CommLib detailed design document

**Author :**

**Version :**

**Date :**

**Status : Draft**

TABLE OF CONTENTS

1.1 References 4

2. Appliance Discovery 4

2.1 CommCentral 4

2.2 Transport contexts 4

2.3 Discovery Strategies 4

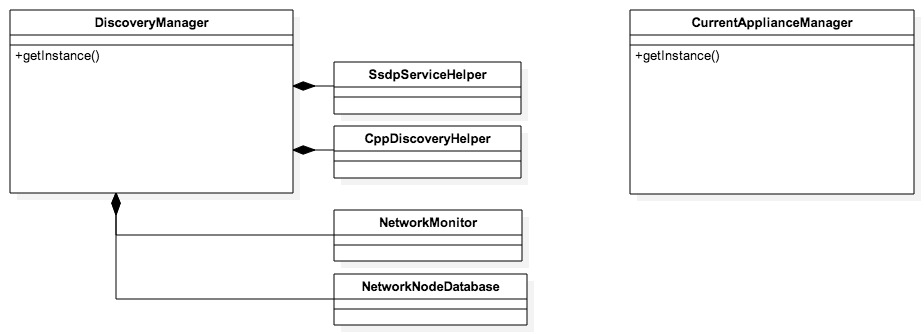
2.4 Appliance manager 5

2.5 Network Nodes 5

3. Communication 6

3.1 Ports 6

3.2 Strategies 6

 8

3.2.1 External interfaces 9

3.2.1.1 DiscoveryManager 9

3.2.1.2 CurrentApplianceManager 11

3.2.1.3 NetworkNode 12

3.2.1.4 DICommApplianceBuilder 13

3.2.1.5 DICommApplianceSettings 13

3.2.1.6 NetworkNodeDatabase 14

3.2.1.7 DiscoveryEventListener 14

3.2.2 Design-in 14

3.3 Local & remote control 15

3.3.1 External interfaces 15

3.3.1.1 DICommAppliance 15

3.3.1.2 DICommPort and its subclasses 16

3.3.1.3 DICommListPort and DICommListEntryPort 18

3.3.2 Internals: Connecting to a connected appliance 20

3.3.2.1 Local and Remote Control 21

3.3.3 Design-in 21

3.4 Discovery 22

3.4.1 External interfaces 22

3.4.2 Static design 22

3.5 Tests 22

3.6 overview 23

4. Roadmap 24

## References

[TODO] Document from CoCoCo team about BLE ref node (for discovery chapter)

# Appliance Discovery

To communicate with an appliance, you must first know of its existence. This is handled by a process called discovery. Classes that play a role in this process are depicted in Figure 1 and discussed in further detail in the paragraphs below.

## Class diagram

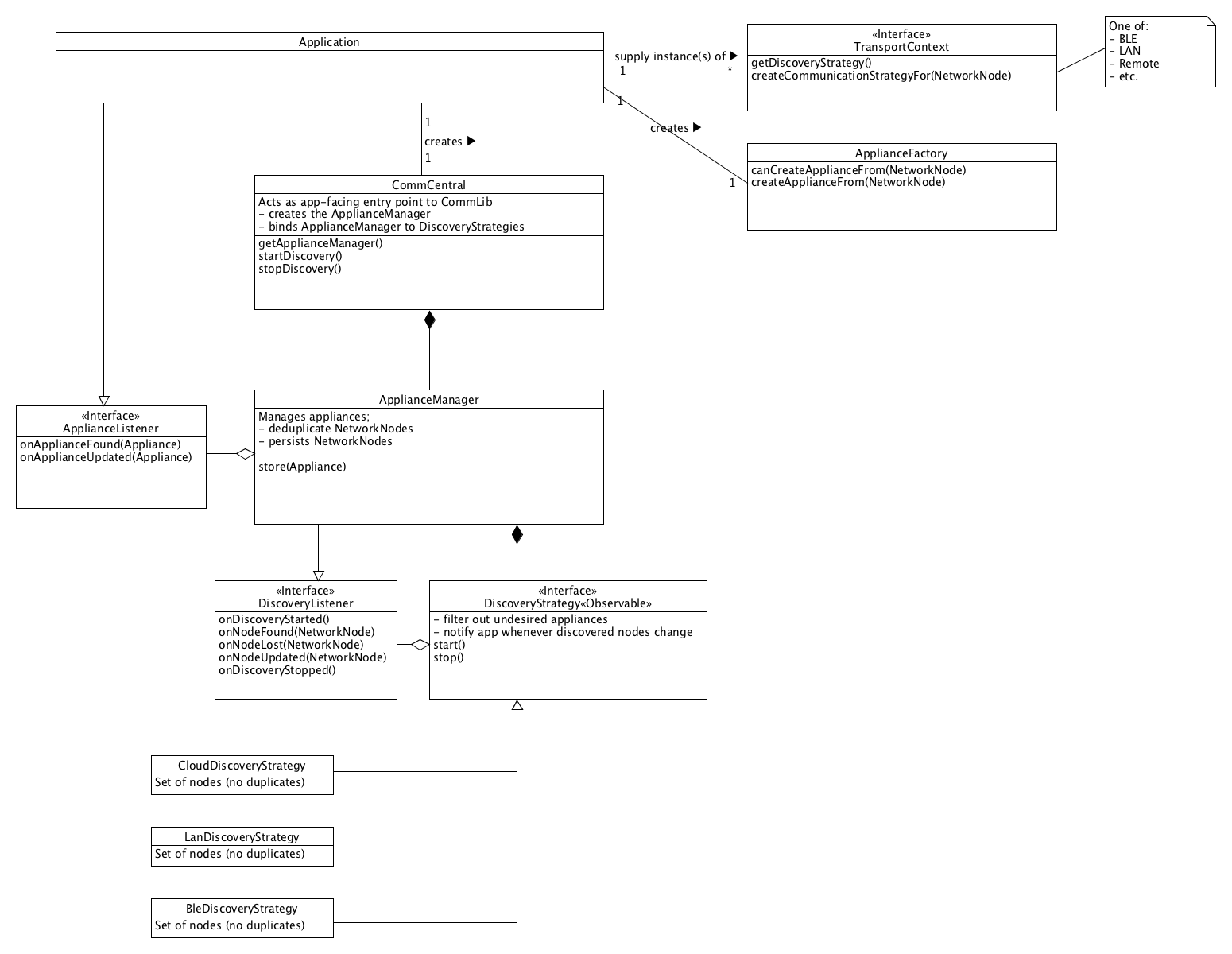


Figure : Class diagram for CommCentral design

## Application entry point

CommCentral is the entry point for an app using CommLib. This class is the glue between the ApplianceManager and the different TransportContexts that the app uses, for example BLE and WiFi. To setup the CommCentral class, the app must also provide an ApplianceFactory instance. Since an app is typically designed for the hardware that it will work with, it also must provide the implementation for the ApplianceFactory that CommLib will use to create Appliance instances.

## Transport contexts

Every transport layer (BLE, Wi-Fi, etc.) makes use of its own implementation of the TransportContext interface. Each holds the specific DiscoveryStrategy for the transport and acts as a factory for creating a CommunicationStrategy for a given NetworkNode.  
For example, a BleTransportContext will provide a BleDiscoveryStrategy implementation and a factory method for creating a BleCommunicationStrategy. The latter is a concrete implementation of CommunicationStrategy that can handle requests for the given NetworkNode.

The sum of these parts provides all means to interact with an appliance given a certain transport:

* Discovery
* Communication (request handling)

## Discovery Strategies

Different communication channels require different implementations of the discovery process. To hide the implementation details from the rest of the library, each discovery mechanism implements the same *DiscoveryStrategy* interface. This interface enforces the following contract:

* Discovery can be started and stopped.
* All *DiscoveryStrategies* should communicate their discoveries in the form of *NetworkNodes*, and nothing else.
* Observer pattern is used to update interested parties of changes in discovered *NetworkNodes*. Each node should be **found only once and updated thereafter**.
* **TODO: DEFINE FILTERING** Suggestion: Provide list of appliance model strings to *DiscoveryStrategy.start()* method. The discovery strategies should only report back discovered NetworkNodes that pass the filter.

Each strategy is responsible for monitoring availability of the communication channel it requires, like WiFi or Bluetooth, and should automatically resume its function when (1) it was previously started and (2) the communication channel becomes available. In other words: strategies are stateful.

*DiscoveryStrategies* can also mark previously discovered nodes as “lost”, for instance when it has not responded to poll requests for some time, or no keepalive messages were received, etc. This information is stored in the *NetworkNode* so that communication may be optimized in the future based on this.

## Appliance manager

* In the discovery process the *ApplianceManager* de-duplicates *NetworkNodes* that are found by more than one *DiscoveryStrategy*. It does this based on the *cppId* field of the node: if this id is the same, the information contained in the two nodes should be merged.
* If, during discovery, any node is not advertising new data anymore 120 seconds after it has done before, ApplianceManager marks the node as lost and notifies its listeners accordingly.
* If discovery is stopped explicitly, all NetworkNodes’ state should be frozen. This implies that the ‘node-lost-timeout’ should also be cancelled and/or removed.
* CommCentral registers ApplianceManager as a DiscoveryListener on all supplied DiscoveryStrategies.
* CommCentral starts discovery for all the registered DiscoveryStrategies.

ApplianceManager responsibilities:

1. Maintaining a list without duplicates for all the discovered nodes from all the difference discovery strategies.
2. Maintaining persistent storage.
3. Combining Discovered NetworkNodes with stored NetworkNodes (Removing duplicates) across all discovery strategies.
4. Creating the appliances nodes using ApplianceFactory.

## Network Nodes

The *NetworkNode* class holds all data needed for (1) identifying an appliance and (2) achieve communication with that same appliance using the various communication strategies (see chapter 3). This means that in the current setup the *NetworkNode* will need to be extended whenever a new communication channel is added to the library. This flaw has been accepted, since it is not expected to happen often.

To make clear what information is stored, consider the following example of *NetworkNode* data:

* Appliance cppId
* Appliance user friendly name
* Appliance model number
* BLE mac address
* BLE discovery status
* LAN ip address
* LAN discovery status
* …

# Communication

## Ports

## Strategies

## 

Figure : Classes involved in lifecycle management and discovery. Both DiscoveryManager and CurrentApplianceManager are part of the external interface.

In order to discover and control DICommAppliances, the DI Comm Android architecture offers two components: DiscoveryManager and CurrentApplianceManager. Since they both require quite some resources to do their work, they should be managed properly within the application lifecycle.

First of all, the DiscoveryManager is responsible for keeping an up to date list with all DICommAppliances that can be controlled by the application. For this, it uses SSDP to discover appliances on the local Wi-Fi network, the Philips CPP back-end to know which appliances can be connected to via the cloud, and a NetworkMonitor to listen to connectivity changes of the mobile device itself. To release all resources associated with the DiscoveryManager the start() and stop() method must be called in the onResume() and onPause() method respectively of every Activity requiring the discovery to run.

