Device Adapter Developers Guide

Overview

The Device Adapter is a component of the Protocol Adapter software, an M2M data collection software that runs on Android (mainly mobile) devices acting as a gateway for sensor devices. The Protocol Adapter was developed as an open source component of the FI-STAR Frontend Platform, in the frame of the FI-STAR project.
It includes a Protocol Adapter Manager (PAManager) service and several Device Adapters (DA) on the same Android device. The PAManager manages the lifecycle of the DAs and provides management interfaces for the application. DAs are software components that manages low-level connections to sensor devices of the same kind and feed the collected data resulting from the measurements carried out by the sensor devices to the PAManager with a well-known structure called Observation.
Generally, DAs provide communication and interoperability at channel and syntactic level. Some operational aspects can also impact the interoperability of DAs with sensor devices.
Finally, please note that, in this document, we are taking as granted that the operating system is Android. The document is targeted to Android developers who have already have a basic knowledge of the Android architecture and of the Android Studio tool.
For more information on the Protocol Adapter architecture and internals, and for a more detailed description of the project, please refer to the Protocol Adapter Developers Guidelines.
For a technical reference, please refer to the Protocol Adapter Library's Javadoc.

Implementing a Device Adapter

A DA generally supports a class of sensor devices that have common channel, syntax and operation features. The number of existing DAs should always be kept to the minimum, so always check if there are existing DA implementations that either support the sensor devices you are interested in or could be easily extended.
In case that a new DA implementation is needed, it is paramount to define very well the class of sensor devices that are supported by the new DA implementation. The communication technology as well as applicable standards should be well identified as requirements for the design phase. Also, especially in the case of device classes sharing a common channel, it is mandatory to define a method for deciding at runtime which sensor devices are managed by this implementation of the DA.
Also keep in mind that DA implementations MUST be able to run at the same time with other DA instances governing other classes of sensor devices.

Developing a Device Adapter

In order to develop a new Device Adapter, the following steps are required:

- Create a new project in Android Studio
- Include the Protocol Adapter Library
- Understand the library's common objects
- Define the Capabilities of the Device Adapter
- Create the Discovery Responder
- Create the Device Adapter main service
- Implement the IDeviceAdapter interface

Once you created the new Device Adapter you should interact with the Protocol Adapter following this flow.

Create a new project in Android Studio

A Device Adapter is a standalone Android application which communicates with the Protocol Adapter by the means of a bound service. So, in order to create a new Device Adapter, you must create a new standard Android project in Android Studio. The package name associated with the application should be prefixed by "eu.fistar.sdcs.pa.da". Finally, you should create one bogus activity that will appear in the drawer (the list of Android applications). This is required because the Device Adapter needs to be manually started once by the user in order to correctly reply to Protocol Adapter's probes. Moreover, you should create one or more activities to allow the user to configure the Device Adapter through a GUI, if this is your desired behaviour. You can find out more on both these topics later on.

Include the Protocol Adapter Library

The first thing to do in order to work with the Protocol Adapter is to include the Protocol Adapter Library inside your project. Since Android Studio is now the only official IDE for Android (ADT plugin for Eclipse is not mantained anymore), and since we used Android Studio as our IDE when developing the entire project, we will cover here the AAR package inclusion on Android Studio. For other IDEs, you should be able to find abundant resources on-line.
Unluckily, to date (Feb. 2015) Android Studio does not offer a straightforward method to include an AAR package, but nevertheless the process is quite simple.

First of all you should copy the AAR file inside the `libs` directory of your app module. You can do this by simply copy-pasting the file from your file manager directly in the IDE.

Then you should edit the build.gradle file of your app module and add these lines. If the `dependencies` section already exists (this is the most common case) just add the `compile` line to the existing section.

```
// This entry is added in order to use the AAR library
repositories {
    flatDir { dirs 'libs' }
}
```

```
dependencies {
  // This is the entry that adds the dependency from the AAR library
  compile 'eu.fistar.sdcs.pa.common:protocol-adapter-lib:3.4.4@aar'
}
```

The string passed as an argument of `compile` is made of 4 parts: the package name, the file name, the library version and the @aar suffix. To date (Feb. 2015) 3.4.4 is the latest version of the library, but you should take care of inserting the right version of the library here, the one that matches with the file you just copied in the project.
Finally, you should force a sync of the project with gradle files. You can do this by clicking the specific button.
If you want to use a directory other than `libs` just use the same name in the `build.gradle` file.

