|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Document History** | | | | |
| **Version** | **Date** | **Author** | **Section** | **Changes** |
| 1.0.1 | 30-07-2015 | Bas Flaton | All | Added permissions remark  Updated maven integration text  Added chapters 6, 7, 8 written by Jeroen Mols (to be merged with rest of doc) |
| 1.0 | 24-07-2015 | Deepthi Shivakumar | All | V1 features |

|  |  |
| --- | --- |
| **Author** | Deepthi Shivakumar |
| **Approved by** |  |
| **Email Id** | Deepthi.Shivakumar@philips.com |

DIComm Client Android Integration

1. INTRODUCTION 3

2. LIBRARY INTEGRATION 3

2.1 Maven repository Integration for internal Philips team 3

2.1.1 App build.gradle 3

2.1.2 Root build.gradle 3

2.1.3 Required permissions 4

2.2 Library Integration for external team 4

2.3 Library versioning 4

3. INITIALIZATION 4

3.1 Initialization steps 4

4. Application components overview 5

4.1 How to create an appliance and use it? 5

4.2 How to discover an appliance? 5

4.3 How to add a port ? 5

4.4 How to set communication strategy? 6

4.5 What is current appliance manager? 6

4.6 Key encryption and decryption 6

4.7 Local communication 6

4.8 KPSConfiguration info 7

4.9 Cpp or remote communication 7

4.10 Subscription 7

4.11 Push Notifications 8

4.12 App update 8

5. Notes 8

6. Initialization – high level description – will be merged with chapter 3 in future 9

7. Communication – high level description – will be merged with chapter 4 in future 10

8. Resource management – will be merged with chapter 4 in future 11

# INTRODUCTION

This library provides an overview of DIComm client library integration procedure for all android applications.

DIComm client is a horizontal component which allows mobile applications to communicate with connected appliances by implementing DICOMM protocol.

**Git Source Path:**

[**http://ingbtcpic2lx253.blr.pin.philips.com:8080/scm/git/hor-dicommclient-release-android**](http://ingbtcpic2lx253.blr.pin.philips.com:8080/scm/git/hor-dicommclient-release-android)

# LIBRARY INTEGRATION

## Maven repository Integration for internal Philips team

The easiest and preferred way to use these components is using maven.

**Note:** Maven test set up is used for v1 release.

Check out the code from above path where in you can find sample app which depends on dicomm client library’s aar file.

Please refer sample app and root build.gradle and settings.gradle

### App build.gradle

In the dependencies section of your app module build.gradle file, you will need to add the DiComm client lib compile dependency. This looks like (copied from sample app):

dependencies {  
 *//sample app will always use latest version of lib* compile **'com.philips.cdp:dicommClientLib:+'** *//you can also choose specific/older version like this  
 //compile 'com.philips.cdp:dicommClientLib:1.+'  
 //compile 'com.philips.cdp:dicommClientLib:1.0.0'*  
}

### Root build.gradle

In the repositories section of your main project build.gradle file, you will need to add an additional maven repository: the CDP artifactory repository. This look like (copied from sample app):

allprojects {  
 repositories {  
 jcenter()  
 maven { url **'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/libs-release-local'** }  
 }  
}

### Required permissions

Add the following permissions to your app:

* **android.permission.ACCESS\_NETWORK\_STATE**
* **android.permission.ACCESS\_WIFI\_STATE**
* **android.permission.INTERNET**
* **android.permission.CHANGE\_WIFI\_MULTICAST\_STATE**

## Library Integration for external team

Delivery to external teams is not yet supported in a formal way, but for now any such team will simple receive a zip file containing all release artifacts, including a .aar file of the lib, which they need to include in their project as you would do any .aar.

## Library versioning

Library version can be obtained by using below API

DICommClientWrapper.getDICommClientLibVersion();

# INITIALIZATION

DICommClientWrapper is the class which is used for library initialization.

We need to create discovery manager and cppcontroller instance as a first step of initialization. This class provides APIs to get discovery manager. Application need to create cppcontroller instance if it needs to connect/communicate remotely via CPP.

## Initialization steps

1. public static synchronized <U extends DICommAppliance> void initializeDICommLibrary(Context context,DICommApplianceFactory<U> applianceFactory, DICommApplianceDatabase<U> applianceDatabase, CppController cppController)

* DICommAppliance – base class for appliance, verticals need to extend this class to create their own representation of appliance.
* Context – application context.
* DICommApplianceFactory – an abstract class to identify the appliance. Apps need to identify the appliance by networkNode.getModelName() Ex: if model name is AirPurifier, Air app will return true.
* DICommApplianceDatabase – an abstract class which need to be extended if application likes to persist any data like cppid, modelname etc. **Please pass null if it is not required.**
* CppController – wrapper class to communicate to cpp. **Please pass null if it is not required.**
* NetworkNode – set of properties associated with an appliance.