Secondly, the CurrentApplianceManager introduces the concept of one single DICommAppliance that can be controlled by the application at any given point in time. It will automatically subscribe to all necessary DI Comm ports of that device and will modify the subscription whenever the ConnectionState of the appliance changes. To release all of the associated resources, all DICommApplianceEventListeners must be registered in the onResume() method and unregistered in the onPause() method. When there are no listeners interested in getting subscription updates, the CurrentApplianceManager will automatically stop its subscription.

### External interfaces

#### DiscoveryManager

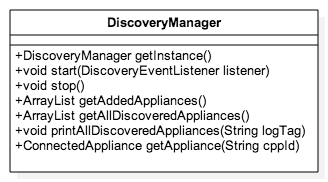


Figure : Public API of DiscoveryManager

The DiscoveryManager is responsible for managing a list of all DICommAppliances and to keep the ConnectionState of each of these up to date. This ConnectionState is an enumeration consisting of only three different states as shown by figure 4.

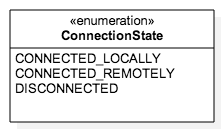


Figure : Possible connection states of an DICommAppliance object

In an effort to minimise cost, the DiscoveryManager will always attempt to connect to the DICommAppliance on the current Wi-Fi network, before attempting to connect to it via CPP. Further it uses quite some external information to make the discovery process as fast as possible. The most notable are:

* Changes in connectivity: Wi-Fi switches, mobile data or no connection.
* SSDP to discover devices on a Wi-Fi network
* Request CPP for online DICommAppliances and subscribe to changes
* Last known DICommAppliance information from database
* …

Note that the list of DICommAppliances does not only contain all online devices, but also all other DICommAppliances that the user added to his list of known devices. The latter list of known devices is persisted in a database and restored on every start of the application.

The DiscoveryManager is a singleton class, which can be accessed in the following way:

public static synchronized DiscoveryManager getInstance();

This will simply return the singleton instance of the DiscoveryManager, but will not yet start the discovery mechanism. This is because the discovery process is quite resource intensive (background Threads, BroadcastReceivers, DCS connection…) and hence the application should have fine-grained control over these resources throughout its lifecycle.

In order to know which appliances to build, the DiscoveryManager needs to have a DICommApplianceBuilder, which can be achieved by calling the createInstanceWithBuilder() method. This method can only be called when no shared instance was created yet, so calling this more than once will fail. Passing NULL for any of the parameters will also cause the method to fail.

+ (DICommApplianceManager\*) createSharedInstanceWithBuilder:(DICommApplianceBuilder\*)builder settings:(DICommApplianceSettings\*)settings

This method should be called in the onCreate() of the Application object so it is always ready to be used.

In order to start the discovery process, each Activity requiring up to date Discovery information must call the start() method in its onResume() method.

public void start(DiscoveryEventListener listener);

The DiscoveryListener parameter will get called on every change to the list of trusted DICommAppliances and whenever something changes to a DICommAppliance in that list. Consequently the onDiscoveredDevicesListChanged() callback should be used as a trigger to update the UI.

Note that the start() method only takes one single DiscoverEventListener as a parameter and that there is no way to add extra DiscoveryEventListeners. This is not a limitation, but instead an intentional architecture choice to only allow Activities to call the Start() method on the DiscoveryManager. And because there can only be one visible Activity, this restriction makes sense.

Similarly there is a stop() method which must be called in the onPause() method by each Activity that started the discovery process.

public void stop();

This will stop the discovery process, remove the DiscoveryEventListener and clean up all resources. Note that the list of trusted discovered appliances can still be accessed after calling stop(), but the information in the list will be out of date.

Calling start() in onResume() and stop() in onPause() avoids having resource leaks and makes discovery robust. Subsequent calls to start(), stop(), start() (for example going from one Activity to the other), are buffered out and have no effect.

Note that only Activities that require up to date discovery information should actually start and stop the DiscoveryManager.

The rest of the interface is straightforward: getAllDiscoveredAppliances() returns the most up to date list of all DICommAppliances. The list can be empty, but never null.

public ArrayList<DICommAppliance> getAllDiscoveredAppliances();

Mostly however, apps should not use the list of all DICommAppliances, but instead only use the DICommAppliances that were explicitly added by the user. This list is a subset of the list returned by getAddedAppliances() and should be used to display DICommAppliances in the application.

public List<DICommAppliance> getAddedAppliances();

There is also a convenience method for looking up a DICommAppliance based on its cppId.

public DICommAppliance getDICommAppliance(String cppId);

Finally for debugging purposes it is often useful to log the entire list of all DICommAppliances so it can be easily inspected. There is also a convenience method to make this possible.

public void printAllDiscoveredAppliances(String logTag);

#### CurrentApplianceManager

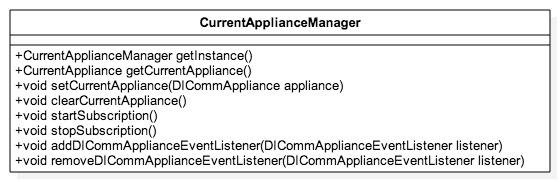


Figure : Public API of CurrentApplianceManager

The Android DIComm architecture has the concept of a current DICommAppliance, which is the only appliance that can be controlled at any given point in time. The main advantage of this design decision is that DIComm users don’t need to explicitly (un) subscribe to DICommAppliance events, simplifying the interaction.

In DIComm Android the CurrentApplianceManager is responsible for managing the current DICommAppliance and to ensure that the user is always subscribed to that appliance. This means it will automatically clean up all subscription resources and start a new subscription when the current DICommAppliance changes. Further it will also respond to changes in the ConnectionState of the current appliance and adjust the current subscription accordingly.

To start interaction with the CurrentApplianceManager, the singleton instance of it can be obtained via the getInstance() method.

public static synchronized CurrentApplianceManager getInstance();

After the DiscoveryManagers discovers its first appliances, the current DICommAppliance should be set to the CurrentApplianceManager by calling the setCurrentAppliance() method. This method will automatically start all necessary subscriptions for the new DICommAppliance.

public synchronized void setCurrentAppliance(DICommAppliance appliance);

Note that if there was already a current DICommAppliance set, this method will first stop all subscriptions for that appliance, before starting the new ones. Further this method will also register an Observer for the current DICommAppliance and toggle the subscription based on changes in its ConnectionState.

When set, the current DICommAppliance can be accessed by using the getCurrentAppliance() method or removed via the removeCurrentAppliance() method.

public synchronized DICommAppliance getCurrentAppliance();

public synchronized void clearCurrentAppliance();

Similarly to discovery, receiving events is a quite resource intensive operation and must hence be disabled when the application goes to the background. For this CurrentApplianceManager offers a mechanism to start and stop the subscription. These methods should be called in onResume() and onPause() respectively.

public void startSubscription();

public void stopSubscription();

To get notified of new DICommPort subscription events, a DICommApplianceEventListener needs to be subscribed to the CurrentApplianceManager. This can be done using the addDICommApplianceEventListener() and removeDICommApplianceEventListener() methods.

public void addDICommApplianceEventListener(  
 DICommApplianceEventListener DICommApplianceEventListener);

public void removeDICommApplianceEventListener(  
 DICommApplianceEventListener DICommApplianceEventListener);

There is no limitation on the amount of listeners that can be registered in this way.

#### NetworkNode

This class stores information about network-related aspects of connected appliances. It is a data class has the following properties.

|  |  |  |
| --- | --- | --- |
| DINetworkNode (iOS) / com.philips.cl.di.comm.NetworkNode (Android) | | |
| name | String | The name that the user gave to the appliance |
| modelName | String | The model name of the appliance |
| modelType | String | The model type of the appliance |
| connectionState | enum | DISCONNECTED, CONNECTED\_LOCALLY or CONNECTED\_REMOTELY |
| ipAddress | String | The IP address on which the appliance can be reached |
| homeSsid | String | The network name of the network in which the appliance is configured |
| cppId | String | The identifier by which the appliance is known at the Philips CPP back-end |
| bootId | Integer | A number that can be used to detect that the appliance has rebooted |
| allowRemoteControl | Boolean | Indicates whether the appliance may be controlled remotely. This property is not persistent. |
| encryptionKey | String | The encryption key used for communication within the home Wi-Fi network |
| isOnlineViaCpp | Boolean | Indicator if Philips CPP back-end is able to connect to the appliance or not |
| pairedState | enum | PAIRED, NOT\_PAIRED, UNPAIRED, PAIRING |
| lastPairingTime | Long | Last time when the status of the pairing relation was checked |

#### DICommApplianceBuilder

This builder is used to create instance of DICommAppliance. Most projects will have a special-purpose subclass of DICommAppliance. In that case you also need a special-purpose subclass of DICommApplianceBuilder that can construct instances of the DICommAppliance subclass.

public DICommAppliance buildDICommAppliance(NetworkNode node);

This method constructs an instance of ConnectedApplication and configures it with the given node.

public boolean canBuildDICommAppliance(NetworkNode node);

Discovery may return multiple types of Philips connected appliances. Specific projects usually only support one (or a few) of those. The builder should implement this method to indicate whether it supports the given node (usually by checking the node’s model name). If true, buildDICommAppliance() may be called. If false, that method will return null.

Note that the SSDPService currently only searches for appliances with

#### DICommApplianceSettings

This class stores information that needs to persist between launches of the app. It has properties that store the list of CPP IDs of the appliances that the user has added. Also, it stores the CPP ID of the one that the user has currently selected in the user interface:

private List<String> addedDICommAppliancesCppIds;

private String selectedDICommApplianceCppId;

Besides these properties, there is also a property that controls whether or not remote control is supported by the app. This controls whether the CurrentApplianceManager will sign into CPP on startup. You can find more information about this in section 2.2.2.1.

private boolean remoteControlSupported;

All of these properties have appropriate getter and setter methods in order to access and mutate their values. Every change in properties is immediately persisted in the shared preferences of the android device.

Finally, note that the new preferences can be added to the DICommApplianceSettings by creating a subclass and injecting that into the CurrentApplianceManager and DiscoveryManager.

#### NetworkNodeDatabase

// TODO describe a generalized mechanism to store DICommAppliance information between consecutive app launches.

#### DiscoveryEventListener

The DiscoveryEventListener gets notified of changes to the list of managed devices. This will probably be mainly used to dynamically update the UI. The onAllAppliancesListChanged() method will get called whenever an appliance is added or removed or when the ConnectionState of an appliance changes.

public void onAllAppliancesListChanged();

This callback is flexible enough to support even Easy Wi-Fi Setup, as developers can simply loop over all appliances to find one with a matching boot id.

### Design-in

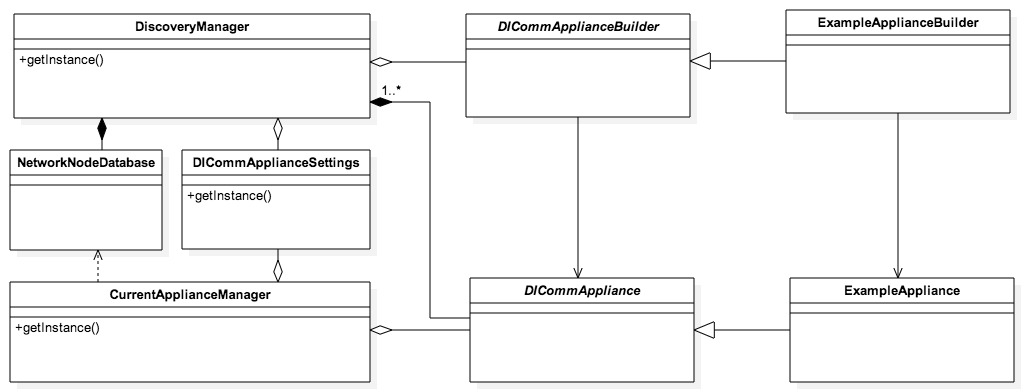


Figure : Example of design-in on the lifecycle management layer. The Example classes are vertical-specific. The other classes are re-used as-is.