Understand the library's common objects

In the Protocol Adapter Library we provided some classes to help represent the devices and handle the data. So far (v3.4.4), we have four different objects:

- **DeviceDescription**, an object representing a sensor device and all of its characteristics, including the sensors equipped. You must create one instance of this object for every device that it's connected with the Device Adapter. The instance of this object associated with a connected device must be a representation of that device.
- **SensorDescription**, an object representing a sensor and all of its characteristics, such as the sensor name, the property name and the measurement unit. You must create one instance of this object for every property provided by a connected device. All the SensorDescription objects associated with a connected device must be inserted inside the DeviceDescription object associated with that device.
- **Observation**, an object representing a measurement, containing data samples and a reference to the device and sensors. You must create one instance of this object every time you receive a measurement from a device. That instance must contain the received data plus the reference to the property which that data is associated to. Moreover, when you send this object to the Protocol Adapter, you must include the instance of DeviceDescription object associated with the device that produced the data.
- **Capabilities**, an object containing information about the abilities and requirements of the Device Adapter and its underlying technology. It contains crucial information for automating the interaction between the PA and the DA. The Capabilities object, being associated directly with the Device Adapter, must be created only once per DA. It should be provided in reply to PA's probes or upon direct request via the API. The Capabilities object will be discussed in the next chapter.

Apart from Capabilities, the other three objects are used regularly to exchange data between DAs and the PA.

The provided classes should be suitable for most cases, so good chances are that you can just use these objects directly, without further modifications. However you could have the need to extend one or more of these classes in order to add additional parameters or methods that you may find useful during the lifecycle of your Device Adapter.
If you do so, you must keep in mind two things. First, the Protocol Adapter's methods only accept the original objects as parameters, not subclasses. This means that in every case you must provide in your subclasses all the basic information required by the original objects, while all the other information will be lost during the conversion to the superclass. Second, while the original library's objects are optimised to flow through AIDL interfaces, their subclasses may be not. So, please DO NOT send your subclasses to the Protocol Adapter; instead create a new instance of the superclass passing the instance of your subclass as an argument to the constructor; then send the newly created instance to the Protocol Adapter. Please note that all these three objects have a constructor that takes an object of the same type as a parameter, so you can safely pass a subclass to it and all the important data will be extracted.

Define the Capabilities of the Device Adapter

The Capabilities object is used to describe the capabilities of the Device Adapter. The DA must create this object when it starts (usually defining it as a constant) and must provide it to the Protocol Adapter in reply to PA's probes or upon direct request via the API.
The Capabilities object has these fields that can be set to influence the behaviour of the interaction between the Protocol Adapter and the Device Adapter:

