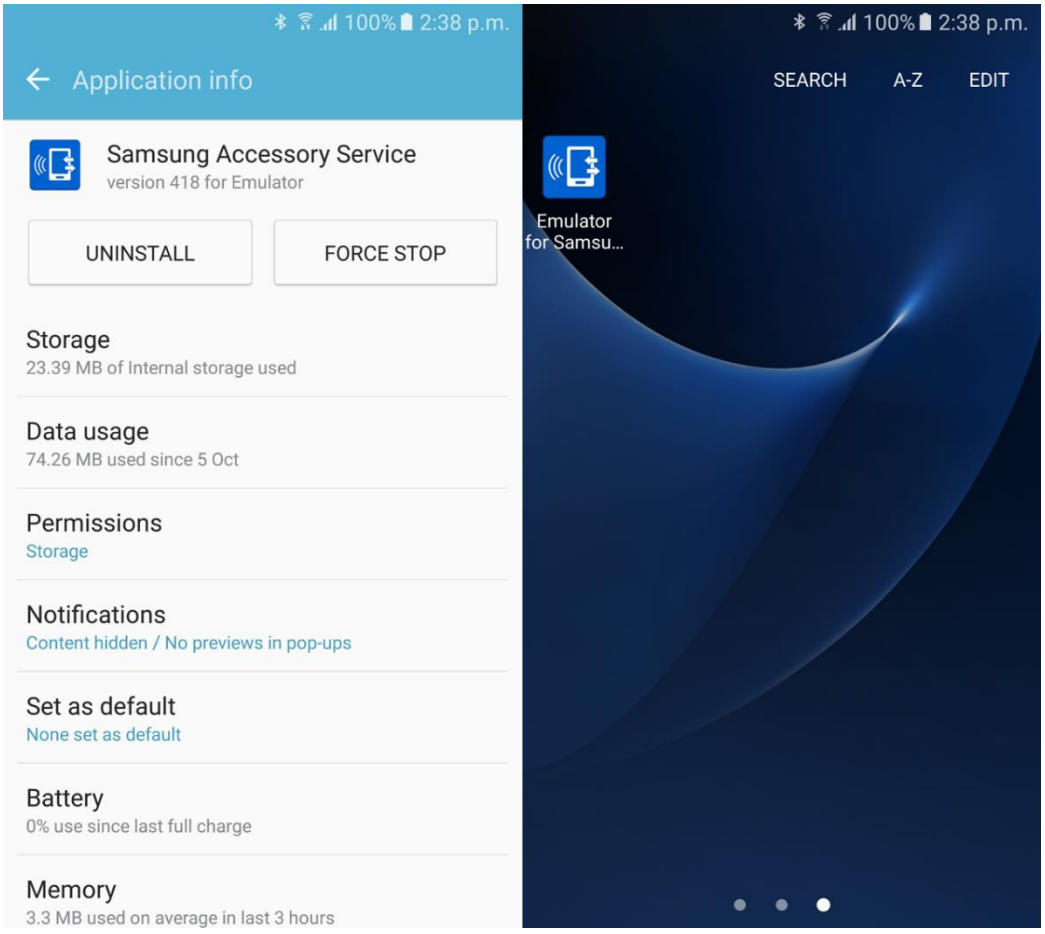


Figure 20: Install Application for Emulator

After installing that file is done, reboot the Smart device and then run *Emulator for Samsung Accessory* application.

Setting up the connection to GEAR IDE:

1. Connect the Smart device and PC via USB.
2. Open the terminal (or the command window in Windows®).
3. Execute the command below:

```
adb -d forward tcp:8230 tcp:8230
```

4. Run *Emulator for Samsung Accessory* application in the Smart device.
5. Turn on Wi-Fi or Mobile Network of Smart device
6. In the Tizen Studio, run the Emulator Manager and create a new virtual machine.
7. Run the virtual machine of GEAR IDE.

   The application's '*Disconnected*' text is changed to '*Connected*', showing that the connection through the SAP server has succeeded. If the 'Disconnected' text does not change, restart the emulator with the hardware key or the sdb shell command below (DO NOT use the sdb shell command `reboot`)

```
shutdown -r now
```

   or enter below sdb shell command (the command requires root authority).

```
killall sapd
```

NOTE. Prior to trying the connection, make sure the following are ready: Installing the Android Debug Bridge (ADB), setting up the path of ADB into System Variables for utilizing it in command window, installing the GEAR IDE and running the GEAR IDE.

Cleaning Up the Test Environment

For returning the Accessory test environment to one for the actual GEAR devices, uninstall the existing application for Emulator: *Samsung Accessory Service* in your Smart device. Then, launch the Samsung Gear Manager.

NOTE. **Failure to** uninstall that application for Emulator could result for it not to work properly on communicating with the actual GEAR devices.