1. public static synchronized void createSharedInstance(Context context, KpsConfigurationInfo kpsConfigurationInfo) {

KpsConfigurationInfo – an abstract class to be overridden to perform key provisioning

**If it is pass null, cppcontroller instance cannot be created.**

# Application components overview

DIComm defines a set of (abstract) classes that can be sub-classed in an application using DIComm library to override the existing behaviour.

## How to create an appliance and use it?

1. DICommAppliance is an abstract class that denotes an appliance. Vertical applications need to subclass it and can add extra properties.
2. Check the super class constructor and implement accordingly in child class.
3. If subscribe and unsubscribe methods are called on DICommAppliance, it is applicable for all the ports.
4. Also call addListenerForAllPorts and removeListenerForAllPorts.
5. Currently DIComm component searches all Philips appliances, it depends on application implementation to build required appliance for which application has to create subclass of DICommApplianceFactory and override below two methods.
6. canCreateApplianceForNode() where in application should check for modelname from network node like networkNode.getModelName() and if it matches with the appliance model name, then return true else return false.
7. createApplianceForNode() – create an appliance by passing in communication strategy and DiSecurity instance.(if appliance supports encrypted data)

DiSecurity – class which takes of encryption and decryption

## How to discover an appliance?

1. Implement DiscoveryEventListener interface to get notified about discover events. Call addDiscoveryEventListener() and removeDiscoverEventListener() appropriately.

2. Call start() method of DiscoveryManager class in onResume() of the activity.

3. Call stop() method of DiscoveryManager class in onPause() of the activity.

4. Once the appliances are discovered, onDiscoveredApplianceListChanged callback will be triggered and please use **getAllDiscoveredAppliances**() api to get the list of appliances.

5.  **Please do not use getAddedAppliances().**

## How to add a port ?

1. DIComm has a notion of port which is a set of properties grouped together.

2. DICommAppliance is a super class which contains all default ports like deviceport, firmwareport, wifiport etc. Create subclass of DICommAppliance and make sure you add appliance specific ports to it and expose methods to get those ports Ex: getAirPort(), getCookerPort(), etc.

3. addPort() – is the protected API used.

## How to set communication strategy?

We use strategy pattern to switch between local and remote communication and hence we have 4 strategies which are NullStrategy, LocalStrategy, RemoteStrategy and CommunicationMarshal.

Communication strategy object is created and passed in super call of DICommAppliance.

* LocalStrategy is used when it is sure that communication happens only locally.
* RemoteStrategy is used when it is sure that communication happens only remotely via CPP.
* CommunicationMarshal is used to switch between local and remote dynamically.
* NullStrategy is used when appliance is not in connected state.

DIComm users have to apply appropriate strategy when appliance is created and if communicationmarshal is used, dicomm component will take care of choosing appropriate strategy either local or remote depending on connection state of appliance.

## What is current appliance manager?

1. We have CurrentApplianceManager which is responsible for managing current appliance. Apis are exposed to set and remove current appliance, to add and remove appliance listener.

2. Use CurrentApplianceManager apis to add and remove appliance listener in onResume () and onPause () of activity respectively.

Please check if there are any specific requirements to have these listeners in few fragments.

## Key encryption and decryption

1. Key encryption and decryption is taken care by DIComm client library for local communication.
2. DISecurity is the wrapper class which performs these operations. If the instance of DISecurity is created while creating appliance, the requests will be encrypted and responses will be decrypted. Apps need not do anything else apart from creating DISecurity object.
3. If it is set as null then encryption and decryption is not performed.
4. ICPClient library is used internally for cpp communication which takes care of encryption and decryption during remote communication with CPP.

## Local communication

1. Local communication happens through HTTP request and depending on whether app needs security, request will encrypted.
2. App has to set right communication strategy and call APIs like below on each port,

putProperties() - for setting properties

getProperties() – for getting properties

subscribe() - to subscribe port events from appliance

unsubscribe() - to unsubscribe port events from appliance

stopResubscribe() - after 5 minutes it automatically resubsribes, if not interested please stop by calling this API.

1. Call addPortListener() and removePortListener() appropriately
2. Implement DICommPortListener to receive callbacks on port updates or if any error occurred on request.
3. Even subscription events are received through DICommPortListener callbacks.

## KPSConfiguration info

1. It requires setting of parameters like BootStrapId, BootStrapKey, product id, product version, component id, app id, app version, app type, country code, language code, device port url, component count and filterstring.
2. Please contact technical architect of the team to get above parameters.
3. Applications need to override KpsConfigurationInfo class and its object needs to be passed in while creating CppController.
4. Language code and country code which is set inside KpsCofigurationInfo should be dynamic and should always provide latest locale information.

This locale will be used inside cppcontroller frequently.