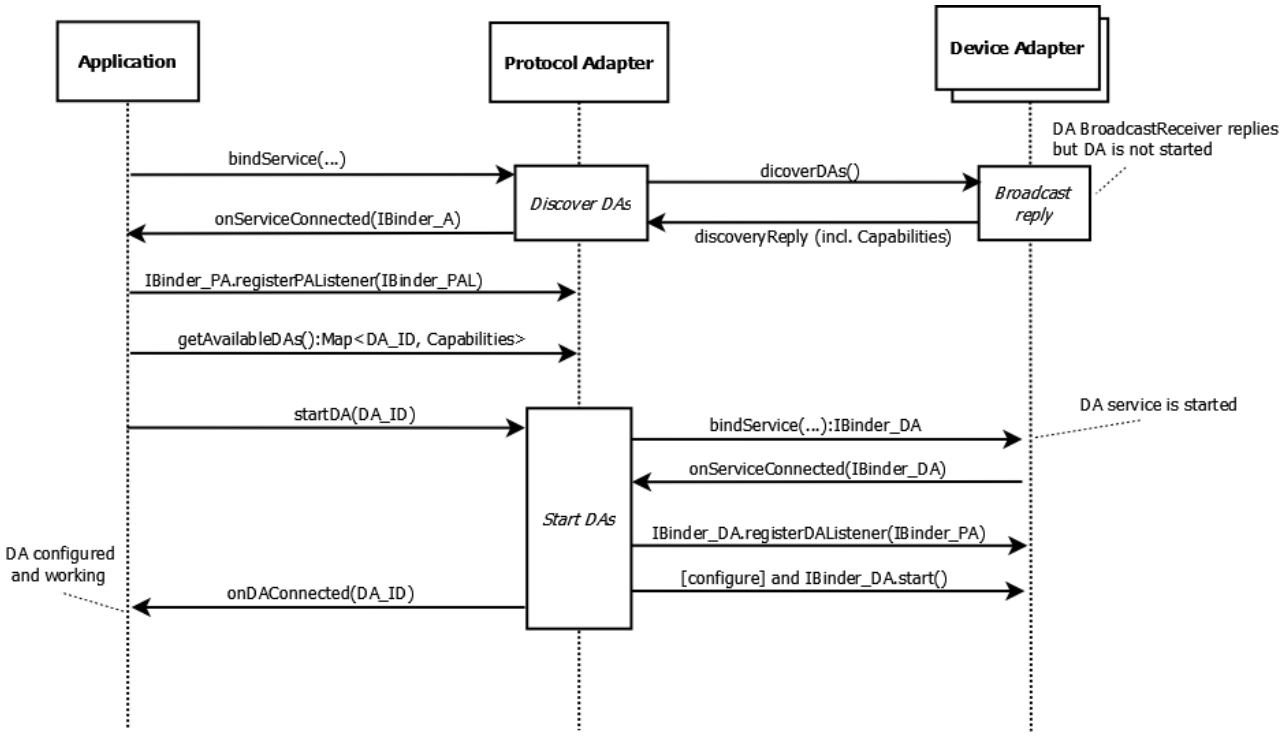
- boolean **blacklistSupport** - States whether Device Adapter supports blacklist or not. If true, the Device Adapter must provide working implementation of the following methods: *addDeviceToBlackList()*, *removeDeviceFromBlacklist()*, *getBlacklist()*, *setBlacklist()*.
- boolean **whitelistSupport** - States whether Device Adapter supports whitelist or not. If true, the Device Adapter must provide working implementation of the following methods: *addDeviceToWhiteList()*, *removeDeviceFromWhitelist()*, *getWhitelist()*, *setWhitelist()*.
- String **guiConfigurationActivity** and String **guiConfigurationActivityPackage** - These are, respectively, the names of the configuration activity (including the full classpath) and its package. If these parameters are not null, then the Device Adapter will supports configuration. You can check if a Device Adapter supports configuration and retrieve its configuration activity by invoking, respectively, the *isGuiConfigurable()* and *getConfigActivityName()* methods of its Capabilities object. The configuration activity is used as the entry point for the Device Adapter GUI-based configuration. This activity will be called by the Protocol Adapter when a user wants to configure the DA, and the design of its behaviour is completely left to the DA's developers.
- int **deviceConfigurationType** - Retrieve the information about whether the configuration is supported by the Device Adapter and, if so, what kind of configuration it supports. If supported, the Device Adapter must provide working implementation of

the following method: *setDeviceConfig()*.
Acceptables values are between 0 and 3:
    - 0 = CONFIG_NOT_SUPPORTED, configuration is not supported
    - 1 = CONFIG_RUNTIME_ONLY, configuration can only be made at runtime
    - 2 = CONFIG_STARTUP_ONLY, configuration can only be made upon startup
    - 3 = CONFIG_STARTUP_AND_RUNTIME, configuration can be made both at runtime or upon startup