Appendix A. Programming Tips

A.1. Using File Transfer

It's helpful to remember the following tips when implementing file transfers:

- Accessory File Transfer Service maintains its own queue for all file sending operations. Individual applications need not and must not maintain their own queues to control file transfer. All `SAFileTransfer.send()` calls are queued and serviced sequentially, even when they came from multiple user applications.
- There is a timeout of 10 seconds when the sending application sends the file transfer request to the receiving application. If the receiver does not accept or reject the file transfer within that time, it is cancelled and an error code is thrown on the sender side through the `SAFileTransfer.EventListener.onTransferCompleted()` callback. This is also the case when the application forgets to register an incoming file transfer request broadcast receiver or to call `SAFileTransfer.receive()`.
- Accessory File Transfer Service checks whether there is enough space on the receiving device to receive the incoming file. If not, it rejects the file transfer automatically without informing the receiving application.
- Accessory File Transfer Service checks whether there is already a file with the same name present in the location provided. If there is, it appends a timestamp to the given file name. If no file path is provided, the file is stored in the external storage directory under a generated file name.
- In the current Accessory File Transfer Service implementation, files can be transferred with or without a service connection between user applications. This is different from the previous implementation, where a service connection was necessary.
- It is mandatory to implement the `EventListener` interface for file transfer updates.
- If you have multiple `SAAgent` implementations in your application, all using Accessory File Transfer, each one must create its own `SAFileTransfer` object. On the receiver side, all the agents must be registered. Accessory File Transfer SDK resolves the intended `SAAgent` implementation for every incoming file transfer request and notifies it with its specific `SAFileTransfer.EventListener.onTransferRequested()` callback.
- There is one binding to Accessory File Transfer per application, regardless of the number of `SAAgent` implementations in the application.
- `SAFileTransferIncomingRequestReceiver` must be declared in the receiving application's manifest.
- On the receiver side, one `SAAgent` implementation must maintain only a single `SAFileTransfer` object in its lifetime. If multiple instances are created, the app will receive the `SAFileTransfer.EventListener.onTransferRequested()` callback for every registered instance during an incoming file transfer request. A suggested failsafe is to call `SAFileTransfer.close()` and then set the `SAFileTransfer` object to null in the `onDestroy()` of the `SAAgent` implementation.
- `SAFileTransfer.close()` API should be called by the application only when the application has no use of File Transfer Service. Once the API is called, the application won't be able to receive or send files until it creates a new SAFileTransfer instance. Application should make sure all the ongoing and pending requests are completed before calling `SAFileTransfer.close()` else a `RuntimeException` exception will be thrown.
- Accessory File Transfer is using the package name as the default authority for `FileProvider`. If application defines provider only for using File Transfer Content URI, it can continue to use the same. For applications with multiple providers, they can declare their authority for File Provider with different strings like adding some extra string in the end of package name.

A.2. Validating Accessory Service Profile XML

Validating the defined Accessory Service Profile lowers the chances of registration failure by wrong Accessory Service Profile description. The Samsung Accessory Service Framework provides two kinds of validating methods: *Document Type Definition (DTD)* Schema and XML Schema.

