|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Document History** | | | | |
| **Version** | **Date** | **Author** | **Section** | **Changes** |
| 3.0 | 21-11-2017 | Peter Fortuin | All | Updated document for release of dicomm client lib 2017.5.0 |
| 2.0 | 14-10-2016 | Bas Flaton | All | Updated document for release of dicomm client lib 2.0.0 |
| 1.0 | 27-08-2015 | Deepthi Shivakumar | All | Addressed Jeroen’s remarks and merged the chapters |
| 0.2 | 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) |
| 0.1 | 24-07-2015 | Deepthi Shivakumar | All | V1 features |

DIComm Client Android Integration

|  |  |
| --- | --- |
| **Author** | Bas Flaton |
| **Approved by** |  |
| **Email Id** | bas.flaton1@philips.com |

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 settings.gradle 3

2.1.3 Root build.gradle 4

2.2 Library Integration for external team 4

2.3 Library versioning 4

3. INITIALIZATION 5

3.1 Initialization steps 5

4. Application components overview 5

4.1 How to create an appliance? 5

4.2 How to discover an appliance? 6

4.3 How to add a port ? 6

4.4 How to set communication strategy? 6

4.5 What is current appliance manager? 7

4.6 Key encryption and decryption 7

4.7 Local communication 7

4.8 KPSConfiguration info 8

4.9 Cpp or remote communication 8

4.10 Subscription 8

4.11 Push Notifications 9

4.12 App update 9

5. Notes 9

# INTRODUCTION

This library provides an overview of DIComm client library integration procedure for all android applications. DIComm client is a common component which allows mobile applications to communicate with connected appliances by implementing DICOMM protocol.

# LIBRARY INTEGRATION

## Library locations

The source code for the library can be found here:

ssh://tfsemea1.ta.philips.com:22/tfs/TPC\_Region24/CDP2/\_git/cml-android-commlib

**This includes the commlib-demouapp that can be used as starting point for implementation.**

The released artifacts can be found on artifactory:

<http://artifactory-ehv.ta.philips.com:8082/artifactory/webapp/#/artifacts/browse/tree/General/platform-pkgs-android-release/com/philips/cdp/commlib>

## Gradle integration

For successful integration you will need to:

* Add the correct maven repository to your root build.gradle file. The url can be found in the commlib-demouapp root build.gradle file.
* Add the dicomm client lib dependency to your module build.gradle (usually the ‘app’ module, but you can choose to put it in other modules of course). If you want to use the default implementation of the CloudController, also add a dependency to it.
* Depending on your app requirements you need to add the lan/wifi and/or ble dependency

compile **'com.philips.cdp:cloudcontroller:2017.5.0'**compile **'com.philips.cdp:commlib:2017.5.0'**

compile **'com.philips.cdp:commlib-lan:2017.5.0'**

and/or

compile **'com.philips.cdp:commlib-ble:2017.5.0'**

# Library usage

For detailed info on how to initialize and use the library, please consult the commlib-demouapp app that can be found in the git archive. Start by taking a look at the *DefaultCommlibUappDependencies* class. This class creates everything to initialize Commlib.

# Application components overview

DIComm creates necessary super classes and abstract classes where in an application which uses dicomm library can subclass them and override the existing behaviour.

## How to create an appliance?

1. *Appliance* is a super class which denotes an appliance. Vertical applications need to subclass 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 Appliance, it is applicable for all the ports.
4. Also call use addListenerForAllPorts and removeListenerForAllPorts.
5. Currently DIComm component searches all kind of Philips appliances, it depends on application implementation to build required appliance for which application has to create subclass of ApplianceFactory and override below two methods.
6. canCreateApplianceForNode() where in application should check for deviceType from network node like networkNode.getDeviceType() and if it matches with the appliance device type, 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 care of encryption and decryption

## How to discover an appliance?

1. Implement ApplianceListener interface to get notified about discover events. Call addApplianceListener() and removeApplianceListener() on the ApplianceManager (available in CommCentral) appropriately.

2. Call startDiscovery() method of CommCentral class in onResume() of the activity.

3. Call stopDiscovery() method of CommCentral class in onPause() of the activity.

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

## How to add a port ?

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

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

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 5 strategies which are NullCommunicationStrategy, LanCommunicationStrategy, CloudCommunicationStrategy, BleCommunicationStrategy and CombinedCommunicationStrategy.

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

* LanCommunicationStrategy is used for communication locally via (w)lan.
* CloudCommunicationStrategy is used for communication remotely via CPP.
* BleCommunicationStrategy is used for communication via BLE.
* CombinedCommunicationStrategy is used for appliances that can communicate via multiple channels.
* NullCommunicationStrategy is used when appliance is not in connected state.

DIComm users have to apply appropriate strategy when appliance is created and if CombinedCommunicationStrategy is used, dicomm component will take care of choosing appropriate strategy based on the order of the CommunicationStrategies he has.

## 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 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 or HTTPS request and depending on whether app needs security, request will be 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 resubscribes, 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.
4. If the device supports HTTPS communication, the device certificate will be stored at first use by computing the certificate’s pin and saving it on the NetworkNode that is created for the device.
5. Should the certificate on the device change at any time (for example due to a factory reset), the certificate pin on the device will no longer match the one stored in the NetworkNode instance created for this device. The mismatching incoming pin will be stored separately from the already stored pin, to allow for resolving the mismatch via the app using the library.
6. In the case of a request error because of a mismatch between the stored certificate pin and the incoming device certificate pin, the port listener will receive a callback with an Error type INSECURE\_CONNECTION. The app should at this moment handle this scenario by either accepting or rejecting the incoming certificate as the newly stored certificate.

This should be done using the applicable utility methods on the LanTransportContext class, being acceptNewPinFor(Appliance), rejectNewPinFor(Appliance) and findAppliancesWithMismatchedPinIn(Set<Appliance>).

## 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

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.

## 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)

Note: This is only to send your token to the device cloud. Sending your push token to the back end is only a small step in implementing push notifications. Implementing push notifications depends on the “provider” you choose.

# 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 commlib-demouapp for more details.