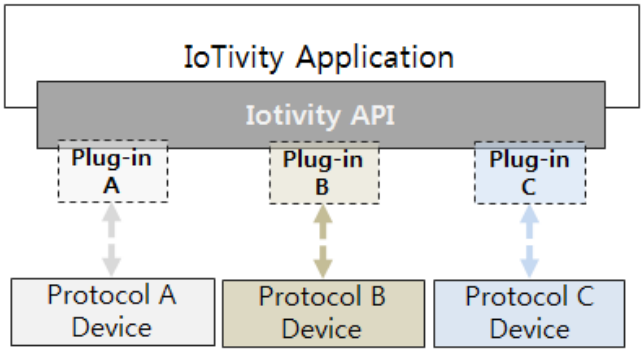
Iotivity Programmer's Guide
– Protocol Plug-in Manager for Tizen

1 CONTENTS

2 OVERVIEW

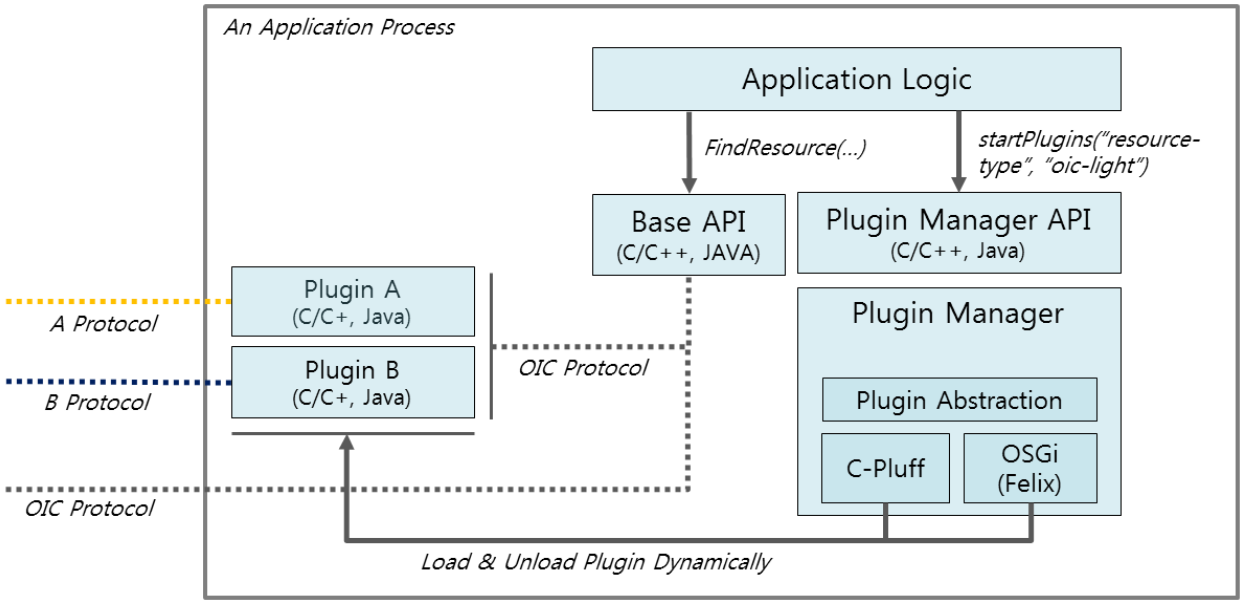
This guide will help you to use protocol plugins. Using protocol plugins, your application can communicate with ”non-OIC” protocol devices using IoTivity APIs as shown in the following diagram.



<Figure 1. Protocol Plugin Concept>

2.1 OVERALL FLOWS

Using Plugin Manager APIs, application can start plugins that are located in a specific folder. After starting a plugin, the plugin will try to find its device using own protocol and creates resource server when the device is found. Then application can find and communicate with the resource using base APIs similar to normal IoTivity resource. Following diagram describes the flows.



<Figure 2. Overall Flow>

3 USING PLUGIN MANAGER

This section describes plugin configuration, manifest file of the plugin and the required folder structure for the plugins in the application.