## Cpp or remote communication

1. App has to create shared instance of cppcontroller through the below method as mentioned above in the document.

public static synchronized void createSharedInstance(Context context, KpsConfigurationInfo kpsConfigurationInfo)

1. This is a singleton class and hence use getInstance() API to access the instance.
2. App need not bother about remote communication, app need to just assign right communication strategy and call the same APIs for local communication as mentioned above. DIComm will internally take care of routing requests remotely based on connection state.

## Subscription

1. If app sets current appliance to currentappliancemanager, internally dicomm will take care of enabling subscription for that appliance. Similarly it disables subscription when it is removed as current appliance.
2. Also apps can call these APIs to enable and disable ubsubcriptions on DICommAppliance.

enableSubscription() – created UDPThread if required and listens to subscription events on UDP port.

disableSUbscription() - stops UDPThread if there are no more listeners and also stops listening to UDP events.

## Push Notifications

1. App has to call sendNotificationRegistrationId(String gcmRegistrationId, String provider) API by passing registration id and provider name.
2. App has to implement SendNotificationRegistrationIdListener interface.
3. Call this API, setNotificationListener(SendNotificationRegistrationIdListener listener)

## App update

1. App has to implement AppUpdateListener interface and take actions on each callback events. Creating notifications UI is part of applications.
2. Please refer AirPurifier code.

# Notes

1. ICP client library is developed as separate library project. Whenever there is a library change, DIComm library is subjected to change.
2. Please refer sample application for more details.

# Initialization – high level description – will be merged with chapter 3 in future

Before the DICommClient can start its operation it needs to be configured with vertical specific information such as:

* Which appliances does the app support?
* What is the configuration to talk to the HSDP Device cloud (formerly CPP)?
* Where should the DICommClient store appliance specific information?
* ….

For this, the DICommClientWrapper offers an initialize method that allows you to pass in the following parameters:

1. Context – MANDATORY  
   Needs to be the Application context, because the lifespan of the DICommClient is the same as the Application object.
2. ApplianceFactory – MANDATORY  
   This allows to define which appliances the application will support by filtering based on the model name or type property of the discovered NetworkNode. The provided ApplianceFactory will typically return a vertical specific subclass of DICommAppliance.
3. ApplianceDatabase – OPTIONAL  
   This can be used to easily store additional information of a DICommAppliance.  
   When provided, the DiscoveryManager will automatically call the save/load methods at the right moments during device discovery.
4. CppController – OPTIONAL

To communicate to a DICommAppliance via the HSDP Device Cloud, the DICommClient needs to have a fully configured instance of the CppController. Note that the CppController configuration is app specific and that you will need to contact someone from the HSDP Device Cloud to get a correct configuration.

# Communication – high level description – will be merged with chapter 4 in future

Obviously, the DICommProtocol is used as a communication language between the app, appliance and backend. But before you can actually start communicating, you will need to tell the DICommClient how it actually has to communicate (local/remote/both) and what the vertical specific communication ports are. This can all be done in your custom subclass of DICommAppliance.

First of all choose which CommunicationStrategy to use:

* LocalStrategy: always does local communication over a Wi-Fi network. All communication is encrypted (AES, Diffie Hellman key exchange), so you must provide a DISecurity to initialize it.
* RemoteStrategy: always does communication via the HSDP device cloud backend. In order for this strategy to work, you must have provided a fully configured CppController while initializing the DICommClient.
* CommunicationMarshall: dynamically switches between Local or Remote strategy depending on how the DICommAppliance is connected. **This is probably the strategy you should use.**

Secondly specify which DICommPorts it supports:

* Every DICommAppliance will have the default DICommPorts out of the box such as the DevicePort, FirmwarePort, WifiPort,…
* Custom ports can be specified by creating a subclass of DICommPort, creating that instance in the constructor of the DICommAppliance and adding it to the list of ports of the appliciance.   
  **(note that this last step is incredibly important, because if a port is not added to the list, the DICommClient will not be able to properly manage all resources for that port)**

Finally you can start communicating to a DICommAppliance by:

* Getting a reference to a DICommAppliance
* Getting a reference to the DICommPort
* Performing one of the default DIComm actions on that port (setProperties, subscribe,…)

# Resource management – will be merged with chapter 4 in future

Discovering and communicating to appliances are quite resource intensive operations and hence the DICommClient offers quite some convenience methods to properly manage those resources.

* DiscoveryManager:
  + Can only be started/stopped from an Activity
  + Call start in **onResume and stop onPause** method.
  + Note that when the DiscoveryManager is stopped, it will remember the last know state for all appliances and hence it you will not need to start it in every activity.
* CurrentApplianceManager:
  + Can only be started/stopped from an Activity or Fragment
  + Call start in onResume and stop onPause method.
  + CurrentApplianceManager will automatically clean up all the resources for the old device whenever you change to a new one.