To design-in these classes, you need to subclass DICommApplianceBuilder to build the class for your vertical specific DICommAppliance subclass (see 2.2).

## Local & remote control

The component enables local and remote control of the connected appliance as explained in and [DIPRA]. The component automatically selects between local and remote control by detecting whether it is on a local network where it found the connected appliance. In other cases it defaults to remote control. It is possible to configure the component to not use remote control.

### External interfaces

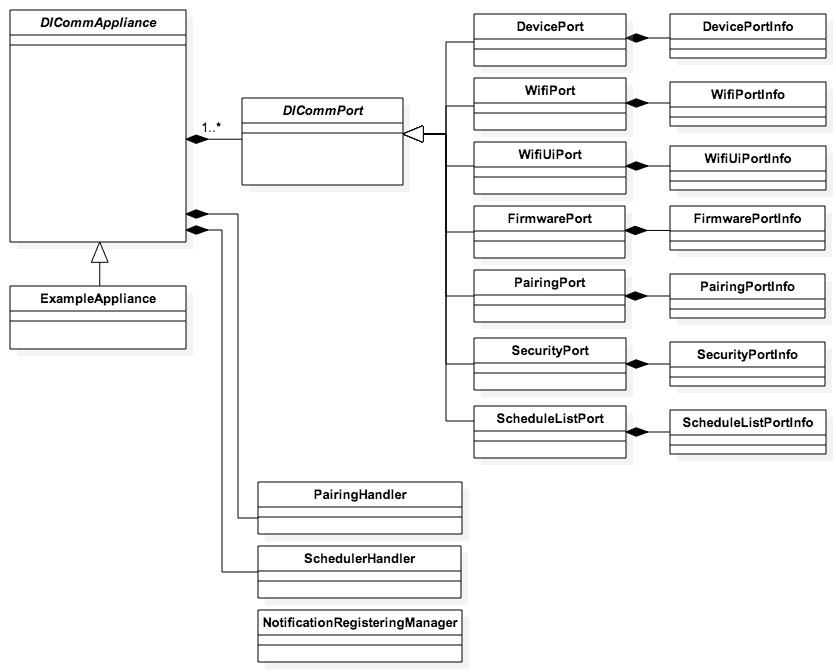


Figure 7: The classes involved in the external interface of local & remote control

The external interface consists of the DICommAppliance class and its ports, which are all instances of the DICommPort subclass. These ports mirror all relevant ports for mobile apps that are available in the specification. Further there are three helper classes available to facilitate with schedules, pairing and push notifications respectively.

TODO: PairingHandler, SchedulerHandler and NotificationRegisteringManager should be better blended into the architecture and not directly exposed to the outside of DI Comm.

#### DICommAppliance

The most import properties are the following. They provide access to the various ports of the DICommAppliance. These ports provide the user of the component with the ability to read and write data to the ports of the connected appliance.

private final DevicePort devicePort;

private final WifiPort wifiPort;

private final WifiUiPort wifiUiPort;

private final FirmwarePort firmwarePort;

private final PairingPort pairingPort;

private final SecurityPort securityPort;

private final ScheduleListPort scheduleListPort;

The DiscoveryManager will set the networkNode property. It contains the information that the DICommAppliance needs to reach the appliance.

private final NetworkNode networkNode;

These methods deal with subscriptions on all ports that support subscription. See explanation about subscription for the DICommPort class.

public void subscribe();

public void unsubscribe();

To enable the use of subscriptions, some transport layers need to keep a connection open. This is done by calling enableCommunication() on an Appliance. On the ble transport layer this will keep the connection to the Appliance open between calls which has the added benefit of allowing faster communication. It is advised to do this before attempting to transfer big amounts of data.

// TODO description of special ports (pairing and notifications)

#### DICommPort and its subclasses

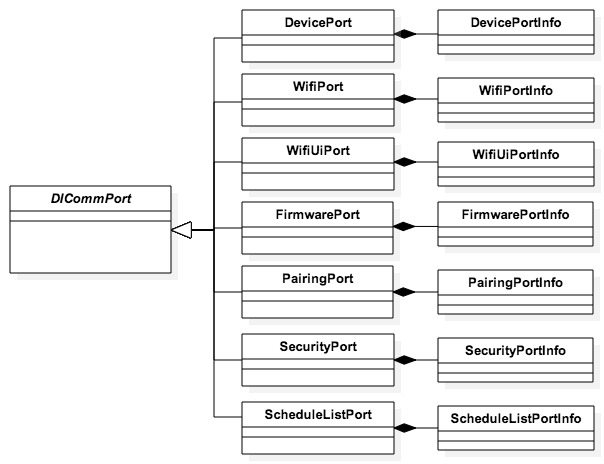


Figure 8: DICommPort and its subclasses. Functionality of DICommPort will never be called directly but only through its subclasses. When designing-in a vertical specific DICommPort subclass needs to be created.

DICommPort is an abstract class and can never be instantiated directly. All access to its code will go via its subclasses. DICommPort provides all the basic mechanisms for:

1. Connecting to a connected appliance via the protocol.
2. Maintaining subscriptions
3. Handling the UI/UX performance flow explained in [DIPRA].
4. Caching the data received from the connected appliance.
5. Helper functions for converting the received JSON data to its format.

The port-specific subclasses deal with making the parsed JSON data accessible to the rest of the app and as such they form the external interface. They will not be discussed in detail here, as they are mostly straightforward adaptations from .

// TODO: Describe PairingPort and PushNotificationPort, which handle the entire flow for enabling pairing and push notifications, including connections to CPP.

##### Reading data

There are two ways to get information from a DICommPort subclass. The first method is a “pull”-method: you can get information directly through the properties of the DICommPort subclass. When you try to read a property for which the port does not have a value in the cache this will automatically trigger a reload of the port over the network. This is called “GETPROPS” in terms of . You can force such a reload by calling the getProperties method.

public <V> getPortInfo()

The second method is a “push”-method by registering an instance of SubscriptionEventHandler. This will be called when the port’s properties get updated.

##### Subscriptions

Some ports allow subscriptions to updates on the device. The caller of the API can have the port subscribe to updates by calling the subscribe method on the DICommPort subclass or on DIDICommAppliance. In the latter case DIDICommAppliance will subscribe to updates on all ports that support subscription. The updates received through subscription will trigger calls to the registered DIPropertyUpdateHandlers.

As specified in “SUBSCRIBE” requests will time-out after a certain time. DICommPort will automatically try to refresh the subscription.

##### Sending Data

Sending data is achieved by calling putProperties() on the DICommPort subclasses. This will trigger a “PUTPROPS” request in terminology, which will be sent as soon as possible after. If it failed the requested change will not be reflected on the \*\*\*PortInfo members and the ResponseHandler will be notified. As long as DICommPort is still trying to apply changes the isApplyingChanges() method returns true. The isApplyingChanges() method can be used to display a spinner or other activity indicator to the user.

##### Performance considerations

As seen in it is important that the performance of the connection is perceived as being high. DICommPort implements all the functionality necessary for the mechanism listed in in a way that is mostly invisible to the caller of the API. Performance of sending data is also covered in 2.2.1.2.3.

An important performance consideration is that it is not allowed to send parallel requests to a DI Comm connected appliance. DICommPort makes sure that doesn’t happen so that the callers of the API do not have to worry about this, but this does have a slight impact on the timing on the external interface.

DICommPort tries to minimize the amount of network calls needed. When multiple changes are requested while DICommPort is already performing a request for example, it will combine them in one call. Similarly, if getProperties is called but the current state of the connected appliance’s port is received as part of a “SUBSCRIBE” or “PUTPROPS” request it will treat the reloadProperties as successful without sending a “GETPROPS” request. If the same methods (with the same parameters) are called multiple times, DICommPort will not necessarily perform the associated network calls several times.

The request types are prioritized in this order (high priority to low):

1. Changes (“PUTPROPS”) are prioritized highest because the user usually initiates changes. The time it takes for changes to reflect in reality will be the primary measurement for performance.
2. Subscribe calls are second because they also return the current state of the port, obsoleting possible reloadProperties calls.
3. Unsubscribe calls are third because the user will not care about their performance.
4. ReloadProperties (“GETPROPS”) are prioritized last because the other calls will also return the current state, making the “GETPROPS” network call superfluous.

Note that the DI Comm Android architecture is not yet fully compliant with the UI/UX performance specifications from . As such it doesn’t ignore incoming property updates in case of an ongoing “PUTPROPS” request and it still updates the UI in between two immediately consecutive “PUTPROPS” requests. Finally a “SUBSCRIBE” request is send every time the current appliance changes, regardless of whether there was already an ongoing subscription. These issues need to be addressed in further document updates.

#### DICommListPort and DICommListEntryPort

To accommodate for ports consisting of multiple other ports, the DI Comm Android architecture provides a DICommListPort class. This type of port consists of list of sub ports, which are all an instance of the same subclass of DICommListEntryPort. The exact subclass DICommListPort should use can be specified by defining the generic type <V> while subclassing DICommListPort and by instantiating the correct class in its factory method.

public class DICommListPort<T extends DICommListEntryPort> extends DICommPort <Object> { … }

public abstract <T> createNewListEntryPort(String listPortName, String portKey)

DICommListEntryPort contains a Map consisting of a variable amount of DICommListEntryPort classes; each identified by a unique identifier. (Note that this identifier is a property of DICommListEntryPort itself as will be described further.)

private Map<String, T> listEntryPorts;

To access the DICommListEntryPort classes, it offers the following methods:

public int getNumberOfListEntryPorts();

public <T> getListEntryPort(String identifier);

Creating a new DICommListEntryPort must be done via a two-step approach:

1. Create a new port on the actual appliance and wait for it to be created.

public void addNewListEntryPort(AddListEntryPortListener addListener);

This method will return asynchronously whenever the new port is created.

1. Edit the properties of that port to the correct values using putProperties()

To remove a DICommListEntryPort, the following method is offered.

public void removeListEntryPort(String identifier);

Note that DICommListPort differs slightly from other DICommPorts in the sense that its properties are actually ports. This means that a getProperties() operation will return a list of DICommListEntryPort instead of a list of properties. Whenever the DICommListPort receives a properties update, it will either create a new DICommListEntryPort internally or update an existing DICommListEntryPort when the identifiers match.

Next, the DICommListEntryPort is a simple subclass of DICommPort. This class offers a way to configure the name of its DICommListPort class and also adds unique identifier. Both of these parameters have to be defined in its constructor:

public DICommListEntryPort(String parentPortName, String identifier)

Not only does this ensure the link between the DICommListPort and DICommListEntryPort, but it also allows the port name to be automatically generated by appending the parent port name and the unique identifier. Consequently the abstract getName() method of DICommPort no longer has to be overridden by subclasses.