3.1 SETTING PLUGIN CONFIGURATION

For plugin configuration, PluginManager.xml file should be located in the “lib” folder of the application. Then, plugin manager can load the config information when the application creates plugin manager instance.

```
<?xml version="1.0" encoding="utf-8"?>
<pluginManager>
  <pluginInfo
    PluginPath="PATH_TO_PLUGIN_FOLDER_OF_THE_APPLICATION”>
  </pluginInfo>
</pluginManager>
```

Example :

```
<?xml version="1.0" encoding="utf-8"?>
<pluginManager>
  <pluginInfo
    PluginPath=”/opt/usr/apps/org.iotivity.service.ppm.ppmsampleapp/lib/plugins”>
  </pluginInfo>
</pluginManager>
```

3.2 LOCATING PLUGIN AND MANIFEST FILE

Before starting the plugins, plugin binary should be located in the path specified in the plugin configuration and each plugin should be present in the path in a separate folder.

Below is an example for how to locate plugin (“mqtt-fan”) in a Tizen Application.

```
/sample-app/tizen/Application_Name/lib/plugins/mqtt-fan
    - fanserver_mqtt_plugin.so
    - plugin.xml  (Described below)
```

Following XML description is a plugin manifest file of “mqtt-fan” plugin.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin id="oic.plugin.mqtt-fan"
  version="0.1"
  name="mqtt-fan"
  url="fan"
  resourcetype="oic.fan">
<runtime library="fanserver_mqtt_plugin" funcs="mqtt_plugin_fanserver_funcs"/>
</plugin>
```

Each plugin has a manifest XML file in the same folder and will have following information.

Key Name	**Description**
id	Unique id of the plugin
version	Version of the plugin
name	Name of the plugin
url	URL of the plugin
resourcetype	Supported OIC resource type by the plugin
Runtime Library	.so file of the plugin

4 USING PLUGIN RESOURCES

This section describes how to communicate with non-oic devices using plugins and IoTivity API.

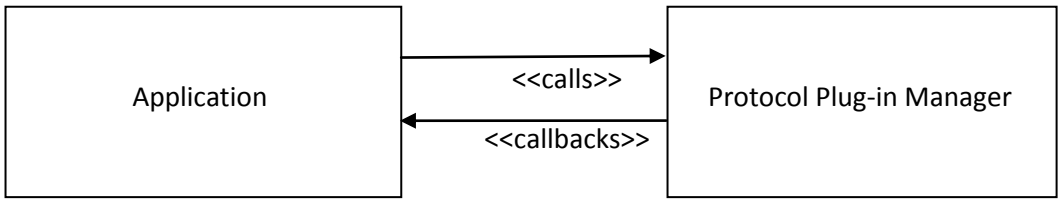
4.1 MQTT FAN PLUGIN

Application can find MQTT FAN device using “oic.fan” resource type and communicate with following attributes

Attribute Key	Attribute Value	Type	Description
power	“on”, “off”	String	Turn on/off the fan

5 SDK API

This section provides information on the APIs exposed by Protocol Plugin Manager service for the use by applications. SDK API is the facet of Protocol Plugin Manager to applications as shown in the Figure 1.



<Figure 3. Protocol Plug-in Manager SDK APIs and Application >

5.1 PROTOCOL PLUGIN MANAGER API

“Protocol Plugin Manager” APIs provide methods for application to start and stop the plugins, scan for plugins in the registered directory, get the list of plugins and also the state of plugins. The operations provided in the SDK are listed below:

- startPlugins
- stopPlugins
- rescanPlugin
- getPlugins
- getState

startPlugins API can be used to start the plugins by specifying key and value as parameters. Using the plugin information described in the manifest file, application can start plugins either by “resourceType” or by “id” as shown below:

startPlugins(“resourcetype”, “oic.fan”);

startPlugins(“id”, “oic.plugin.mqtt-fan”);

After starting, the plugin will try to find its device using its own protocol and will create a resource server when the device is found. Then the application can find and communicate with the resource using the base API as a normal IoTivity resource.

Prototype:

int startPlugins(const std::string key, const std::string value)

Parmaters:

- key - Key string of the plugin to be started.
- value - Value string of the plugin to be started.

Return Value:

- Returns 1 on Success, 0 on Failure.

stopPlugins API can be used to stop the plugins by specifying key and value as parameters. Key can be name of a resource type (Example: ResourceType) and value is the resource type value (Example: device.light). Once this API is called, the application can no longer find and communicate with the resource.

Prototype:

int stopPlugins(const std::string key, const std::string value);

Parmaters:

- key - Key string of the plugin to be stopped.
- value - Value string of the plugin to be stopped.

Return Value:

- Returns 1 on Success, 0 on Failure.

rescanPlugin API can be used to rescan for plugins in the registered directory and to install those plugins in the plugin manager table.

Prototype:

int rescanPlugin(void);

Return Value:

- Returns 1 on Success, 0 on Failure.

getPlugins API can be used to get the list of Plugins that are installed. An application can get the information of plugin as folllows

Prototype:

std::vector<Plugin> getPlugins(void);

Return Value:

- Returns available plugins' information in a vector.

getState API can be used to get the state of the plugin by providing plugin ID as parameter. This API returns the plugin state.

Prototype:

std::string getState(const std::string plugID);

Parmaters:

- plugID - ID of the plugin.

Return Value:

- Returns the state of the plugin as a String.

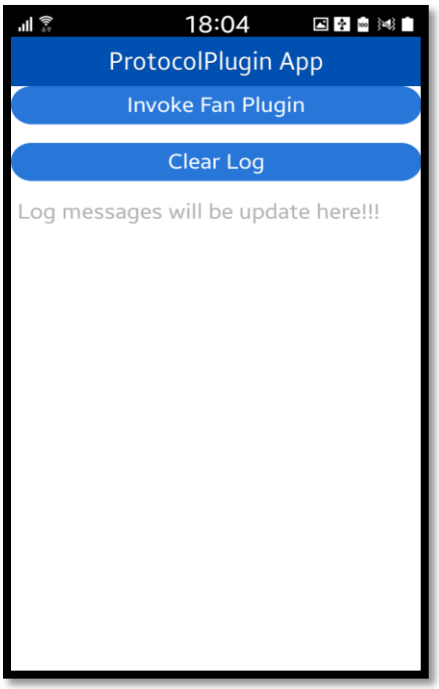
6 EXAMPLE

This section describes Tizen Sample Application for Protocol plugin manager.

6.1 TIZEN SAMPLE APPLICATION

This section describes flow of Sample Application in which we are trying to start, find and perform operation on the "mqtt-fan"plugin located in the plugins folder of the application (Complete folder structure of the application for the plugins is described in **section 4 : Using Plugin Manager**

Application Main Screen:



To perform operations on mqtt-fan plugin, Mosquitto server should be running on the same network.

Starting the Mosquitto server on port(1663) in linux machine:

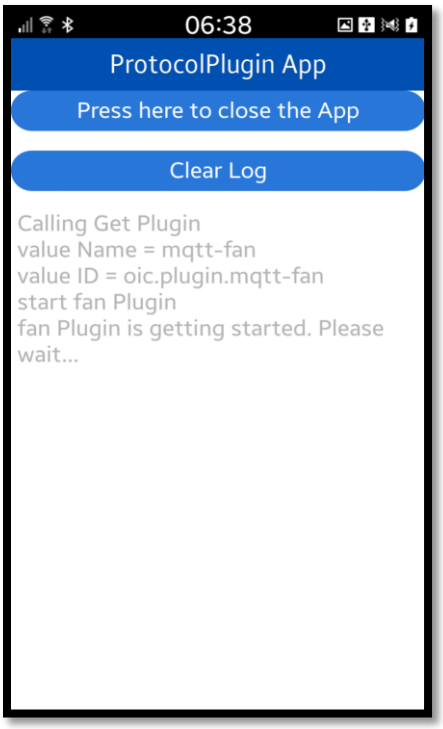
```
$ mosquitto -p 1663
1432720278: mosquitto version 0.15 (build date 2013-01-31 17:53:25+0000) starting
1432720278: Opening ipv4 listen socket on port 1663.
1432720278: Opening ipv6 listen socket on port 1663.
```

The port number of the mosquitto server and the IP address of the machine on which it is running should match with those specified in the service\protocol-plugin\plugins\mqtt-fan\src\fanserver.cpp file.

If User clicks on the “Invoke Fan Plugin” button it will create a instance of PluginManager class and will call getPlugins() API of Pluginmanager to get list of all plugins. In this sample it will return one plugin as there is only one plugin in the plugins folder of the application i.e. mqtt-fan.

After finding, sample application starts the plugin by calling startPlugin() API of PluginManager.

User can see the logs for these calls on the UI :



After starting the plugin, application finds the plugin using findResource API of OCPlatform to see whether mqtt-fan plugin is started or not. If its successfully started, then it will get the fan resource as shown in the UI. Once fan resource is found, application sends PUT request to change the power of the fan resource (1 - on , 0 - off) as shown in the UI.