A.1.1. DTD

The following code snippet shows the Accessory Service Profile DTD file, accessoryservices.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) #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>
    ]>
```

A.1.2. XML Schema

The following code snippet shows the Accessory Service Profile XML Schema file, accessoryservices.xsd.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:element name="resources">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="application" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="application">
        <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:element ref="serviceProfile" />
            </xs:sequence>
```

```
            <xs:attribute name="name" type="xs:normalizedString" use="required" />
        </xs:complexType>
    </xs:element>
    <xs:element name="serviceProfile">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="supportedTransports" />
                <xs:element minOccurs="1" maxOccurs="unbounded" ref="serviceChannel" />
            </xs:sequence>
            <xs:attribute name="serviceImpl" type="xs:normalizedString" use="required" />
            <xs:attribute name="role" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="provider" />
                        <xs:enumeration value="consumer" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="name" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:normalizedString">
                        <xs:minLength value="1" />
                        <xs:maxLength value="30" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="id" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:normalizedString">
                        <xs:minLength value="1" />
                        <xs:maxLength value="30" />
                        <xs:pattern value="\/[a-z0-9_]+(\/([a-z0-9_])+)*" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="version" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:normalizedString">
                        <xs:minLength value="3" />
                        <xs:maxLength value="5" />
                        <xs:pattern value="[1-9]?[0-9][.][0-9]?[0-9]" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="serviceLimit" use="optional" default="any">
                <xs:simpleType>
                    <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="ANY" />
                        <xs:enumeration value="ONE_ACCESSORY" />
                        <xs:enumeration value="ONE_PEERAGENT" />
                        <xs:enumeration value="any" />
                        <xs:enumeration value="one_accessory" />
                        <xs:enumeration value="one_peeragent" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="serviceTimeout" use="optional" default="0">
                <xs:simpleType>
                    <xs:restriction base="xs:integer">
                        <xs:minInclusive value="0" />
                        <xs:maxInclusive value="300" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
```

```
    </xs:element>
    <xs:element name="supportedTransports">
        <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:element ref="transport" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="transport">
        <xs:complexType>
            <xs:attribute name="type" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="TRANSPORT_WIFI" />
                        <xs:enumeration value="TRANSPORT_BT" />
                        <xs:enumeration value="TRANSPORT_BLE" />
                        <xs:enumeration value="TRANSPORT_USB" />
                        <xs:enumeration value="transport_wifi" />
                        <xs:enumeration value="transport_bt" />
                        <xs:enumeration value="transport_ble" />
                        <xs:enumeration value="transport_usb" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
    <xs:element name="serviceChannel">
        <xs:complexType>
            <xs:attribute name="id" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:integer">
                        <xs:minInclusive value="1" />
                        <xs:maxInclusive value="9999" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="dataRate" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="LOW" />
                        <xs:enumeration value="HIGH" />
                        <xs:enumeration value="low" />
                        <xs:enumeration value="high" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="priority" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="LOW" />
                        <xs:enumeration value="MEDIUM" />
                        <xs:enumeration value="HIGH" />
                        <xs:enumeration value="low" />
                        <xs:enumeration value="medium" />
                        <xs:enumeration value="high" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="reliability" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:NMTOKEN">
                        <xs:enumeration value="ENABLE" />
                        <xs:enumeration value="DISABLE" />
                        <xs:enumeration value="enable" />
                        <xs:enumeration value="disable" />
```

```
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

A.1.3. Procedure

Before validating an Accessory Service Profile description, add a reference to the DTD rules to the topmost part of your Accessory Service Profile XML file, *accessoryservices.dtd*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources SYSTEM "accessoryservices.dtd">
<resources>
    <application name= ...
</resources>
```

You can also choose to add a reference to the XML Schema to the topmost part of your Accessory Service Profile XML file, *accessoryservices.xsd*:

```
<?xml version="1.0" encoding="UTF-8"?>
<resources xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="accessoryservices.xsd">
    <application name= ...
</resources>
```

Using the Eclipse IDE, on the toolbar

- Click Window > Preferences and select XML > XML files -> Validation in the left pane.
- Set options ***No grammar specified and Missing root element*** to ***Ignore.***
- Check ***Enable markup validation***.
- Click ***Apply*** and ***OK.***

When you build your application, Eclipse validates the Accessory Service Profile XML file 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***.

***Note*. Validating Accessory Service Profile description using XML Schema is more accurate than using DTD.**

A.1.4. DTD for supporting Accessory Message

The following code snippet shows the Accessory Service Profile DTD file for Accessory Message.

```
<!DOCTYPE resources [
    <!ELEMENT resources (application)>
    <!ELEMENT application (serviceProfile)+>
    <!ATTLIST application name CDATA #REQUIRED>
    <!ELEMENT serviceProfile (supportedTransports, serviceChannel*, supportedFeatures*) >
    <!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_accessory |
one_peeragent) #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>
    <!ELEMENT supportedFeatures (feature)+>
    <!ATTLIST supportedFeatures xmlns:android CDATA #IMPLIED>
    <!ELEMENT feature EMPTY>
    <!ATTLIST feature xmlns:android CDATA #IMPLIED>
    <!ATTLIST feature type (message) #REQUIRED>
    ]>
```

A.3. JAVA Reflection Construction

An Application implements 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 HelloAccessoryService() {
    // HelloAccessoryService extends SAAgent
    // HelloAccessoryServiceConnection extends SASocket
    super("HelloAccessoryService", HelloAccessoryServiceConnection.class);
}
```

An Application also needs to implement the SASocket subclass constructor for Java Reflection construction. The following example illustrates the implementation.

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

A.4. Obfuscating The Application Using *ProGuard*

When you build your application in release mode, you must add **-keep** lines in the ProGuard configuration file of your application to prevent ProGuard from renaming your SAAgent and SASocket subclasses. ProGuard is a tool integrated into the Android build system that obfuscates the code by renaming classes and methods. It's highly recommended that you read the ProGuard Manual for more information.

When running ProGuard as needed by the application, please take note of the following:

If you have been using Android Studio,

1. Upgrade Android Studio or Eclipse with Android Developer Tools (ADT) to latest if possible.
2. Open build.gradle file and change lines to enable option as shown below.

```
android {
    ......
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
    ...
}
```

3. Add the following lines to proguard-rules.pro to exclude Accessory SDK for 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, add the following line.

```
-keepattributes InnerClasses
```

5. Application needs to check these lines and its ProGuard configuration when using Accessory SDK as shown below.