- boolean **commandSupport** - States whether the Device Adapter supports the sending of commands. If supported, the Device Adapter must provide working implementation of the following methods: *execCommand()*, *getCommandList()*.
- boolean **connectionInitiator** - States whether the Device Adapter is the initiator of the communication with the devices (it connects to the devices) or if it's the target (the devices automatically connect to it). If true, the Device Adapter must provide working implementation of the following methods: *connectDev()*, *forceConnectDev()*, *disconnectDev()*, *getConnectedDevices()*.
- boolean **detectDeviceSupport** - States whether the Device Adapter supports the detection of nearby devices. If supported, the Device Adapter must provide working implementation of the following methods: *detectDevices()*.
- boolean **previousPairingNeeded** - States whether the Device Adapter needs the devices to be already paired in order to use them.
- boolean **monitorDisconnectionSupport** - States whether the Device Adapter can monitor the disconnection of the devices. If supported, the Device Adapter must call the following method of the IProtocolAdapter interface upon device disconnection: *deviceDisconnected()*.
- String **friendlyName** - The Friendly Name of the Device Adapter, one that is both human readable and self-explanatory.
- String **actionName** - The Action Name that the Protocol Adapter have to use in order to bind the Device Adapter Service. Please note that since Android 5.0 (Lollipop) implicit intents are not supported anymore to bind services, so the action name specified here must be the classname of the DA service including the full classpath (eg. the.entire.classpath.ServiceClassName).
- String **packageName** - The Package Name of the Device Adapter.
- boolean **availableDevicesSupport** - States whether the Device Adapter has the ability to recognise if it can handle a device or not, and consequently if it can provide the list of the Available Devices or not. If supported, the Device Adapter must provide working implementation of the following methods: *getPairedDevicesAddress()*.

