Accessory File Transfer

Programming Guide

Version 2.1.11

Table of Contents

1. Overview

Accessory File Transfer allows you to transfer files between devices using the Samsung Accessory Service Framework. Applications can use the File Transfer Service, if they are developed using the Accessory SDK specification. File Transfer Service allows the application to send and receive files between smart devices and accessory devices through the Samsung Accessory Service Framework. Files of any size can be sent through all connectivity types supported by the Samsung Accessory Service Framework. You can develop applications, such as the Gallery Application on the Samsung Gear Application. Gallery Application can send an image to the Galaxy Note 4.

1.1. Basic Knowledge

Accessory File Transfer uses the File Transfer Service to transfer files between devices. The file is transferred on a separate service connection. The user's application does not need an existing service connection to use Accessory File Transfer. Both the sending and receiving application needs a `SAAgent` implementation, and an `EventListener` interface implemented to receive file transfer event updates (progress and completion). The sending application must know the peer (`SAPeerAgent`) to which it wants to send the file. Then, the receiving application must create a `SAFileTransfer` 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'.

1.2. Architecture

The following figure shows the Accessory File Transfer architecture.

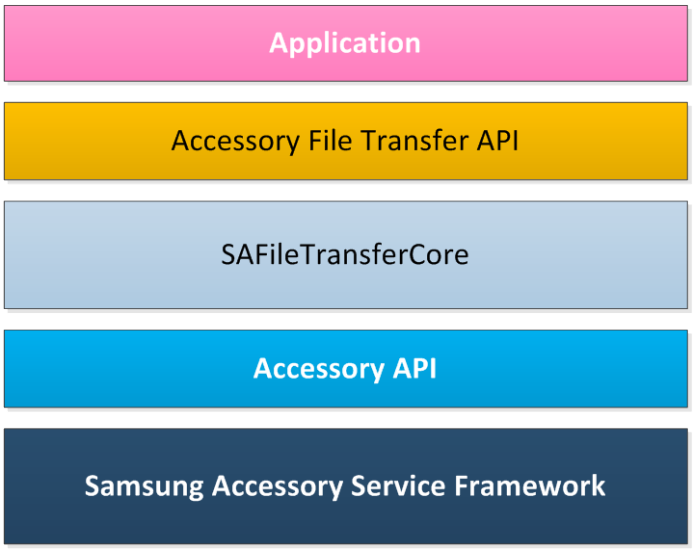

Figure 1: Accessory File Transfer architecture

1.3. Class Diagram

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

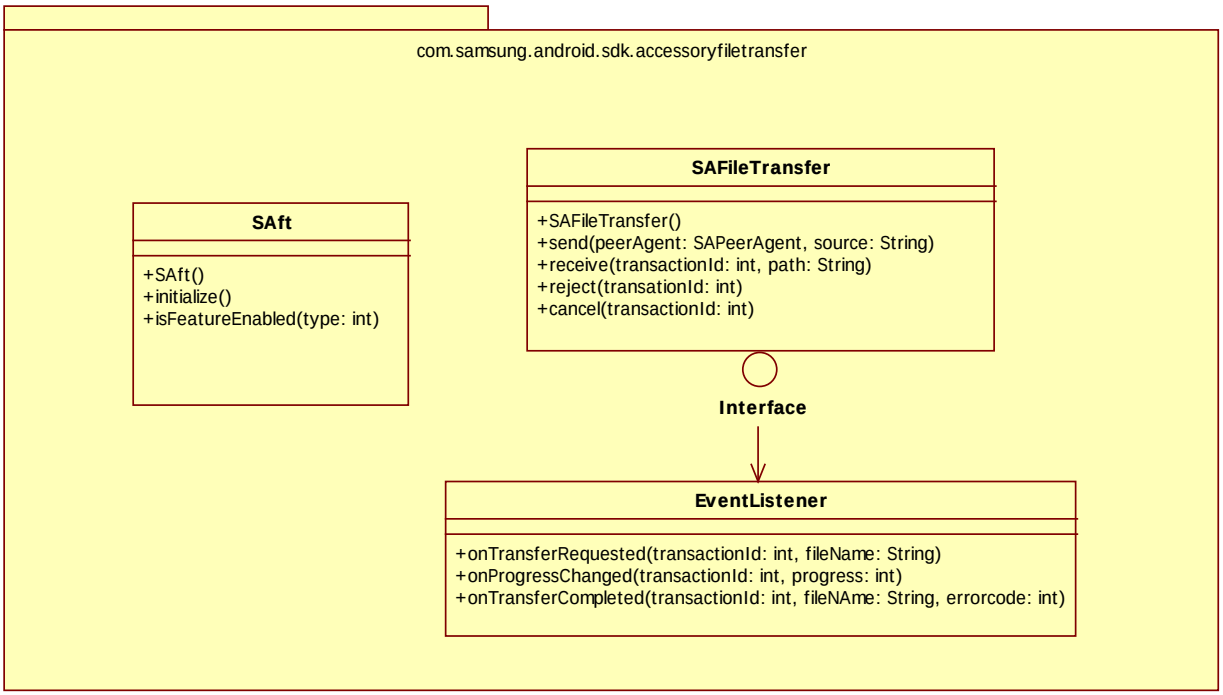

Figure 2: Accessory File Transfer classes and interfaces

The File Transfer classes and interfaces include:

- **SAft:** Initializes Accessory File Transfer.