```
-keepclassmembers class <Application's SASocket or SAAgent extended class>.** { *; }
-keep class <Application's SASocket or SAAgent extended class>** { *; }
```

If you have been using Eclipse with Android Developer Tools (ADT)

1. Upgrade Eclipse with Android Developer Tools (ADT) to latest.
2. Open project.properties file and change lines to enable option as shown below.

```
...
```

```
        ▪ proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt
...
```

3. Add the following lines *to* proguard-project.txt to exclude Accessory SDK for *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, add the following line.

```
-keepattributes InnerClasses
```

5. Application needs to check these lines and its ProGuard configuration when using Accessory SDK as shown below.

```
-keepclassmembers class <Application's SASocket or SAAgent extended class>.** { *; }
-keep class <Application's SASocket or SAAgent extended class>** { *; }
```

A.5. Running SAAgent In Sub-Process

Applications may choose to run SAAgent service implementation in a separate sub-process using android:process attribute. This allows the service to keep on running even if the main process has been killed. The name assigned to this attribute begins with a colon(':').

```
<application>
  ...
  <service android:name="com.samsung.android.sdk.accessory.sample" android:process=":remote” />
  ...
</application>
```

In such case, it is also necessary to export the BroadcastReceivers in the same sub-process to avoid creating multiple bindings to Samsung Accessory Service Framework.

```
<application>
    ...
    <receiver
android:name="com.samsung.android.sdk.accessory.ServiceConnectionIndicationBroadcastReceiver"
              android:process=":remote" />
        <intent-filter>
            <action android:name="com.samsung.accessory.action.SERVICE_CONNECTION_REQUESTED"/>
        </intent-filter>
    </receiver>
    <receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver"
              android:process=":remote" />
        <intent-filter>
            <action android:name="com.samsung.accessory.action.REGISTER_AGENT"/>
        </intent-filter>
    </receiver>
    ...
</application>
```

Appendix B. SDK Migration

B.1. 2.2.2 to 2.3.0 or above

B.1.1. New Changes

B.1.1.1. Intent Action Changes

Version	Intent Action Name
2.2.2 or below	android.accessory.device.action.ACCESSORY_SERVICE_CONNECTION_IND
2.3.0 or above	android.accessory.device.action.REGISTER_AFTER_INSTALL

Version	Intent Action Name
2.2.2 or below	com.samsung.accessory.action.SERVICE_CONNECTION_REQUESTED
2.3.0 or above	com.samsung.accessory.action.REGISTER_AGENT

B.1.2. Behavioral Changes

Class	Method	Application Behavior	2.2.2	2.3.0 or above	Comments
SAAgent	findPeerAgents	Call in onCreate	✓	✓	Works as is
		Call in separate thread	✓	✓	Works as is
		Call in onCreate immediately after application installed	✓	✓	**Refer to B.1.2.1.**
	requestServiceConnection	From callback thread of onFindPeerAgentResponse	✓	✓	Works as is
		Call in separate thread	✓	✓	Works as is
	acceptServiceConnection	From callback thread of onServiceConnectionRequested	✓	✓	Works as is
		Call in separate thread		✓	**Refer to B.1.2.2.**
		Do not override to automatically accept	✓	✓	Works as is
SASocket	send secureSend	From callback thread of onServiceConnectionResponse	✓	✓	Works as is
		From callback thread of onReceive	✓	✓	Works as is
		Call from one Handler thread to send to multiple channels	✓	✓	Works as is
		Call from multiple Handler threads to send in multiple channels	✓	✓	Works as is
		Call from multiple threads to send in one channel	✓		**Refer to B.1.2.3.**
	close	From callback thread of onReceive	✓	✓	Works as is

		Call in separate thread	✓	✓	Works as is
		When large data send in progress	✓	✓	Works as is

B.1.2.1. Finding Peer Agents

Application needs to handle error code (ERROR_AGENT_NOT_INITIALIZED) and wait until `SAAgent` is registered, and then reattempt to call `SAAgent.findPeerAgents()`.

```
class HelloAccessoryConsumer extends SAAgent {
    ...
    void onCreate() {
        super.onCreate();
        // Can be called in onCreate, but super must be invoked first
        findPeerAgents();
    }

    void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        // Cache the peer agent found if result is success
        if (result == PEER_AGENT_FOUND) {
            Cache(peerAgent);
        }
    }

    void onError(SAPeerAgent peerAgent, String errorMessage, int errorCode) {
        // Handle error code to call findPeerAgents API again
        if (errorCode == ERROR_AGENT_NOT_INITIALIZED) {
            findPeerAgents();
        }
    }
    ...
}
```

B.1.2.2. Accepting Service Connection

SDK 2.2.2 expects application to accept or reject the service connection from the `SAAgent.onServiceConnectionRequested()` callback. This restriction is now no longer.