Once the capabilities are defined, the implementation of all unsupported methods should be modified to throw an UnsupportedOperationException.

Create the Discovery Responder

Upon its start the Protocol Adapter will start a discovery phase to find all available DAs in the system. This process is carried out by sending a broadcast discovery Intent with a specific action (eu.fistar.sdcs.pa.discoveryrequest) to which all DA implementations must reply with a reply Intent. The Action name of this Intent is sent in the replyAction field of the Extras in the discovery Intent, while the PA package name used by the DAs to limit the reply Intent is specified as a constant in the library. All the DAs must implement a BroadcastReceiver and register that receiver in their manifest along with an IntentFilter for the discovery Intent. This BroadcastReceiver will then produce and send the reply intent. The DAs must include their Capabilities objects in the daCapabilities field of the Extras of the reply Intent, and their ID (which is actually their package name) in the daId field of the Extras.



For its nature, the BroadcastReceiver should be placed inside a separate class for the IntentFilter to be declared in the manifest. Moreover, the file containing the BroadcastReceiver's class is a really good place to store the Capabilities associated with the Device Adapter. Here you can find a sample implementation of the BroadcastReceiver and the Capabilities object.

```
package eu.fistar.sdcs.pa.da.zephyrbh;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
```

```
import eu.fistar.sdcs.pa.common.Capabilities;
import eu.fistar.sdcs.pa.common.PAAndroidConstants;

/**
 * This class implements the responder to the discovery performed by Protocol Adapter.
 */
public class DiscoveryResponder extends BroadcastReceiver {

    // Create the Capabilities object of the Device Adapter
    public final static Capabilities CAPABILITIES = new Capabilities(
            CapabilitiesConstants.CAP_BLACKLIST_SUPPORT,
            CapabilitiesConstants.CAP_WHITELIST_SUPPORT,
            CapabilitiesConstants.CAP_GUICONFIGURATION_ACTIVITY,
            CapabilitiesConstants.CAP_GUICONFIGURATION_ACTIVITY_PACKAGE,
            CapabilitiesConstants.CAP_DEV_CONF_TYPE,
            CapabilitiesConstants.CAP_COMMANDS_SUPPORT,
            CapabilitiesConstants.CAP_DETECT_DEV_SUPPORT,
            CapabilitiesConstants.CAP_PREVIOUS_PAIRING_NEEDED,
            CapabilitiesConstants.CAP_MONITOR_DISCONN_SUPPORT,
            CapabilitiesConstants.CAP_FRIENDLY_NAME,
            CapabilitiesConstants.CAP_ACTION_NAME,
            CapabilitiesConstants.CAP_PACKAGE_NAME,
            CapabilitiesConstants.CAP_CONNECTION_INITIATOR,
            CapabilitiesConstants.CAP_AVAILABLE_DEVICES_SUPPORT
    );

    @Override
    public void onReceive(Context context, Intent intent) {
        // Extract reply action from the received Intent
        String replyAction = intent.getStringExtra(PAAndroidConstants.DA_DISCOVERY.BUNDLE_REPACT);

        // Create a new Intent using the retrieved Reply Action
        Intent replyIntent = new Intent();
        replyIntent.setAction(replyAction);

        // Limit the intent to the package of PA
        replyIntent.setPackage(PAAndroidConstants.PA_PACKAGE);

        // Set the DA ID and DA Capabilities as extras in the Intent
        replyIntent.putExtra(PAAndroidConstants.DA_DISCOVERY.BUNDLE_DAID, CapabilitiesConstants.DA_ID);
        replyIntent.putExtra(PAAndroidConstants.DA_DISCOVERY.BUNDLE_DACAP, CAPABILITIES);

        // Reply to the Discovery Request sent by the Protocol Adapter
        context.sendBroadcast(replyIntent);
    }

    /**
     * Class containing only the Capabilities related constants
     */
```

```
    static class CapabilitiesConstants {

        // Generic Constants
        public static final String DA_ID = "eu.fistar.sdcs.pa.da.bogusda";

        // Capabilities constants, specific for each DA
        public static final boolean CAP_BLACKLIST_SUPPORT = true;
        public static final boolean CAP_WHITELIST_SUPPORT = true;
        public static final String CAP_GUICONFIGURATION_ACTIVITY = DA_ID + “ConfigActivity”;
        public static final String CAP_GUICONFIGURATION_ACTIVITY_PACKAGE = DA_ID;
        public static final int CAP_DEV_CONF_TYPE = Capabilities.CONFIG_STARTUP_AND_RUNTIME;
        public static final boolean CAP_COMMANDS_SUPPORT = true;
        public static final boolean CAP_DETECT_DEV_SUPPORT = false;
        public static final boolean CAP_PREVIOUS_PAIRING_NEEDED = true;
        public static final boolean CAP_MONITOR_DISCONN_SUPPORT = true;
        public static final String CAP_FRIENDLY_NAME = "Bogus Device Adapter";
        public static final String CAP_ACTION_NAME = DA_ID + ".BogusDeviceAdapter";
        public static final String CAP_PACKAGE_NAME = DA_ID;
        public static final boolean CAP_CONNECTION_INITIATOR = true;
        public static final boolean CAP_AVAILABLE_DEVICES_SUPPORT = true;

    }
}
```