- **SAFileTransfer:** Provides the methods. The sending and receiving applications need to use the Accessory File Transfer. Each `SAAgent` implementation can make their own `SAFileTransfer` object and call `each methods` 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 for file transfer update notifications.

1.4. Supported Platforms

- Android 4.3 or above supports Accessory.
- Devices supporting the latest Samsung Accessory Service Framework, and Galaxy Note 3 or later and Galaxy Gear devices support Accessory File Transfer.
- `SAFileTransfer` works only in the Samsung Accessory Service environment.

1.5. Supported Features

Accessory File Transfer supports the following features:

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

1.6. Components

- Components
    - accessory-v*x.y.z*.jar : Samsung Accessory SDK Library
    - accessoryfiletransfer-v*x.y.z*.jar : Samsung Accessory File Transfer SDK Library
    - sdk-v*x.y.z*.jar : Samsung SDK Library
    - android-support-v4.jar : Android Support Library
- Imported packages:
    - com.samsung.android.sdk.accessoryfiletransfer

1.7. Installing the Package for Eclipse

To install Accessory File Transfer for Eclipse:

Add the following files to the libs folder in Eclipse:

- accessory-v*2.1.11*.jar
- accessroyfiletransfer-v*2.1.11*.jar
- sdk-v*1.0.0*.jar
- android-support-v4.jar

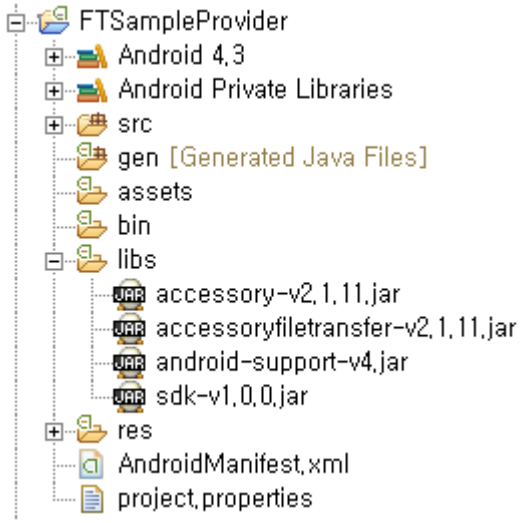

Figure 3: libs folder in Eclipse

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

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

If you don't add the permission,

o Android 4.4.2 (KitKat) and above: SecurityException is thrown and your application doesn't work.

o Prior to Android 4.4.2 (KitKat): No exception. And the application works properly.

2. Hello AccessoryFileTransfer

Hello AccessoryFileTransfer is a simple program that:

1. Initializes Accessory File Transfer.
2. Creates `SAFileTransfer`.
3. Sends a file from `HelloFileSenderService` to `HelloFileReceiverService`.