```
class HelloAccessoryConsumer extends SAAgent {
    ...
    void onServiceConnectionRequested(SAPeerAgent peerAgent) {
        // Can be directly accepted here
        // acceptServiceConnectionRequest(peerAgent);
        // Alternatively, can be accepted in a separate thread
        new WorkerThread() {
            // Do your processing here
            ...
            acceptServiceConnectionRequest(peerAgent);
        }
    }
    ...
}
```

B.1.2.3. Sending data

It is necessary that application uses a single Handler thread to send over a channel. Since write is blocking, this ensures data is sent one after another. Multiple threads can be used to send over multiple channels.

```
class HelloAccessoryConsumer extends SAAgent {
    ...
    void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket socket, int result) {
        // if result is successful, cache socket for using on sending message
        if (result == CONNECTION_SUCCESS) {
            Cache(socket);
        }
        // create new handler threads for each channel
        new WorkerThread(Channel id);
    }

    void sendDataToPeer(String message) {
        // This method can be used to send data from UI thread
        message = composeMessage();
        WorkerThread.obtainMessage(message).sendToTarget();
    }

    class ServiceConnection extends SASocket {
        public void onReceive(int channelId, byte[] data) {
            // Create a new thread do your processing inside
            Create WorkerThread(
                    Parse(data);
            ...
            responseMessage = composeMessage();
            WorkerThread.obtainMessage(responseMessage).sendToTarget();
            );
        }

        void onServiceConnectionLost(int errorCode) {
            // Reset cached peer agent
            ResetCache();
        }

        void onError(int channelId, String errorString, int error) {
            // Error handling
        }`
    }

    WorkerThread extends Handler{
        void handleMessage(Message msg) {
            message = msg.obj;
            try {
                CachedSocket.Send(channel id, message);
            } catch (Exception e) {
                // Error handling
            }
        }
    }
    ...
}
```

B.2. Guide to Use Accessory Message in Legacy Application

B.2.1. Add Intent Action

Application needs to register additional intent action in Androidmanifest.xml as shown below to receive a message.

Intent Action Name
com.samsung.accessory.action.MESSAGE_RECEIVED

```
<application>
  ...
  <receiver android:name="com.samsung.android.sdk.accessory.MessageReceiver">
    <intent-filter>
        <action android:name="com.samsung.accessory.action.MESSAGE_RECEIVED" />
    </intent-filter>
  </receiver>
  ...
</application>
```

B.2.2. Create a SAMessage Instance

Application needs to create SAMessage instance in your SAAgent subclass which can be used to call send() API, when sending a message, and onReceive() callback, when receiving a message.

```
class MessageProvider extends SAAgent {
...
    void onCreate() {
        Create SA;
        try {
            Initialize SA;
        } catch (Exception e) {
            // Error Handling
        }
        ...
        SAMessage mSAMessage = new SAMessage(this) {

            @Override
            protected void onSent(SAPeerAgent peerAgent, int id) {
                // Success to send a message
            }

            @Override
            protected void onError(SAPeerAgent peerAgent, int id, int errorCode) {
                // Failure to send a message
            }

            @Override
            protected void onReceive(SAPeerAgent peerAgent, byte[] message) {
                // Receive a message
            }
```

```
        };
    ...
    }
    void sendMessage(byte[] message){
        mSAMessage.send(peerAgent, message);
    }
...
}
```

B.2.3. Add the supported feature in accessoryservices.xml

To utilize Accessory Message feature in your application, you need to add the additional filed in accessoryservice.xml. This information will be used so that Samsung Accessory Service Framework knows if your application is supporting Accessory Message feature or not. It is also guided to update the latest DTD in A.1.4.

```
<resources>
    <application name="ProviderExample">
        ...
            <supportedTransports>
                <transport type="TRANSPORT_BT"/>
            </supportedTransports>
            <serviceChannel
                id="910"
                dataRate="low"
                priority="high"
                reliability="enable"/>
            <supportedFeatures>
                <feature type="message"/>
            </supportedFeatures>
        ...
    </application>
</resources>
```

B.2.4. How to develop one application for supporting both old and new gear(*Optional)

Accessory Message is a new feature supported from Gear S3. If your application needs to use Service Connection based on SASocket for operating with old gear, you can develop one application to be able to support both old and new gear.

SA.isFeatureEnabled() will return true if Samsung Accessory Service Framework on own device supports Accessory Message and SAPeerAgent.isFeatureEnabled() will return true if Samsung Accessory Service Framework and Peer Agent on the remote device support Accessory Message. You can use SAMessage.send() only when both SA.isFeatureEnabled() and SAPeerAgent.isFeatureEnabled() return true as below. Otherwise, you should use SASocket.send() to exchange data.