Finally, you should add a block like this in the application’s section of the Manifest.

```
<receiver
        android:name=”eu.fistar.sdcs.pa.da.bogusda.DiscoveryResponder”
        android:enabled=”true”
        android:exported=”true”>
        <intent-filter>
                <action android:name="eu.fistar.sdcs.pa.discoveryrequest" />
        </intent-filter>
</receiver>
```

Please note that the BroadcastReceiver (and in turn the whole discovery process) is the reason why every DA needs to be manually started by the user at least once after the installation: infact there is an Android security policy that prevents every application to correctly receive and process broadcast Intents unless the said application has been manually started by the user at least once.

Create the Device Adapter main service

The Device Adapter must contain a main Android service, because it’s bound by the Protocol Adapter. So, you should create a new service in the root of the main package of your application.

Android Studio offers a wizard to do such a thing, that can be found right clicking on the package name and selecting New -> Service -> Service. Feel free to choose the name you prefer, but make sure that the “Enabled” and “Exported” checkbox are checked (or, if you created the service manually, make sure that in the manifest the service is enabled and exported).

Finally, make sure to NOT declare any IntentFilters for the service in the manifest, since it will be started with an explicit Intent by the Protocol Adapter.

Implement the IDeviceAdapter interface

The communication between Protocol Adapter and Device Adapter is made using AIDL. All the AIDL interfaces needed for this communication are included in the library. However in the Device Adapter service you must provide an implementation of the IDeviceAdapter interface and you must override the onBind method to return that implementation to the Protocol Adapter. This implementation will be used by the Protocol Adapter to interact with the Device Adapter. Here you can find a sample implementation that you can copy/paste and personalize to your needs. Please refer to the library’s javadoc for further information on what single methods do.

```
/**
 * Implementation of the Device Adapter API (IDeviceAdapter) to pass to the Protocol Adapter
 */
private final IDeviceAdapter.Stub paEndpoint = new IDeviceAdapter.Stub() {

    /**
     * Receive a binder from the Protocol Adapter representing its interface.
     *
     * @param pa The Protocol Adapter Binder
     */
    @Override
    public void registerDAListener(IBinder pa) {
        paApi = IDeviceAdapterListener.Stub.asInterface(pa);
    }

    /**
     * Return a list of all the devices connected at the moment with the Device Adapter.
     *
     * @return A list of the devices connected at the moment with the Device Adapter
     */
    @Override
    public List<DeviceDescription> getConnectedDevices() throws RemoteException {
        List<DeviceDescription> connDev = new ArrayList<DeviceDescription>();

        // Create a list of DeviceDescription starting from a list of subclasses of DeviceDescription
        for (CustomDevice dev : connectedDevices.values()) {
            connDev.add(new DeviceDescription(dev));
```

```
            }

            // Return the list
            return connDev;
        }

        /**
         * Return a list of the Device IDs of all the devices paired with the smartphone and managed
         * by the specific Device Adapter.
         *
         * @return A list of devices paired and handled by the Device Adapter
         */
        @Override
        public List<String> getPairedDevicesAddress() throws RemoteException {
            return pairedDevices;
        }

        /**
         * Return a list of devices that can be detected with a scanning.
         */
        @Override
        public List<String> detectDevices() throws RemoteException {
            throw new UnsupportedOperationException("Method not supported by " +
DiscoveryResponder.CapabilitiesConstants.CAP_FRIENDLY_NAME + "!");
        }

        /**
         * Set the specific configuration of a device managed by the Device Adapter passing a data
         * structure with key-value pairs containing all possible configuration parameters and
         * their values, together with the device ID. This should be done before starting the Device
         * Adapter, otherwise standard configuration will be used. Depending on capabilities, this
         * could also be invoked when the DA is already runing.
         *
         * @param config
         *                The configuration for the device in the form of a key/value set (String/String)
         *
         * @param devId
         *                The device ID (the MAC Address)
         */
        @Override
        public void setDeviceConfig(Map config, String devId) throws RemoteException {

            // Put your implementation here!

        }

        /**
         * Return the object describing the capabilities of the DA. The implementation for this
         * method is mandatory.
         *
```

```
     * @return An instance of the Capabilities object containing all the capabilities of the
     * device
     */
    @Override
    public Capabilities getDACapabilities() throws RemoteException {
        return DiscoveryResponder.CAPABILITIES;
    }

    /**
     * Start the Device Adapter operations.
     */
    @Override
    public void start() throws RemoteException {

        // Perform your initialization operation here!

    }

    /**
     * Stop the Device Adapter operations. This will not close or disconnect the service.
     */
    @Override
    public void stop() throws RemoteException {

        // Perform a general clean and final operations here!

    }

    /**
     * Connect to the device whose device ID is passed as an argument.
     *
     * @param devId The device ID
     */
    @Override