The DICommListEntryPort also offers a method to access its identifier.

public String getIdentifier();

Note that DICommListEntryPort is similar to a DICommPort in every other way, which means you can do all DIComm commands on this port.

### Internals: Connecting to a connected appliance

This section explains the internals of the local and remote control functionality. In principle this should be totally re-usable, so this is for background information only. Because this is not part of the external interface it will be explained in lesser detail.

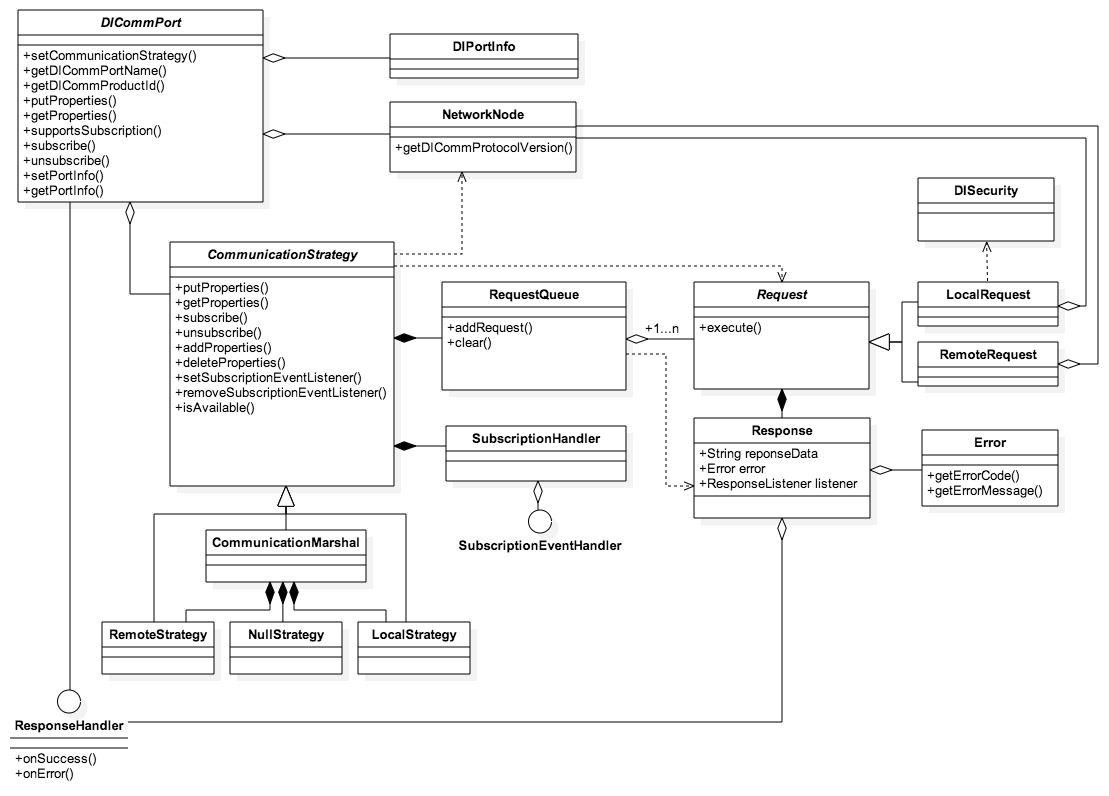


Figure : The class diagram for the strategies.

DICommPort handles all connections via subclasses of CommunicationStrategy. This follows both the Strategy design and the Composite design pattern. CommunicationStrategy will never be used directly. These strategy subclasses exist:

1. LocalStrategy: connects over local Wi-Fi to the connected appliance. It can only be used whenever the DICommAppliance’s node has a local ConnectionState.
2. RemoteStrategy: connects through CPP to the connected appliance. It can be used whenever the DICommAppliance’s node has a local or remote ConnectionState. Due to cost considerations however it is only advised to use it for the latter case.
3. NullStrategy handles the unconnected case. It gracefully fails all the requests. It should only be used for the not connected ConnectionState.
4. Communication marshal: A strategy that dynamically selects between the other strategies depending on the ConnectionState. Further it will prioritize the LocalStrategy over the RemoteStrategy.

Almost all ports share the strategy instances and they all use the same communication marshal strategy instance. This instance contains a local, remote and a null strategy instance.

#### Local and Remote Control

The DI Comm Client supports both local and remote connections. For local connections, the app and the appliance need to be on the same network. Remote connections don’t have that requirement, as they work through the Philips CPP back-end.

For some products, supporting remote control is a requirement, whereas for some other products it should not be supported.

// TODO describe mechanism to prevent CPPController from signing in.

### Design-in

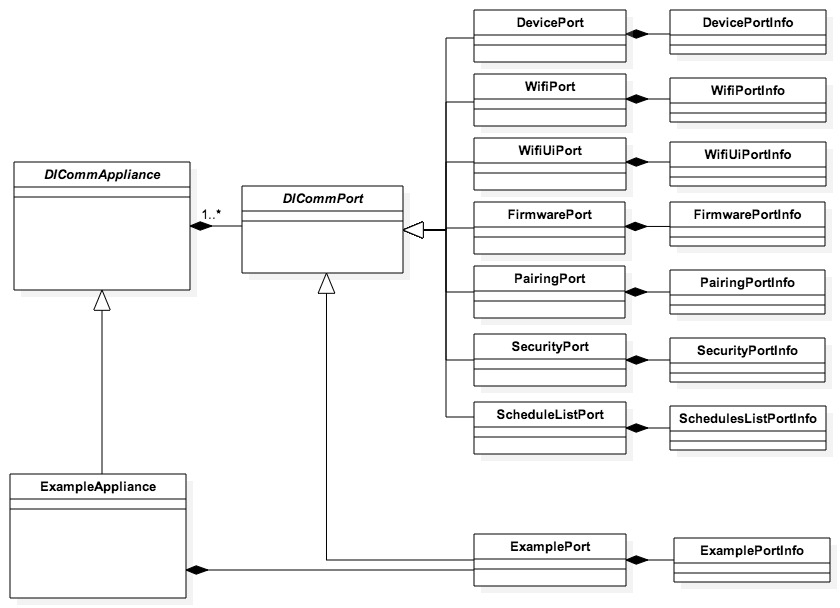


Figure : Example of design-in on local & remote control. The Example classes are vertical-specific. All other classes are re-used as-is.

To design-in these classes, you need to subclass DICommAppliance to add your vertical specific functionality. It will probably need to add a vertical-specific subclass of DICommPort to its list of ports. The DICommPort subclass needs to implement methods to translate the values available from the DICommPort methods to semantic values the remainder of the app can use and vice versa. These need to correspond with the vertical specific specification of the DICOMM protocol.

As an example, the Smart Air app contains the subclass Airpurifier that adds an AirPort instance to the list of ports. AirPort translates the parsed JSON values to integers, enums and strings and makes them available as an AirPortInfo object. When setting these integers, enums and strings the AirPort calls the appropriate setProperty methods on the DICommPort superclass that initiates network calls towards the connected appliance.

## Discovery

The component allows you to discover DI Comm connected appliances on your network as defined in section 3.1.3 of .

### External interfaces

The external interface to discovery is DiscoveryManager, which is described in 2.1.1.1.

### Static design

DiscoveryManager uses the SSDP discovery protocol and for DI Comm implementations on the currently connected Wi-Fi network. The class performs discovery every 15 seconds.

## Tests

All existing unit test must be maintained and run regularly to ensure no regression occurs. While

All functionality contains extensive unit and integration tests. It is important to maintain and run those, so that all the verticals making use of the horizontal component remain compatible with the component.