```
class MessageProvider extends SAAgent {
    ...
    Create SA;
    Create SAMessage;

    SASocket mYourSubSASocket;
```

```
    SAPeerAgent peerAgent;
    ...
    String message = “Hello Message!”;

    if(SA.isFeatureEnabled(SA.SERVICE_MESSAGE) && peerAgent.isFeatureEnabled(SA.SERVICE_MESSAGE) {
        // Send a message without Service Connection.
        SAMessage.send(peerAgent, message.getBytes());
    } else {
        // After establishing Service Connection, send a message using it.
        // if you use SAMessage.send(), you will get onError() callback with the proper error code.
        mYourSubSASocket.send(CHANNEL_ID, message.getBytes());
    }
    ...
}
```

B.2.5. Replace with new SAAgent Subclass’s Constructor(*Optional)

If your application does not need to use Service Connection based on SASocket any more, you can remove SASocket subclass in your application. Then you need to replace with new SAAgent Subclass’s Constructor for which it is not necessary to use SASocket class. This will prevent any build error in your application.

Constructor for Both Legacy Service Connection and Accessory Message	Constructor only for Accessory Message

`public ConsumerService() {`
`  super(TAG, SASOCKET_CLASS);`
`}`
	`public ConsumerService() {`
	`  super(TAG);`
	`}`

B.3. Guide to Use startForeground() to upgrade for Android API 26.

B.3.1. New Changes

According to "Google's official guides", there are background execution limits in Android O. The Activity Manager will kill background services in 5 seconds after being started by the starForegroundService(). So, you have to change the background service to the foreground service by calling startForeground() with notification. It is only needed when you upgrade the target SDK of your application to Android API 26. If you build under Android API 26, you don't need to add the codes. Check the code example below and you can change those codes to whatever your application needs.

B.3.2. How to Changes

***Note*. If you don't need to keep the service in the foreground over 5 seconds, or if you build the project under Android API 26, you can erase the codes below.**

B.3.2.1. onCreate()

```
/************************************************
 * Example codes for Android O OS (startForeground) *
 ************************************************/
if (Build.VERSION.SDK_INT >= 26) {
    NotificationManager notificationManager = null;
    String channel_id = "sample_channel_01";

    if(notificationManager == null) {
        String channel_name = "Accessory_SDK_Sample";
        notificationManager = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
        NotificationChannel notiChannel = new NotificationChannel(channel_id, channel_name,
NotificationManager.IMPORTANCE_LOW);
        notificationManager.createNotificationChannel(notiChannel);
    }

    int notifyID = 1;
    Notification notification = new Notification.Builder(this.getBaseContext(),channel_id)
            .setContentTitle(TAG)
            .setContentText("")
            .setChannelId(channel_id)
            .build();

    startForeground(notifyID, notification);
}
```

B.3.2.2. onDestroy()

```
/************************************************
 * Example codes for Android O OS (stopForeground) *
 ************************************************/
if (Build.VERSION.SDK_INT >= 26) {
    stopForeground(true);
}
```

Appendix C. Using Emulator

It's helpful to remember the following tips on using Emulator:

- "***Emulator for Samsung Accessory***" application does not give any response form "Disconnected".

  ✓ Please check the following items.

  A. Are you sure to reboot after all installation has been finished?

  B. Is USB connected to the Smart device?

  C. Is Wi-Fi or mobile network available on the Smart device?

  D. Did you execute the command, "`adb -d forward tcp:8230 tcp:8230`"?

  E. Is GEAR IDE activated?

  F. Did you execute the command, "`killall sapd`"?

  G. If after doing steps A to F does not make the application work, you need to use Application Manager to clear the data of Samsung Accessory Service. And repeat the steps from A.

- Connection between Consumer and Provider application is not successful.

  ✓ Please check the following items.

  A. Does *Emulator for Samsung Accessory* application show "*Disconnected*"? If not, check the above tip.

  B. Did you add TRANSPORT_WIFI to accessoyservices.xml?