HelloFileSenderService.java:

```
public class HelloFileSenderService extends SAAgent implements DataReader {
    private final String TAG = "SampleProviderServiceImpl1";

    private SAPeerAgent mPeerAgent;
    private Handler mHandler;
    private SASocket mConnectionSocket;

    SAFileTransfer mFileTransfer;
    EventListener mCallback;
    private int tx;

    private final IBinder mBinder = new LocalBinder();

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

    /*
     * Returning binder object to activity on which it can call methods of this class
     */
    @Override
    public IBinder onBind(Intent arg0) {
        return mBinder;
    }

    public HelloFileSenderService() {
        super("---FileSenderService---", EasyServiceConnectionHandler.class);
    }

    @Override
    public void onDataAvailableonChannel(String data, long channelId) {
        /* Send msg to UI */
        if (mHandler != null) {
            Message msg = mHandler.obtainMessage(2, data);
            mHandler.sendMessage(msg);
        }
    }

    /* Called from activity to set a UI handler to post messages to */
    public void setHandler(Handler handler) {
        mHandler = handler;
}

    /*
     * Called by user to find available peers to send files to if not using a service connection
     */
    public void findPeers() {
        mPeerAgent = null;
        findPeerAgents();
    }

    public SAPeerAgent getPeerAgent() {
        return mPeerAgent;
    }
```

```
    @Override
    protected void onServiceConnectionResponse(SASocket socket, int result) {
        Log.i(TAG, "onServiceConnectionResponse : " + result);
        if (result == CONNECTION_SUCCESS) {
            mConnectionSocket = socket;
            mPeerAgent = socket.getConnectedPeerAgent();
            ((EasyServiceConnectionHandler) mConnectionSocket).setDataReader(this, mHandler);

            /* Send message to UI */
            if (mHandler != null) {
                Message msg = mHandler.obtainMessage(2, "Connection Established");
                mHandler.sendMessage(msg);
            }

        } else {
             Toast.makeText(getApplicationContext(), "Connection failed", Toast.LENGTH_SHORT).show();
        }
    }

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

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

    @Override
    protected void onError(final String errorMessage, final int errorCode) {
        new Handler(getMainLooper()).post(new Runnable() {

            @Override
            public void run() {
                Toast.makeText(getBaseContext(),
                               errorMessage + " " + errorCode, Toast.LENGTH_LONG).show();
            }
        });
    }

    /* Called for registering to File Transfer Service on demand */
    private void registerForFileTransfer() {
        SAft accessoryfiletransfer = new SAft()
        // Initialize the SAft package
        try {
            accessoryfiletransfer.initialize(this);
        } catch (SsdkUnsupportedException e) {
            // Error handling
        }

        mCallback = new EventListener() {

            @Override
            public void onProgressChanged(int transId, int progress) {
                Message msg = mHandler.obtainMessage(2, "progress" + progress);
                mHandler.sendMessage(msg);
            }

            @Override
             public void onTransferCompleted(int transId, String fileName, int errorCode) {
                mFileTransfer = null;
            }

            @Override
            public void onTransferRequested(int id, String fileName) {
            }
        };

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

```
    }

    /* Called by user to send a file to remote peer */
    protected int sendFile(String fileName) {
        if (mFileTransfer == null)
            registerForFileTransfer();
        tx = mFileTransfer.send(mPeerAgent, fileName);
        return tx;
    }

    /* Called by user to cancel an ongoing or a queued file transfer request */
    public void cancelFile() {
        if (mFileTransfer != null)
            mFileTransfer.cancel(tx);
    }
}
```

HelloFileReceiverService.java:

```
public class HelloFileReceiverService extends SAAgent implements DataReader {
    private final String TAG = "SampleConsumerServiceImpl2";

    private SAPeerAgent mPeerAgent;
    private Handler mHandler;
    private SASocket mConnectionSocket;
    private Context mContext;
    private final IBinder mBinder = new LocalBinder();

    SAFileTransfer mFileTransfer;
    EventListener mCallback;
    private int mCurrentTransaction;

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

    /* Returning binder to activity on which it can call methods of this class */
    @Override
    public IBinder onBind(Intent arg0) {
        return mBinder;
    }

    public HelloFileReceiverService() {
        super("---FileReceiverService---", EasyServiceConnectionHandler.class);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        /*
         * Registering ahead at start of service to receive incoming file
         * transfer request notifications
         */
        registerForFileTransfer();
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDataAvailableonChannel(final String data, long channelId) {
        if (mHandler != null) {
            Message msg = mHandler.obtainMessage(3, data);
            mHandler.sendMessage(msg);
        }
    }

    /* Register handler in activity to post messages to */
    public void setHandler(Handler handler) {
        mHandler = handler;
    }
```

```
    /* Called by user to find and connect to remote peer */
    public void findPeers() {
        mPeerAgent = null;
        findPeerAgents();
    }

    public void closeConnection() {
        if (mConnectionSocket != null) {
            mConnectionSocket.close();
        }
    }

    @Override
    protected void onServiceConnectionResponse(SASocket socket, int result) {
        Log.i(TAG, "onServiceConnectionResponse : " + result);
        if (result == CONNECTION_SUCCESS) {
            mConnectionSocket = socket;
            ((EasyServiceConnectionHandler) mConnectionSocket).setDataReader(this, mHandler);