## overview

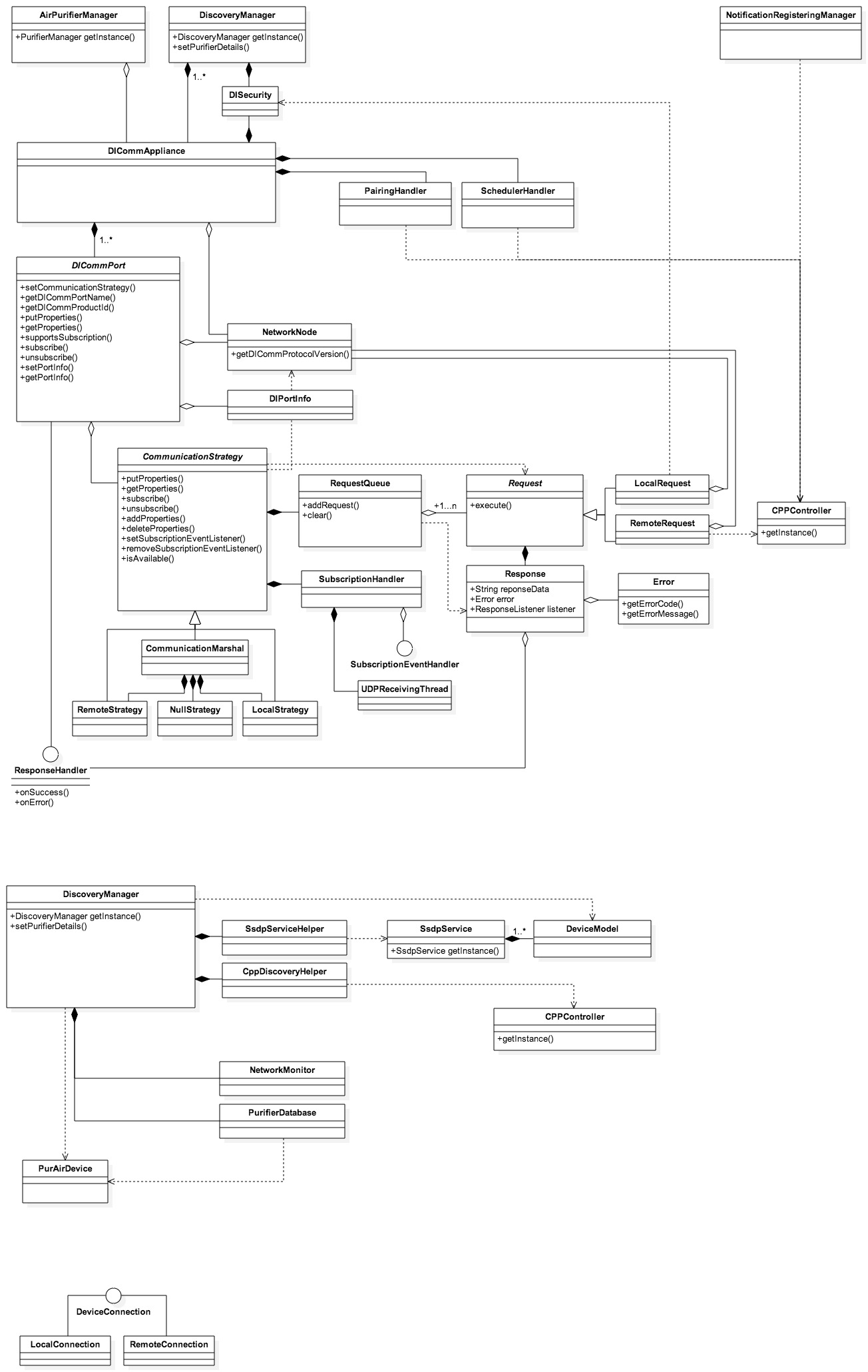


Figure : Overview of entire DI Comm Android architecture

# Roadmap

1. Basic reuse from SmartAir Android

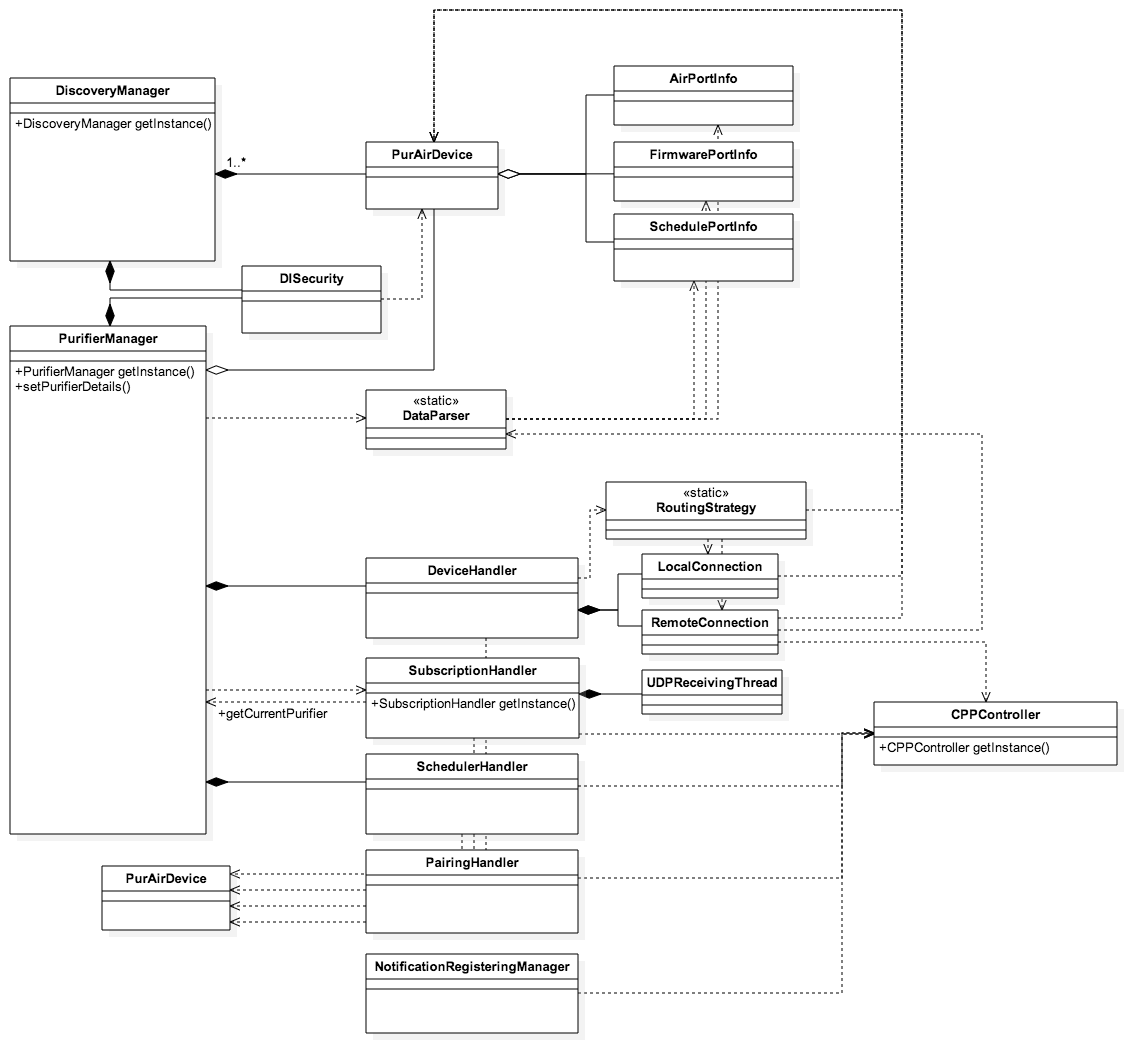


Figure : Current architecture of Smart Air Android. Significant changes are necessary to make this architecture compliant with the reference design.

When developing the existing architecture of Smart Air Android, reuse was not an explicit goal and hence significant changes are necessary to make it reusable. This is mainly because the architecture does not explicitly use DICommPort classes and because all PurAirDevice interaction is routed through the PurifierManager.

This appendix aims at being a comprehensive step-by-step guide for making the architecture reusable. The user of this guide should follow all steps sequentially and after every (sub)chapter, all unit tests must be run, basic testing needs to be performed and the code must be checked in. Fundamental changes will be highlighted and their risks will be pointed out so proper testing can be performed.

* 1. Introduce NetworkNode object

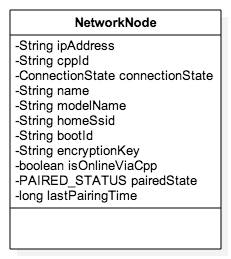


Figure : NetworkNode object for DiComm Android architecture

Introduce a new class NetworkNode as a member of PurAirDevice and move the following fields to that class: mPurifierIp, mEui64, mConnectionState, mName, mLastKnownSsid, mBootId, mEncryptionKey, pairedState, mLastPairedTime, isOnlineViaCpp. Make sure getter and setter methods are generated for all these members in NetworkNode and that all interaction to these members are passed through the PurAirDevice.

Rename the members in NetworkNode and their corresponding getters and setters to make them consistent with the reference design:

* mPurifierIp -> ipAddress
* mEui64 -> cppId
* mConnectionState -> connectionState
* mName -> name
* mLastKnownSsid -> homeSsid
* mBootId -> bootId
* mEncryptionKey -> encryptionKey
* mLastPairedTime -> lastPairedTime

Add a new member modelName to NetworkNode and add getters and setters for it. Finally inline all interactions to the NetworkNode class via the PurAirDevice and make sure that the following classes are no longer dependent on the PurAirDevice class (but on the NetworkNode instead):

* RoutingStrategy
* LocalConnection
* RemoteConnection
* DISecurity
* JSONBuilder
* DeviceHandler
* PairingHandler
* ScheduleHandler

After, in the SubscriptionHandler class, change the method arguments for subscribe() and unSubscribe() to use a NetworkNode instead of a PurAirDevice.

Finally, the NetworkNode should be transferable from the Easy Wi-Fi Setup component to the reset of the application. In order to make this possible, have the NetworkNode implement the Parcelable interface and add a Creator method to the class.

* 1. SubscriptionHandler can no longer be singleton
     1. Remove dependency of CppDiscoveryHelper on SubscriptionHandler

Modify the CPPController so it can redirect discovery events directly to the CppDiscovery helper by adding the following to CPPController:

* Add a new member CppDiscoveryListener and a setter for this member
* Move the following code from SubscriptionHandler.onDCSEventReceived() directly into the notifyDCSListener() method:

if (DataParser.parseDiscoverInfo(data) != null) {  
 ALog.i(ALog.SUBSCRIPTION, "Discovery event received - " + action);  
 boolean isResponseToRequest = false;

if (action != null && action.toUpperCase().trim().equals(AppConstants.DISCOVER)) {  
 isResponseToRequest = true;  
 }

if (cppDiscoverEventListener != null) {  
 cppDiscoverEventListener.onDiscoverEventReceived(data,isResponseToRequest);  
 }  
 return;  
}

Next, modify the CppDiscoveryHelper to directly talk to the CPPController:

* Set the CppDiscoveryListener directly to CPPController in the constructor
* Enable/disable remote subscription directly via the CPPController  
  (inline mSubHandler.enableRemoteSubscription() and mSubHandler.disableRemoteSubscription())
* Remove mSubHandler member
* Remove injection of SubscriptionHandler from Constructor

Finally note that enabling and disabling the remote subscription should be made smarter (see Appendix B).

* + 1. Prepare SubscriptionHandler to be instantiated more than once.

Every SubscriptionHandler will register itself to the CPPController and hence the CPPController needs to be prepared to handle multiple DCSEventlisteners:

* Add a new member HashMap<String, DCSEventListener> dcsEventListeners
* Add a new method addDCSEventListener(String cppId, DCSEventListener dcsEventListener)
* Add new method getDCSEventListener(String cppId)

Change the SubscriptionHandler so it takes the NetworkNode as an argument and registers itself to the CPPController when the PurifierManager tries to get access to the singleton instance.

* Change the signature of getInstance() to getInstance(NetworkNode networkNode)
* Register the instance to the CPPController every time the getInstance() method is called by adding the following line before the return statement:

public SubscriptionHandler getInstance(NetworkNode networkNode) {  
 …  
 CppController.getInstance(PurAirApplication.getAppContext)  
 .addDcsEventListener(networkNode.getCppId(), mInstance);  
 return mInstance;  
}

* change the PurifierManager to insert the relevant networkNode in every invocation of Subscriptionhandler.getInstance(NetworkNode networkNode)

Prepare the CppController to handle multiple instances of the SubscriptionHandler:

* use the correct DCSEventListener according to the returned cppId in notifyDCSListener() and use that one instead of the member dcsEventListener
* remove the method setDCSEventListener(DCSEventListener dcsEventListener) and its reference from the SubscriptionHandler
* remove the member dcsEventListener
  + 1. Remove dependency on PurifierManager

The SubscriptionHandler currently relies on the PurifierManager to get the current PurAirDevice whenever a subscription is started and stopped. This dependency can be removed by:

* Adding a private member: private NetworkNode networkNode
* Adding a setter for that member setNetworkNode(NetworkNode networkNode)
* Storing the NetworkNode element in the getInstance() method

public SubscriptionHandler getInstance(NetworkNode networkNode) {  
 …  
 mInstance.setNetworkNode(networkNode);  
 return mInstance;  
}

* Use the new NetworkNode member instead of all references to PurifierManager.getInstance().getCurrentPurifier().getNetworkNode()

Remark: Setting the NetworkNode explicitly might seem a bit off, but it will be removed Later on, when the SubscriptionHandler becomes part of the PurAirDevice (see A.3.4) .

* 1. Move the SubscriptionHandler into PurAirDevice.
     1. Inject external dependencies into SubscriptionHandler

Modify the SubscriptionHandler, so all its external dependencies are injected in the constructor.