```
<resources>
    <application ...
        <serviceProfile
            ...
            <supportedTransports>
                <transport type="TRANSPORT_WIFI" />
                ...
            </supportedTransports>
            ...
        </serviceProfile>
    </application>
</resources>
```

  C. If the application is still not working, please refer to "3**. Accessory**" once again.

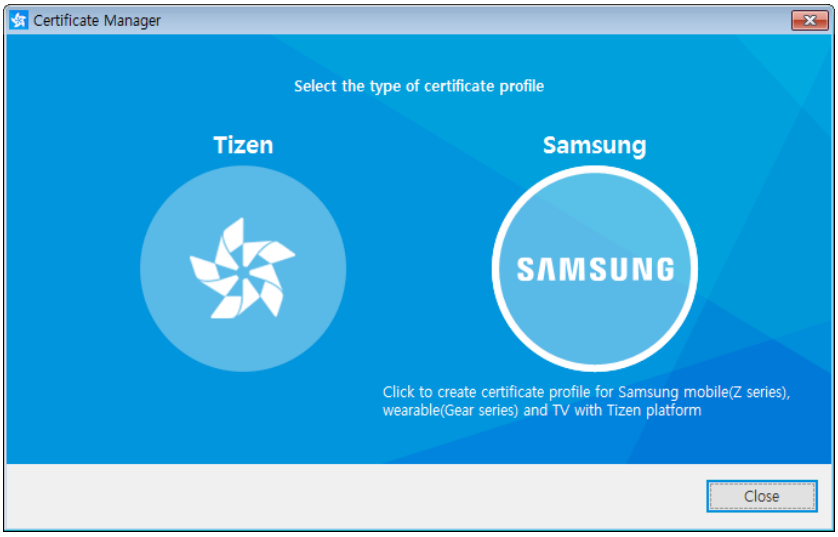
Appendix D. Creating Gear Author Certificate Using Android Keystore

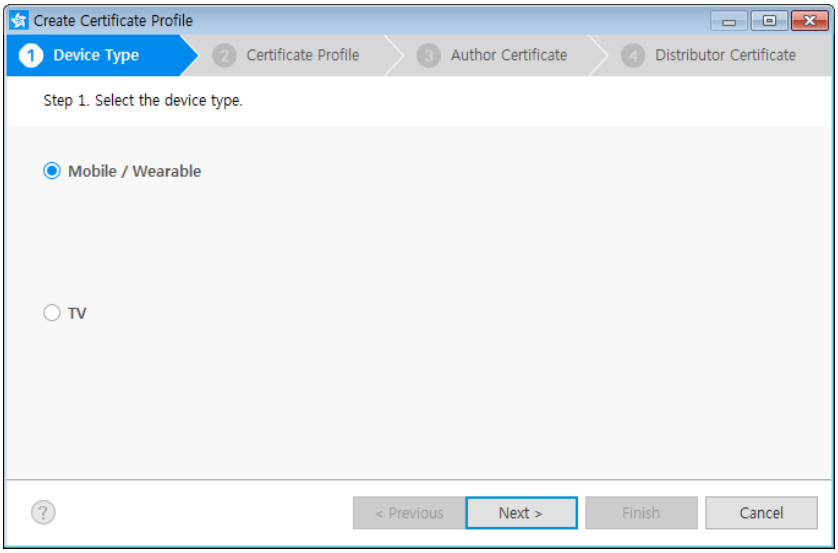
Author certificates helps in maintaining secure peer authentication between the Tizen Gear App and the Android mobile app. The Certificate Extension SDK support creation of Tizen author certificate based on Android keystore file. For instructions on installing this SDK, please refer to the link: *Tizen Extension SDK Guide*.

This section will only explain about creating author certificate using Android keystore. For a full guide on Samsung Tizen certificates creation using this SDK, please refer to the link: *Getting the Certificates*.