    public void connectDev(String devId) throws RemoteException {

        // Check if the device is paired and supported
        if (!deviceIsInTheList(devId, pairedDevices)) throw new IllegalArgumentException("The device " + devId + " is
not paired or not supported by Device Adapter!");

        // Connect to the device using the forceConnectDev
        forceConnectDev(devId);

    }

    /**
     * Force connection to the device whose devID is passed as an argument. This method can be
     * used to connect a supported device that, for some reasons, is not recognised by the DA.
     * Remember that, depending on the capabilities, you may want to perform a check here to
     * find out if the device is allowed to connect based on the value of blacklist/whitelist.
```

```
 * Moreover, depending on capabilities, you may want to perform a check here to find out
 * if there is a configuration for the device to connect to, or, if it is appropriate, to
 * provide default configuration.
 *
 * @param devId The device ID
 */
@Override
public void forceConnectDev(String devId) throws RemoteException {

    // Check whether the device is allowed to connect based on blacklist/whitelist

    // Check if there is a configuration entry for this device, otherwise use the default configuration

    // Perform the low level connection
}

/**
 * Disconnect from the device whose DevID is passed as an argument.
 */
@Override
public void disconnectDev(String devId) throws RemoteException {

    // Put your implementation here!

}

/**
 * Add a device to the Device Adapter whitelist, passing its device ID as an argument.
 * Note that this insertion will persist, even through Device Adapter reboots, until
 * the device it's removed from the list. Every device adapter should check the format
 * of the address passed as an argument and, if it does not support that kind of
 * address, it can safely ignore that address.
 *
 * @param devId The Device ID
 */
@Override
public void addDeviceToWhitelist(String devId) throws RemoteException {

    // Put your implementation here!

}

/**
 * Remove from the whitelist the device whose device ID is passed as an argument.
 * If the device is not in the list, the request can be ignored.
 *
 * @param devId The Device ID
 */
@Override
public void removeDeviceFromWhitelist(String devId) throws RemoteException {
```

```
    // Put your implementation here!

}

/**
 * Retrieve all the devices in the whitelist of the DA. If there's no devices, an
 * empty list is returned.
 *
 * @return The list containing all the devices ID in the whitelist
 */
@Override
public List<String> getWhitelist() throws RemoteException {
    // Just return the whitelist
    return whitelist;
}

/**
 * Set a list of devices in the whitelist all together, passing their device IDs as an
 * argument. Note that this insertion will persist, even through Device Adapter reboots,
 * until the devices are removed from the list. Every device adapter should check the format
 * of the address passed as an argument one by one and, if it does not support that kind of
 * address, it can safely ignore that address.
 *
 * @param devicesId A list containing all the devices ID to set in the whitelist
 */
@Override
public void setWhitelist(List<String> devicesId) throws RemoteException {

    // Put your implementation here!

}

/**
 * Add a device to the Device Adapter blacklist, passing its device ID as an argument.
 * Note that this insertion will persist, even through Device Adapter reboots, until
 * the device it's removed from the list. Every device adapter should check the format
 * of the address passed as an argument and, if it does not support that kind of
 * address, it can safely ignore that address.
 *
 * @param devId The Device ID
 */
@Override
public void addDeviceToBlackList(String devId) throws RemoteException {

    // Put your implementation here!

}

/**
```

```
 * Remove from the blacklist the device whose device ID is passed as an argument.
 * If the device is not in the list, the request can be ignored.
 *
 * @param devId The Device ID
 */
@Override
public void removeDeviceFromBlacklist(String devId) throws RemoteException {

    // Put your implementation here!

}

/**
 * Retrieve all the devices in the blacklist of the DA. If there's no devices, an
 * empty list is returned.
 *
 * @return A list containing all the devices ID in the blacklist
 */
@Override
public List<String> getBlacklist() throws RemoteException {
    // Just return the blacklist
    return blacklist;
}

/**
 * Set a list of devices in the blacklist all together, passing their device IDs as an
 * argument. Note that this insertion will persist, even through Device Adapter reboots,
 * until the devices are removed from the list. Every device adapter should check the format
 * of the address passed as an argument one by one and, if it does not support that kind of
 * address, it can safely ignore that address.
 *
 * @param devicesId A list containing all the devices ID to set in the blacklist
 */
@Override
public void setBlackList(List<String> devicesId) throws RemoteException {

    // Put your implementation here!

}

/**
 * Return all the commands supported by the Device Adapter for its devices.
 *
 * @return A list of commands supported by the Device Adapter
 */
@Override
public List<String> getCommandList() throws RemoteException {

    // Put your implementation here!
```

```
    }

    /**
     * Execute a command supported by the device. You can also specify a parameter, if the command
     * requires or allows it.
     *
     * @param command The command to execute on the device
     * @param parameter The optional parameter to pass to the device together with the command
     * @param devId The Device ID
     */
    @Override
    public void execCommand(String command, String parameter, String devId) throws RemoteException {

        // Put your implementation here!

    }
};
```