* Change constructor signature to take a NetworkNode as an argument and assign that to the networkNode member.
* Change constructor signature to take a SubscriptionEventListener as an argument and assign that to the subscriptionEventListener member.
* Register the current instance as a DCSEventListener to the CPPController in the constructor.
* The entire constructor should now look like this:

public SubscriptionHandler(NetworkNode networkNode, SubscriptionEventListener   
 SubscriptionEventListener) {  
 this.networkNode = networkNode;  
 this.subscriptionEventListener = subscriptionEventListener;  
 CppController.getInstance(PurAirApplication.getAppContext).addDcsEventListener(  
 NetworkNode.getCppId(), this);  
}

* + 1. Add SubscriptionHandler to PurAirDevice

Next, the PurAirDevice needs to process all events received via DCS or UDP so this responsibility can be removed from the PurifierManager. To do this, make the following changes to the PurAirDevice:

* Implement the SubscriptionEventListener interface and reroute all calls to the PurifierManager. For example:

public void onLocalEventReceived(String encryptedData, String purifierIp) {  
 PurifierManager.getInstance().onLocalEventReceived(encryptedData, purifierIp);  
}

* Add a new member SubscriptionHandler subscriptionHandler
* Initialize the member at the end of the existing constructor

Public PurAirDevice() {  
 …  
 subscriptionHandler = new SubscriptionHandler(networkNode.getCppId(), this);  
}

* Add a temporary getter method for the member: getSubscriptionHandler()
  + 1. Refactor PurifierManager to use SubscriptionHandler from PurAirDevice

Now the PurifierManager needs to start using the SubscriptionHandler from the PurAirDevice instead of its own. In the PurifierManager:

* Change all references from SubscriptionHandler.getInstance() to getCurrentPurifier().getSubscriptionHandler();
* Don’t register a listener by removing the following line from the constructor: SubscriptionHandler.getInstance(…).setSubscriptionListener(this);
* Move the following items from the PurifierManager to the PurAirDevice:
  + Constant RESUBSCRIBING\_TIME
  + Members handler and subscribeRunnable (and access them via getters)
  + Method subscribeToAllEvents(PurAirDevice purifier)   
    (in PurifierManager : getCurrentPurifier.subscribeToAllEvents())
  + Method unSubscribeFromAllEvents(PurAirDevice purifier)  
    (in PurifierManager : getCurrentPurifier.unSubscribeToAllEvents())
    1. Clean up remainders from singleton

Finally the following clean up can be performed:

* PurAirDevice
  + Inline and remove getters to handler and subscribeRunnable
  + Remove the getSubscriptionHandler() method
* SubscriptionHandler:
  + Remove the methods setSubscriptionListener() and setNetworkNode()
  + Remove the getInstance(), setDummySubscriptionManagerForTesting() method and mInstance member
  + Simplify all unit tests to directly use the SubscriptionManager instead of injecting it via the previous method.

Note that the architecture will now have multiple SubscriptionHandlers instead of one. This is a significant change and has to be properly tested to remove all side effects.

* 1. Move the DeviceHandler into PurAirDevice

Prepare the PurAirDevice so it holds its own DeviceHandler:

* Add a member private DeviceHandler deviceHandler
* Instantiate it in the constructor using the PurAirDevice instance as listener
* Add a getter for the member getDeviceHandler()

Change the PurifierManager so it starts using the DeviceHandler from the PurAirDevice:

* replace all mDeviceHandler references to getCurrentPurifier().getDeviceHandler()
* remove the member mDeviceHandler and its instantiation in constructor
* Move setPurifierDetails() method into the PurAirDevice

(Note that NotificationsFragment and PurifierControlPanel should call: PurifierManager.getInstance().getCurrentPurifier().setPurifierDetails())

Finally, perform the following clean up in the PurAirDevice:

* inline the getDeviceHandler() method everywhere
* remove the getDeviceHandler()method

Note that this change causes one queue to exist per purifier instead of one queue for all purifiers. This is a significant change and has to be properly tested to remove all side effects.

* 1. Move the SchedulerHandler into PurAirDevice

Prepare the PurAirDevice so it holds its own SchedulerHandler:

* Add a member private SchedulerHandler schedulerHandler
* Instantiate it in the constructor using the PurAirDevice instance as listener
* Add a getter for the member getSchedulerHandler()

Change the PurifierManager so it starts using the DeviceHandler from the PurAirDevice:

* replace all schedulerHandler references to getCurrentPurifier().getSchedulerHandler()
* remove the member schedulerHandler and its instantiation in constructor
* Move sendScheduleDetails() method into the PurAirDevice

(Note that SchedulerActivity should call: PurifierManager.getInstance()  
.getCurrentPurifier().sendScheduleDetailsToPurifier())

Finally, perform the following clean up in the PurAirDevice:

* inline the getSchedulerHandler() method everywhere
* remove the getSchedulerHandler()method
  1. Create a new class DICommPort
* must be abstract: public abstract class DICommPort
* define the abstract method:
  + public void isResponseForThisPort(String response)
  + public void processResponse(String response)
  1. Create a new class AirPort

The AirPort class should extend the DICommPort class and use empty implementations for its abstract methods isResponseForThisPort() and processResponse(). Further do the following:

* Add a member private DeviceHandler deviceHandler
* Add a constructor argument DeviceHandler and assign it to the member

Next, the PurAirDevice class should start using an AirPort object to store all AirPortInfo and to access all AirPort functions. In the PurAirDevice class, this can be accomplished by:

* Adding a member private AirPort airPort and a getter getAirPort()
* Initializing the member in the constructor, using the deviceHandler from the PurAirDevice instance
* Moving the airportInfo member into the airPort member and updating getAirPortInfo() and setAirPortInfo() to return/modify the object from the AirPort
* Inline the methods getAirPortInfo() and setAirPortInfo() in all classes, so the AirPortInfo is accessed as follows:

PurifierManager.getCurrentPurifer().getAirPort().getAirPortInfo();

* Moving the setPurifierDetails() method from PurifierManager into AirPort  
  (Note that NotificationsFragment and PurifierControlPanel should call: PurifierManager.getInstance().getCurrentPurifier().getAirPort().setPurifierDetails())

Then the AirPort class needs be modified so it can process some actual logic:

* Add a method parseResponse():

private AirPortInfo parseResponse(String reponse) {  
 return DataParser.parseAirPurifierEventData();  
}

* Implement the method isResponseForThisPort():

public boolean isResponseForPort(String response) {  
 return (parseResponse(response) != null);  
}

* Implement the method processResponse():

public void processResponse(String response){  
 AirPortInfo info = parseResponse(response);  
 if (info != null) {  
 setAirPortInfo(info);  
 return;  
 }  
 // Log something here  
}

Finally the PurifierManager needs to be modified so the event handling can be moved into the PurAirDevice later:

* Make the following changes in notifySubscriptionListeners():
  + Introduce a new parameter PurAirDevice purifier at the top of the method and initialize it by calling getCurrentPurifier()
  + add a null check to exit the method when this parameter is null.
  + Replace DataParser.parseAirPurifierEventData() and the if check for AirPortInfo not null with the following code:

if(purifier.getAirPort().isResponseForThisPort()) {  
 purifier.getAirPort().processResponse();  
}

* Make the following changes in notifyAirPurifierEventListeners():
  + Remove the call to setAirPortInfo()
  + Remove the method setAirPortInfo() from the PurifierManager
  + Remove the AirPortInfo object from the method signature
  1. Create a new class FirmwarePort

The FirmwarePort class should extend the DICommPort class and use empty implementations for its abstract methods isResponseForThisPort() and processResponse().

The PurAirDevice class should start using a FirmwarePort object to store all FirmwarePortInfo and to access all FirmwarePort functions. In the PurAirDevice class, this can be accomplished by:

* Adding a member private FirmwarePort firmwarePort and a getter getFirmwarePort()
* Initializing the member in the constructor
* Moving the firmwarePortInfo member into the firmwarePort member and updating getFirmwarePortInfo() and setFirmwarePortInfo() to return/modify the object from the FirmwarePort
* Inline the methods getFirmwarePortInfo() and setFirmwarePortInfo() in all classes, so the FirmwarePortInfo is accessed as follows:

PurifierManager.getCurrentPurifer().getFirmwarePort().getFirmwarePortInfo();

Then the FirmwarePort class needs be modified so it can process some actual logic:

* Add a method parseResponse():

private FirmwarePortInfo parseResponse(String reponse) {  
 return DataParser. parseFirmwareEventData();  
}

* Implement the method isResponseForThisPort():

public boolean isResponseForPort(String response) {  
 return (parseResponse(response) != null);  
}

* Implement the method processResponse():

public void processResponse(String response){  
 FirmwarePortInfo info = parseResponse(response);  
 if (info != null) {  
 setFirmwarePortInfo(info);  
 return;  
 }  
 // Log something here  
}

After, the PurifierManager needs to be modified so the event handling can be moved into the PurAirDevice later:

* Make the following changes in notifySubscriptionListeners():
  + Replace DataParser.parseFirmwareEventData() and the if check for FirmwarePortInfo not null with the following code:

if(purifier.getFirmwarePort().isResponseForThisPort()) {  
 purifier.getFirmwarePort().processResponse();  
}

* Make the following changes in notifyFirmwareEventListeners():
  + Remove the call to setFirmwarePortInfo()
  + Remove the method setFirmwarePortInfo() from the PurifierManager
  + Remove the FirmwarePortInfo object from the method signature

Finally, inline the DataParser.parseFirmwareEventData() method in FirmwarePort and remove it from DataParser entirely.

* 1. Create a new class ScheduleListPort

The ScheduleListPort class should extend the DICommPort class and use empty implementations for its abstract methods isResponseForThisPort() and processResponse(). Further do the following:

* Add a member private SchedulesHandler schedulesHandler
* Add a constructor argument SchedulesHandler and assign it to the member

Next, the PurAirDevice class should start using a ScheduleListPort object to store all ScheduleListPortInfo and to access all ScheduleListPort functions. In the PurAirDevice class, this can be accomplished by:

* Adding a member private ScheduleListPort ScheduleListPort and a getter getScheduleListPort()
* Initializing the member in the constructor, using the schedulesHandler from the PurAirDevice instance
* Rename the following members/methods:
  + mSchedulerPortInfoList to ScheduleListPortInfoList
  + getmSchedulerPortInfoList() to getScheduleListPortInfoList()
  + setmSchedulerPortInfoList() to setScheduleListPortInfoList()
* Moving the ScheduleListPortInfoList member into the ScheduleListPort member and updating getScheduleListPortInfoList() and setScheduleListPortInfoList() to return/modify the list from the ScheduleListPort
* Inline the methods getScheduleListPortInfoList() and setScheduleListPortInfoList() in all classes, so the ScheduleListPortInfoList is accessed as follows:

PurifierManager.getCurrentPurifer().getScheduleListPort().getScheduleListPortInfoList();

* Moving sendScheduleDetailsToPurifier() method from PurifierManager into ScheduleListPort  
  (Note that SchedulerActivity should call: PurifierManager.getInstance()  
  .getCurrentPurifier().getScheduleListPort().sendScheduleDetailsToPurifier())

Then the ScheduleListPort class needs be modified so it can process some actual logic:

* Add a method parseResponseAsSingleSchedule():

public SchedulePortInfo parseResponseAsSingleSchedule(String reponse) {  
 // TODO make method private after refactoring SchedulerActivity  
 return DataParser.parseScheduleDetails();  
}

* Add a method parseResponseAsScheduleList ():

public List<SchedulePortInfo> parseResponseAsScheduleList(String reponse) {  
 // TODO make method private after refactoring SchedulerActivity  
 return DataParser.parseSchedulerDto();  
}