            /* Send message to UI */
            if (mHandler != null) {
                       Message msg = mHandler.obtainMessage(3, "Connection Established");
                mHandler.sendMessage(msg);
            }

        } else {
            if (mHandler != null) {
                Message msg = mHandler.obtainMessage(3, "Connection failed:" + result);
                mHandler.sendMessage(msg);
            }
        }
    }

    public void registerForFileTransfer() {
        SAft accessoryfiletransfer = new SAft()
        // Initialize the SAft package
        try {
            accessoryfiletransfer.initialize(this);
        } catch (SsdkUnsupportedException e) {
            // Error handling
        }

        mCallback = new EventListener() {
            @Override
            public void onProgressChanged(int transId, int progress) {
                Message msg = mHandler.obtainMessage(3, "progress" + progress);
                mHandler.sendMessage(msg);
            }

            @Override
            public void onTransferCompleted(int transId, String fileName,
                    int errorCode) {
                Log.d(TAG, "transfer completed for transaction id : " + transId);
                Log.d(TAG, "file name : " + fileName);
                Log.d(TAG, "error code : " + errorCode);
            }

            @Override
            public void onTransferRequested(int id, String fileName) {
                Message msg = mHandler.obtainMessage(3, "File transfer request for service 2");
                mHandler.sendMessage(msg);
                Message msg1 = mHandler.obtainMessage(101, "2:" + id + ":"+ fileName);
                mHandler.sendMessage(msg1);
            }
        };

        mFileTransfer = new SAFileTransfer(HelloFileReceiverService.this, mCallback);
    }

    @Override
    protected void onServiceConnectionRequested(SAPeerAgent peerAgent) {
        Log.i(TAG, "onServiceConnectionRequested - accept");
```

```
    }

    @Override
    protected void onFindPeerAgentResponse(SAPeerAgent peerAgent, int result) {
        if (result == PEER_AGENT_FOUND) {
            mPeerAgent = peerAgent;
            requestServiceConnection(peerAgent);
        }
    }

    @Override
    protected void onError(final String errorMessage, final int errorCode) {
        if (errorCode == SAAgent.ERROR_FATAL) {
            findPeerAgents();
        }

        new Handler(getMainLooper()).post(new Runnable() {
            @Override
            public void run() {
                 Toast.makeText(getBaseContext(),errorMessage + " " + errorCode, Toast.LENGTH_LONG).show();
            }
        });

    }

    /* Called from activity by user to cancel ongoing file transfer */
    protected void cancelFile() {
        if (mFileTransfer != null)
            mFileTransfer.cancel(mCurrentTransaction);
    }

    /* Called from activity after user accepts/rejects file transfer request */
    public void receiveFile(int transId, String destPath, boolean accepted) {
        if (mFileTransfer != null) {
            if (accepted) {
                /* If user accepted */
                mFileTransfer.receive(transId, "");
            } else {
                /* If user rejected */
                mFileTransfer.reject(transId);
            }
        }
        mCurrentTransaction = transId;
    }
}
```

3. Using the SAft Class

The SAft class provides the following methods:

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