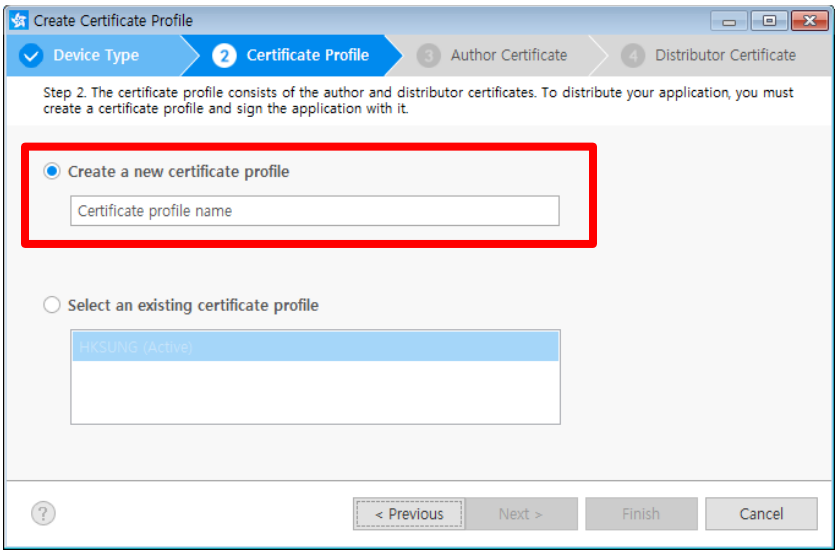
D.1. Steps

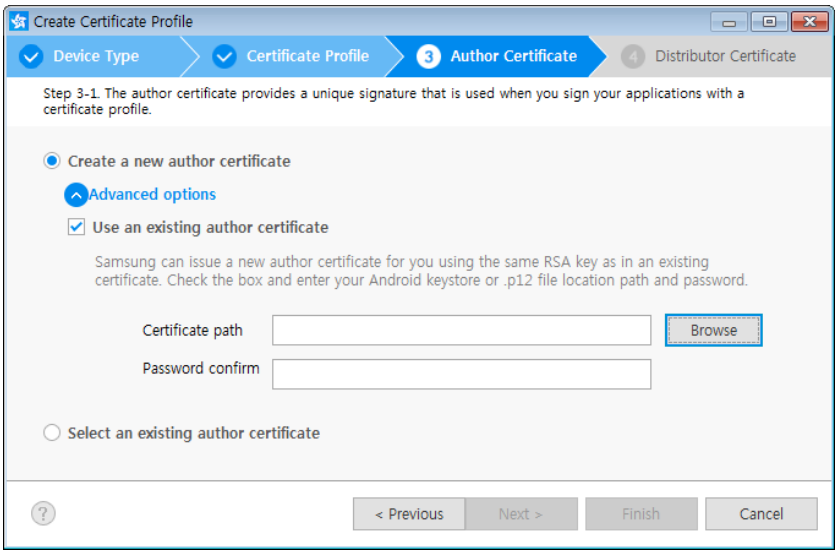
1. After creating your own Certificate Profile, you will create author certificate from **Step 2.**



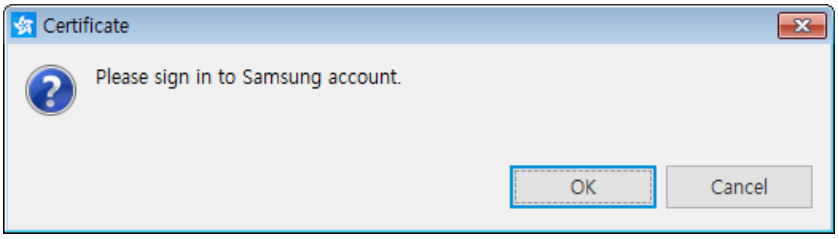


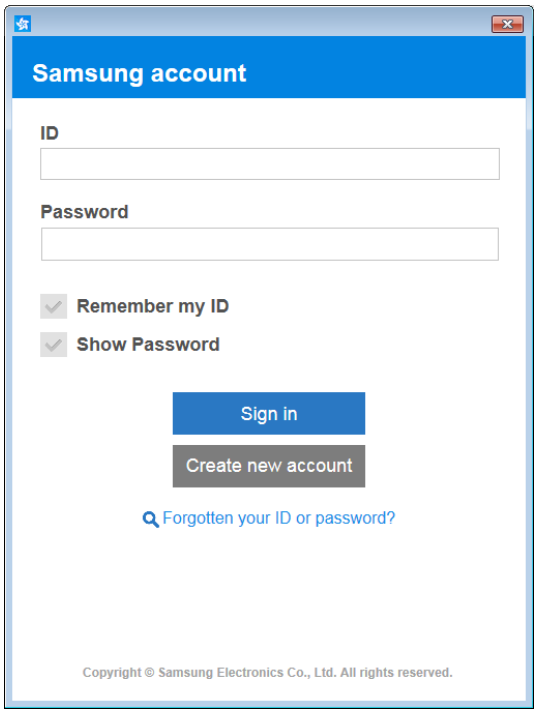
2. Select **Create a new certificate profile** to create new one. With **Advanced options menu**, you can check **Use an existing author certificate** option to select the Android keystore. Browse and select your Android keystore file, input correct password and click **Next**.



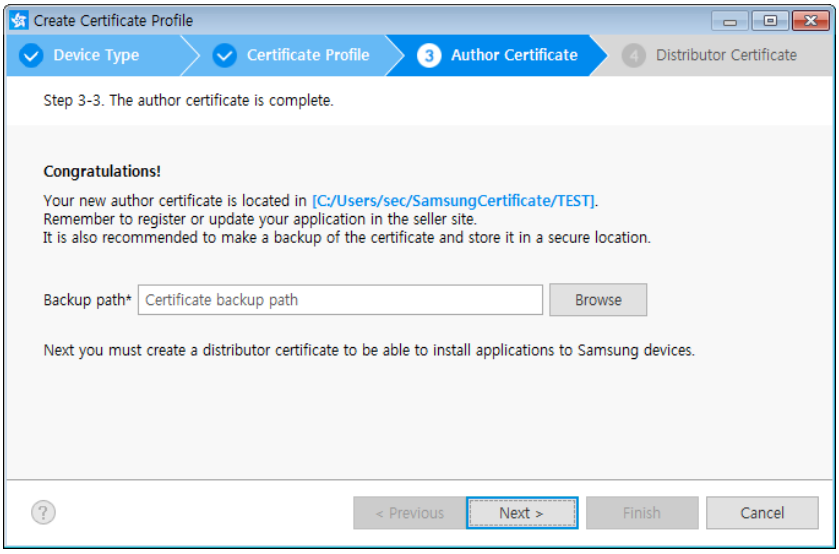


3. At the **Next** step, a Samsung Account sign-in will be opened.





4. After a few seconds, author certificate will be generated. The Next step is creating distributor certificate.



5. Click **Next**, after a few seconds you can get the distributor certificate. Then, click **Finish**.