* Implement the method isResponseForThisPort():

public boolean isResponseForPort(String response) {  
 if (parseResponseAsSingleSchedule(response) != null) return true;  
 if (parseResponseAsScheduleList(response) != null) return true;  
 return false;  
}

* Implement the method processResponse():

public void processResponse(String response){  
 throw new RuntimeException("Method not implemented – Refactor   
 SchedulerActivity")  
}

Then the PurifierManager needs to be modified so the event handling can be moved into the PurAirDevice later:

* Make the following changes in notifySubscriptionListeners():
  + Replace DataParser.parseAirPurifierEventData() and the if check for AirPortInfo not null with the following code:

if (purifier.getScheduleListPort().isResponseForThisPort()) {  
 SchedulePortInfo ScheduleListPortInfo =   
 purifier.getScheduleListPort().parseResponseAsSingleSchedule(data);  
 if (ScheduleListPortInfo != null) {  
 notifyScheduleListenerScheduleReceived(schedulePortInfo);  
 return;  
 }

List<SchedulePortInfo> ScheduleListPortInfoList =   
 purifier.getScheduleListPort().parseResponseAsScheduleList(data);  
 if (ScheduleListPortInfoList != null) {  
 notifyScheduleListenerScheduleListReceived(ScheduleListPortInfoList);  
 return ;  
 }  
}

// TODO determine better way to detect schedule out of memory error if (data.contains(AppConstants.OUT\_OF\_MEMORY)) {  
 notifyScheduleListenerErrorOccured(  
 SchedulerHandler.MAX\_SCHEDULES\_REACHED);  
 return;  
}

* + Add the following methods:

private void notifyScheduleListenerScheduleReceived(SchedulePortInfo   
 schedulePortInfo) {  
 if (scheduleListener == null) return;  
 scheduleListener.onScheduleReceived(schedulePortInfo);  
 }

private void notifyScheduleListenerScheduleListReceived(  
 List<SchedulePortInfo> ScheduleListPortInfoList) {  
 if (scheduleListener == null) return;  
 scheduleListener.onSchedulesReceived(ScheduleListPortInfoList);  
}

private void notifyScheduleListenerErrorOccured(int error) {  
 if (scheduleListener == null) return;  
 scheduleListener.onErrorOccurred(error);  
 }

* + Refactor onLocalEventLost() to call notifyScheduleListenerErrorOccurred() in the SCHEDULER case.

Finally, the following clean ups can be performed in the ScheduleListPort class:

* Inline the DataParser.parseScheduleDetails() and remove the method from the DataParser class entirely
* Inline the DataParser.parseScheduleDto() and remove the method from the DataParser class entirely
  1. Move Encryption into the PurAirDevice

In order to make encryption and decryption completely transparent, DISecurity should become a part of the PurAirDevice .

First of all, prepare the PurAirDevice by:

* Implement the interface KeyDecryptListener, leave the keyDecrypt() method empty
* Add a new member private DISecurity diSecurity and getter getSecurity()
* Initialize the member in the constructor, using the PurAirDevice instance as the interface
* Move the keyDecrypt() method from PurifierManager to the PurAirDevice.
* Clean up/simplify the keyDecrypt() method by:
  + Remove purifier parameter and replace with calls to the current object
  + Change log tag to purifier

Next, the PurifierManager can be simplified:

* Replace mSecurity with purifier.getSecurity()in onLocalEventReceived()
* Remove the member mSecurity

Note that in the future DISecurity should be further simplified to only accommodate for one single PurAirDevice and that its usage in other classes should also be refactored to handle that change (see Appendix B).

* 1. PurAirDevice must handle received events

The PurAirDevice should handle all events directly so that the PurifierManager doesn’t have to know about this logic anymore. In order to do this, make the following changes:

First of all, the onLocalEventReceived() method should be moved from PurifierManager to PurAirDevice.

* Make notifySubscriptionListeners() method public in PurifierManager
* Inline onLocalEventReceived() call through in the onLocalEventReceived() method of the PurAirDevice and remove the method from PurifierManager
* Clean up onLocalEventReceived() in PurAirDevice by removing the purifier variable

Second, the onRemoteEventReceived() method should be moved from PurifierManager to PurAirDevice.

* Inline onRemoteEventReceived() call through in the onRemoteEventReceived() method of the PurAirDevice and remove the method from PurifierManager
* Clean up onRemoteEventReceived() method in PurAirDevice by removing purifier variable

Third, the onLocalEventLost() method should be moved from PurifierManager to PurAirDevice.

* Extract a method for the AQI\_THRESHOLD case in PurifierManager:

public void notifyAirPurifierEventListenersErrorOccured(PurifierEvent purifierEvent) {  
 synchronized (airPurifierEventListeners) {  
 for (AirPurifierEventListener listener : airPurifierEventListeners) { listener.onErrorOccurred(purifierEvent);  
 }  
 }  
}

* Inline onLocalEventLost() call through in the onRemoteEventReceived() method of the PurAirDevice and remove the method from PurifierManager

Finally, the PurifierManager can be cleaned up a little bit:

* Remove the SubscriptionEventListener interface (all methods were removed in the previous steps)
* Move the notifySubscriptionListeners() method to PurAirDevice and convert all calls to notify subscription listeners to their corresponding call on the PurifierManager. For instance:

getPurifierManager().notifyAirPurifierEventListeners();

* 1. Finalize architecture

Rename the following classes:

* PurAirDevice to AirPurifier
* PurifierManager to AirPurifierManager

After all of the proposed changes above, the Android architecture will now look like the one in Figure 14. Clearly a lot of responsibility has been shifted from the PurifierManager to the AirPurifier object and the existence of DICommPorts makes this architecture extensible for future products.

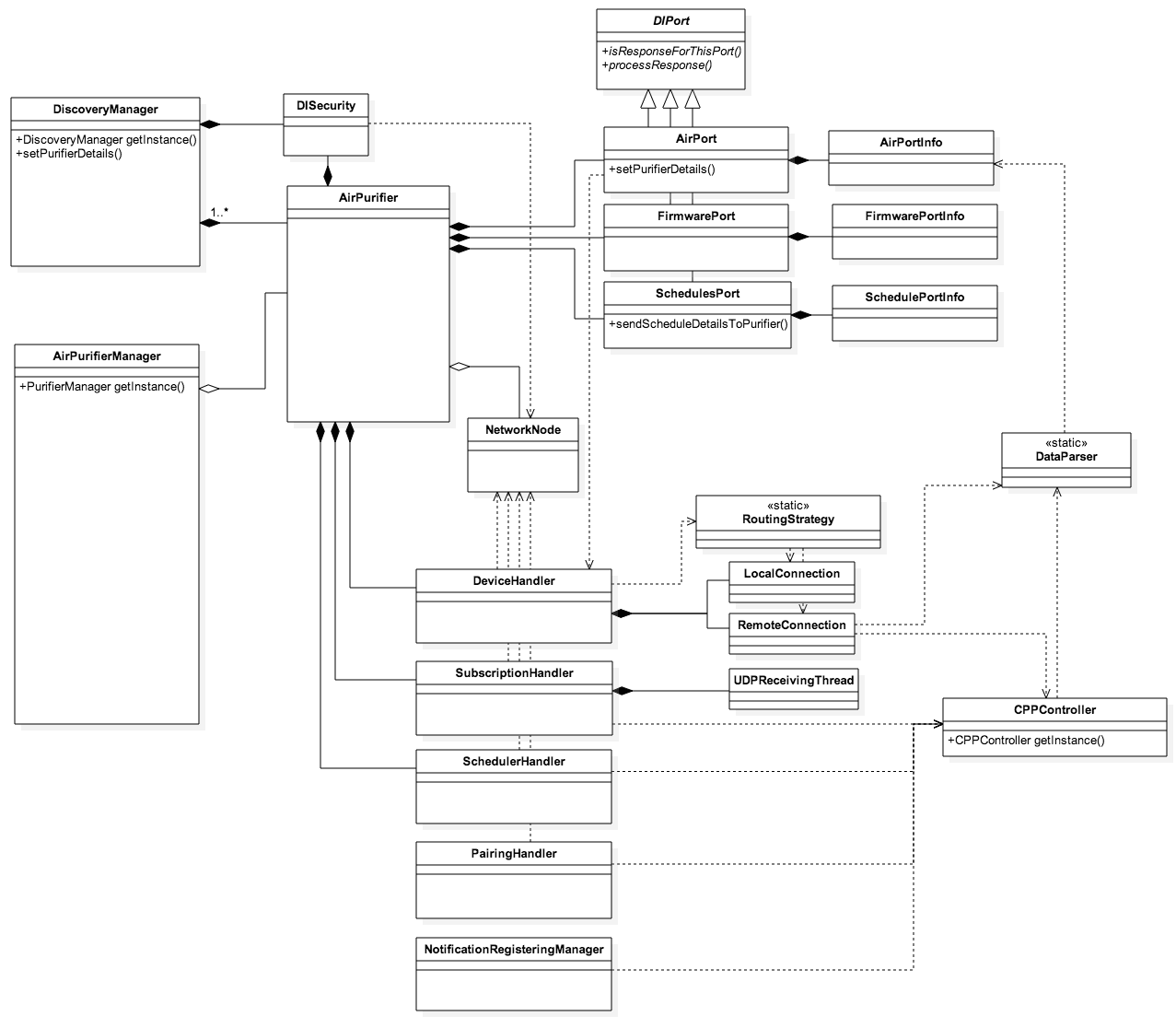


Figure : Proposed reusable architecture for Smart Air Android

1. Refactor request architecture

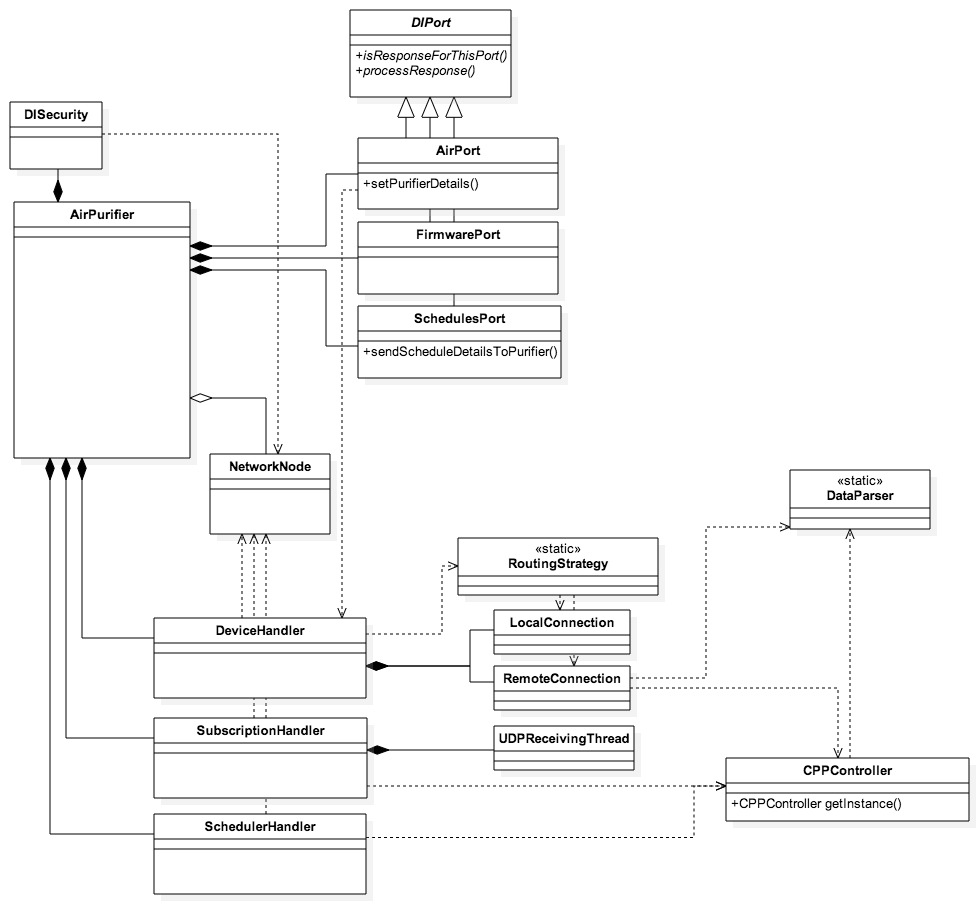


Figure : Exitsing DIComm request architecture.   
Reuse is limited by code duplication and the lack of standardized components.