```
SAft accessoryfiletransfer = new SAft();
try {
    accessoryfiletransfer.initialize(this);
} catch (SsdkUnsupportedException e) {
    // Error handling
    e.printStackTrace();
 }
```

3.1. Using the initialize() Method

The `SAft.initialize()` method:

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

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

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

3.2. Handling SsdkUnsupportedException

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

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

- **LIBRARY_NOT_INSTALLED**: The library is not installed on the device.

3.3. Checking the Availability of Accessory File Transfer Features

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

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

The flowing types are defined in the SAft class:

- DEVICE_ACCESSORY

4. Using Accessory File Transfer

The following sections describe how to implement a sending or receiving application for File Transfer. Below are methods of each class.

SAFileTransfer

- `send(SAPeerAgent peerAgent, String source)`: Called by the sending application to send the file to the given peerAgent. Returns the transaction ID for the file transfer.
- `receive(int transactionId, String path)`: Called by the receiving application to accept an incoming file transfer request with the given ID. The file is stored at the storage path provided by the receiver.
- `reject(int transactionId)`: Called by the receiving application to reject an incoming file transfer request with the given ID.
- `cancel(int transactionId)`: Called by the sending application to cancel an ongoing or queued file transfer request, or by the receiving application to cancel an ongoing transfer.

SAFileTransfer.EventListener

- `onTransferRequested(int transactionId, String filename)`: The receiving application is notified of the incoming file transfer request with the given ID and file name.
- `onProgressChanged(int transactionId, int progress)`: Both applications are notified by the progress changes in percentage of the file transfer with the given ID.
- `onTransferCompleted(int transactionId, String filename, int errorCode)`: Both applications are notified by the file transfer completion whether success or failure cases. In case of success, the error code is given as 0, otherwise one of the error codes defined in SAFileTransfer.java is used

4.1. Creating a Sending Application

To send files:

1. Declare below permissions in AndroidManifest.xml file to use Accessory File Transfer.

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

2. Create a `SAFileTransfer` instance.

The application must have a `SAAgent` subclass and an implementation of the `eventListener` interface. The application must 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) {
        Log.d(TAG, "progress received : " + progress + " for transaction : " + transId);
    }

    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
        Log.d(TAG, "transfer completed for transaction id : " + transId);
        Log.d(TAG, "file name : " + fileName);
        Log.d(TAG, "error code : " + errorCode);
    }

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

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

3. To send a file, the application must know the Accessory Peer Agent to which it 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 `send(SAPeerAgent, String)` on the `SAFileTransfer` object.

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

The file name provided must be a fully qualified path for the file. Application can also supply a content or data URI. The data must be stored in a publicly-visible location, for example, on */mnt/sdcard*. Accessory File Transfer cannot access files residing in the application internal storage, for example, */data/data/<packageName>*. A unique transaction ID is returned to the application, which the application can retain for future reference.

***Notice: It should not be used to transfer sensitive or private information, since this method does not support any security. 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.**

4. During the file transfer, progress updates are notified with the `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.

5. The sending application can cancel the file transfer at any time by calling `cancel()`.

If a file transfer is cancelled, the `onTransferCompleted()` is called with a proper error code.

```
ACTION_INCOMING_FT_REQUEST = "com.samsung.accessory.ftconnection"
```

4.2. Creating a Receiving Application

To receive files:

1. Declare below permissions in AndroidManifest.xml file to use Accessory File Transfer.

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

2. Define a broadcast receiver (defined in SAFileTransfer.java) in the receiver application.

```
ACTION_INCOMING_FT_REQUEST = "com.samsung.accessory.ftconnection"
```

3. Declare the receiver in the manifest.

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

4. The receiver is triggered when the receiver receives a file transfer request.

The application must register with Accessory File Transfer by creating an `EventListener` instance and a `SAFileTransfer` object.

```
mCallback = new EventListener() {

    @Override
    public void onProgressChanged(int transId, int progress) {
        Message msg = mHandler.obtainMessage(3, "progress" + progress);
        mHandler.sendMessage(msg);
    }

    @Override
    public void onTransferCompleted(int transId, String fileName, int errorCode) {
        Log.d(TAG, "transfer completed for transaction id : " + transId);
        Log.d(TAG, "file name : " + fileName);
        Log.d(TAG, "error code : " + errorCode);
    }

    @Override
    public void onTransferRequested(int id, String fileName) {
        /* Notify user of incoming request */
        Message msg = mHandler.obtainMessage(3, "File transfer request received");
        mHandler.sendMessage(msg);
        Message msg1 = mHandler.obtainMessage(101, id + ":" + fileName);
        mHandler.sendMessage(msg1);
    }
};

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

The `EventListener` instance and the `SAFileTransfer` object are needed to enable the receiving application to receive incoming file transfer requests. The file transfer application notifies the receiving application about the incoming request with the `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.

5. The application must call `receive()` on the `SAFileTransfer` object to receive the file.

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

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 the 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.

The application must call `reject()` on the `SAFileTransfer` object to reject the file transfer request.

```
mFileTransfer.reject(transId);
```

6. The sending application starts sending data only after `receive()` is called.

During the file transfer, progress updates are notified with the `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 `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 error. Error code 0 refers to a successful file transfer.

The application can cancel the file transfer at any time by calling `cancel()` with the correct transaction ID.

```
mFileTransfer.cancel(mCurrentTransaction);
```

If the file transfer is cancelled, `onTransferCompleted()` is called with a requisite error code.

4.3. File Transfer Tips

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 `send()` calls are queued and serviced sequentially, even when they come 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 with the `onTransferCompleted()` callback. This is also the case when the application forgets to register the incoming file transfer request broadcast receiver or to call `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 `onTransferRequested()` callback.
- There is one binding to Accessory File Transfer per application, irrespective 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 `onTransferRequested()` callback for every registered instance during an incoming file transfer request. A suggested failsafe is to set the `SAFileTransfer` object to null in the `onDestroy()` of the `SAAgent` implementation.

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/