Please note that you are highly encouraged to use instance variables to store information about the status of devices and lists. In particular you should:

- use a final static Map<String, String> to store a default configuration (name/value of parameters), if this is coherent with your capabilities;
- use a IDeviceAdapterListener field to store the reference to the Protocol Adapter's API;
- use a final Map<String, Map<String, String>> to store the configuration of the devices;
- use a final Map<String, DeviceDescription> to store the device ID and the DeviceDescription of every connected device;
- use a final List<String> to store the device IDs of the paired devices available in the system, if this is coherent with your capabilities;
- use a final List<String> to store the device IDs of the devices in blacklist, if this is coherent with your capabilities;
- use a final List<String> to store the device IDs of the devices in whitelist, if this is coherent with your capabilities.

Interaction flow with the Protocol Adapter

Your Device Adapter is now ready to communicate and interact with the Protocol Adapter. This interaction follows a certain flow that will be explained here.

First of all, the PA is in charge of the DA management and also it's in charge of starting the DAs. The PA (or the application) can decide to stop any DA, causing it to be killed by the Android OS. A DA should never start itself and should not keep itself running artificially.

However it's the Device Adapter that is in charge for the communication of the data. It handles its devices its own way and it's the only responsible for the communication with them. So it

can do whatever it needs or whatever it wants when communicating with the devices, as long as it offers some basic functionalities to the Protocol Adapter and it sticks to a certain flow.

This is the flow:

- Upon the start of the Protocol Adapter, the discovery process is performed. This process is described in the previous chapters.
- At a certain point the Protocol Adapter starts a Device Adapter. To do this, it binds to the Device Adapter main service; when the service is bound, the Protocol Adapter will pass the reference to its API to the Device Adapter, will restore the whitelist and blacklist inside the Device Adapter (if supported by Capabilities), and finally will invoke the start() method of the Device Adapter. At this point the initialization phase is over and the Protocol Adapter and Device Adapter are ready to communicate.
- Depending on the capabilities, the connection to devices can be spontaneous (the devices decide when to connect) or generated by an explicit request of the Protocol Adapter (and in turn, of the Application) which will call the “connectDev” or the “forceConnectDev” methods of the Device Adapter. No matters what’s the case, when a connection is established with a sensor device, the Device Adapter must instantiate a new DeviceDescription object. This object contains all the relevant information about the newly connected sensor device, including a list of SensorDescription objects. Each SensorDescription object, in turn, contains all the relevant information about one of the properties exposed by the sensor device. Then the Device Adapter must call the “registerDevice” method of the Protocol Adapter, passing the DeviceDescription object and the Device Adapter’s ID as arguments. This makes the Protocol Adapter register the newly connected device and its sensors and notify the Application about this event. At this point, it’s possible for the Device Adapter to push data coming from the sensor device.
- This step is optional. If not all properties of the sensor device are known at connection time, it’s always possible to register a new property later calling the “registerDeviceProperties” method of the Protocol Adapter. In this case you must provide an updated version of the previously created instance of DeviceDescription object which must now include the new SensorDescription objects. *Note: only for Bluetooth devices, the deviceID should be the MAC address of the device, so deviceId has the same value as address in ISensorDescription*.
- When data is available from the sensor device, the Device Adapter must create a new instance of the Observation object for every sensor that is sending data and put them in a list. The Observation object contains all the relevant data of the incoming measurement. Then the Device adapter must call the “pushData” method of the Protocol Adapter passing the list of Observation and the previously created instance of DeviceDescription as arguments. This step must be repeated every time a new measurement is available during the lifecycle of the association between the Device Adapter and the sensor device.
- If the connection with the device is terminated unexpectedly, the Device Adapter must take care of notifying the Protocol Adapter about this event and wait for the device to

connect back (or, depending on the capabilities, try to restore the connection). The notification is performed by invoking the “deviceDisconnected” method of the Protocol Adapter.

- Finally, when the connection between the Device Adapter and the sensor device is ended, the Device Adapter must call the “deregisterDevice” method of the Protocol Adapter passing the previously created instance of DeviceDescription as an argument.
- At any time after the initialization and coherently with the capabilities, the Protocol Adapter can perform one of these action with the Device Adapter:
    - Retrieving the list of devices connected with the Device Adapter by invoking the “getConnectedDevices” method.
    - Retrieving the list of paired devices available in the system by invoking the “getPairedDeviceAddress” method.
    - Requesting the detection of nearby supported devices by invoking the “detectDevices” method.
    - Setting the configuration for a particular device by invoking the “setDeviceConfig” method.
    - Retrieving the Capabilities object of the Device Adapter by invoking the “getDACapabilities” method.
    - Getting or setting the whitelist/blacklist of the Device Adapter by invoking the appropriate methods.
    - Adding or removing a device from the whitelist/blacklist of the Device Adapter by invoking the appropriate methods.
    - Retrieving the list of commands supported by the Device Adapter by invoking the “getCommandList” method.
    - Executing a command on a specific device by invoking the “execCommand” method.
- At the end of the interaction between the Protocol Adapter and the Device Adapter, a termination phase is performed. That phase is started by the Protocol Adapter by invoking the “stop” method of the Device Adapter; at this point, the Device Adapter should close all the connections that are still opened with its sensor devices and perform a general clean of the resources; after that, the Protocol Adapter will unbind the Device Adapter main service, and no more interaction will happen unless a new binding is performed.