The existing request architecture does not allow to easily add new DIComm ports to it and contains a lot of code duplication to achieve similar goals. Therefore quite some changes are required in order to maximize code reuse and minimize the implementation effort while adding new ports.

In order to make to allow simple DIComm ports to be added, the architecture must get a uniform mechanism to perform local and remote requests and a mechanism is necessary to choose the most appropriate connection.

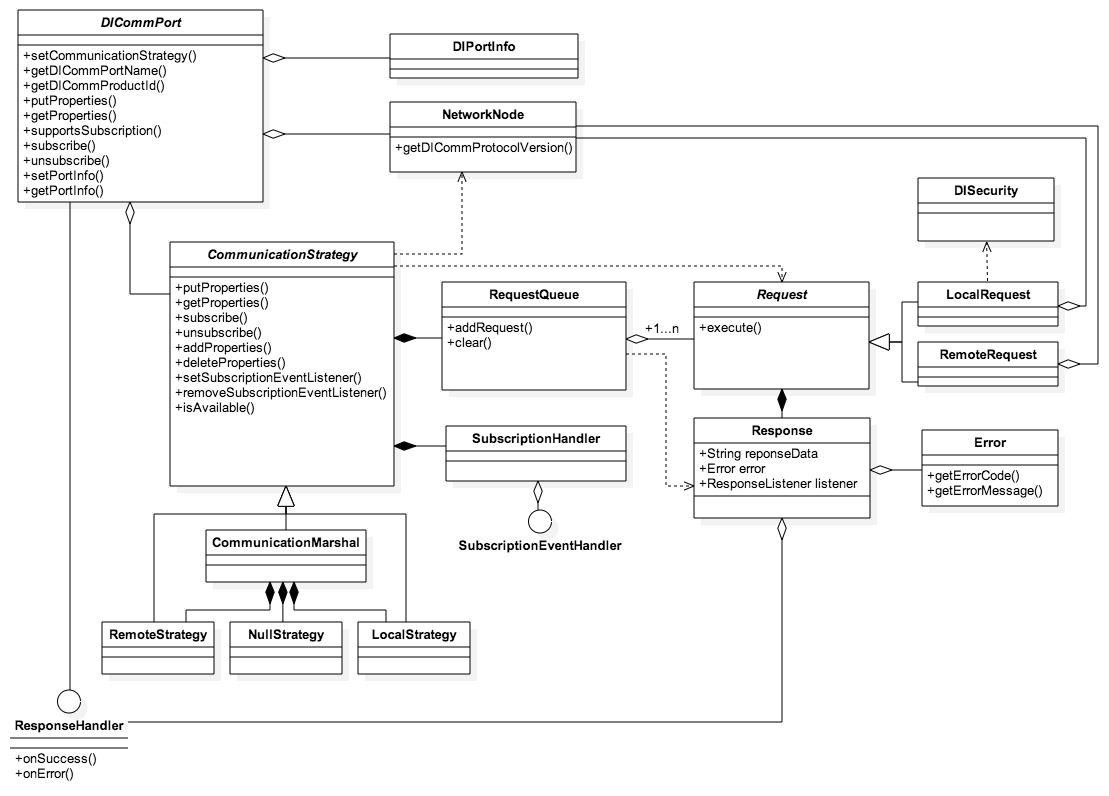


Figure : Proposed request architecture.   
The central Queue, CommunicationStrategy and Request objects enables reuse.

* 1. Add DIComm port configuration to DICommPort

Define two abstract methods in DICommPort class and implement them in all DICommPort subclasses.

public abstract String getDICommPortName();

public abstract String getDICommProductId();

* 1. Add DIComm protocol version to NetworkNode

Define a new integer field diCommProtocolVersion and add a getter for it to the NetworkNode class. Make sure this member is injected and initialized in the constructor.

private final int diCommProtocolVersion;

public int getDICommProtocolVersion() { … }

* 1. Generalize the SubscriptionEventListener interface

The SubscriptionEventListener interface should be generalized to completely remove the distinction between local and remote events. This will enable response decryption and potentially request retries in lower levels of the Android DIComm architecture.

In order to do this, first of all change the signatures of the onLocalEventReceived() and onRemoteEventReceived() methods to no longer take the ipAddress or CppId as a parameter. Note that the methods should no longer check if the event is coming from the right AirPurifier as every appliance now has their own handlers (e.g. DeviceHandler).

Next all references to onLocalEventReceived() method should first decrypt the received response before calling the onLocalEventReceived() method. In order to do this, use the DISecurity class to do the decryption and verify the decrypted message by checking if it is a valid JSON message.

Now the onLocalEventReceived() and onRemoteEventReceived() methods contain exactly the same methods and can hence be merged into one method.

public void onSuccess(String response)

Similarly the PurifierEvent parameter needs to be removed from the onLocalEventLost() method and replaced by an Error object parameter. This Error object should at least contain a unique error code and human readable message. Then rename the onLocalEventLost() method to a more generic version:

public void onError(Error error)

Finally, rename the SubscriptionEventListener interface to ResponseHandler.

* 1. Add a Response object

In order to generalize local and remote requests later on, one uniform Response object is necessary. This object should have the following:

* String response -> raw JSON (decrypted) response from local or remote request
* Error error -> Error object containing error code and message of the request
* ResponseHandler responseHandler -> object that needs to receive the result of the request, this is necessary to provide the Result on the main Thread later on.
* A constructor accepting all three parameters
* A method notifyResponseHandler() that calls the right method on the ResponseHandler with the correct data.
  1. Generalize the DeviceConnection interface

Convert the DeviceConnection interface to an abstract class modify the following:

* Rename DeviceConnection interface to Request
* Make setPurifierDetails() method abstract, change its return type to return a Response object and rename it to execute()
* Add a method convertDataToJson(Map<String, String> data) to convert any Map containing JSON keys and their values to one JSON as defined in [DICOMM].

The LocalConnection and RemoteConnection objects now need to be generalized so the can perform any kind of local/remote request that is required by the DIComm protocol. This will remove all request code duplication and provide a reusable component for all DICommPorts to use.

First of all, rename the LocalConnection to LocalRequest. Next change the constructor so it takes the following arguments:

* NetworkNode networkNode -> necessary to get the IP address of the AirPurifier
* String portName -> the port name as described by [DICOMM]
* Int productId -> the product id for a port as described by [DICOMM]
* RequestType requestType -> enum which is either PUT, GET, POST, DEL or ADD
* Map<String,String> data -> (optional) data to send with the request
* ResponseHandler responseHandler -> to handle the response once the request is completed

Next, all local requests in requests in the entire DIComm architecture need to be merged into the LocalRequest class and the correct configuration needs to be added according to the RequestType and the optional presence of data.

* LocalSubscription
* TaskPutDeviceDetails
* DemoModeTask
* EWSTasks
* DISecurityTask

Move the URL\_BASEALLPORTS from AppConstants to LocalRequest and add the necessary logic to LocalRequest to build port URLs from the DIComm version (in NetworkNode), portName and productId. Also move the decryption logic into the LocalRequest class and add the ability to do try one single key exchange when decryption fails before failing. Finally ensure the execute() method returns a proper Result object according to the success or failure of the requests.

Secondly, rename the RemoteConnection to RemoteRequest and change its constructor to take the following arguments:

* NetworkNode networkNode -> necessary to get the IP address of the AirPurifier
* String portName -> the port name as described by [DICOMM]
* Int productId -> the product id for a port as described by [DICOMM]
* RequestType requestType -> enum which is either SUBSCRIBE, UNSUBSCRIBE, ADDPROPS, DELPROPS, GETPROPS, PUTPROPS[[1]](#footnote-1)
* Map<String,String> data -> (optional) data to send with the request
* ResponseHandler responseHandler -> to handle the response once the request is completed

Extract a base template JSON for all simple DICommPorts from the getter methods in JSONBuilder class and move that template to RemoteRequest. Add the necessary logic to RemoteRequest to build the correct port JSON from the portName and productId using the template and to insert the proper parameters into the publishEvent() method from the CPPController. Finally ensure the execute() method returns a proper Result object according to the success or failure of the requests.

Finally add proper logging information to all Request classes and ensure that there are proper unit tests in place.

* 1. Replace the DeviceHandler by a RequestQueue

The DeviceHandler needs to be able to process generic Request objects, which means it needs to be made more generic than it currently is. Because these Requests can come from multiple ports, the DeviceHandler can no longer collect key-value pairs in a Map and send them all in once. (This logic will be moved to the DICommPort class later on.) Finally, a physical queue needs to be added to the DeviceHandler and it should no longer stop and start its Thread when idle because that will become inefficient due to the larger number of Requests that it will process. For all these reasons it will be more efficient to write an entirely new class than to modify the existing one.

The new RequestQueue class should adhere to the following requirements:

* It should have a queue of Requests
* It should be possible to add Requests to the back of the queue.
* It should process requests one by one
* Once a request is executed, the notifyResponseHandler() method should be called on the Response object. This callback must be invoked on the main Thread and the queue Thread does not need to wait for the callback to finish before starting the next Request.
* It should be possible to cancel all pending requests
  1. Add CommunicationStrategy classes

In order to differentiate between local and remote connection, every DICommPort class will have a CommunicationStrategy that does the proper routing of requests. This abstract class should have the following interface:

public abstract void getProperties(String portName, int productId, ResponseHandler responseHandler, NetworkNode networkNode);

public abstract void putProperties(Hasmap<String, String> data, String portName, int productId, ResponseHandler responseHandler, NetworkNode networkNode);

public abstract void addProperties(Hasmap<String, String> data, String portName, int productId, ResponseHandler responseHandler, NetworkNode networkNode);

public abstract void deleteProperties(String portName, int productId, int arrayPortId, ResponseHandler responseHandler, NetworkNode networkNode);

public abstract void subscribe(String portName, int productId, int timeToLive, ResponseHandler responseHandler, NetworkNode networkNode)

public abstract void unsubscribe(String portName, int productId, ResponseHandler responseHandler, NetworkNode networkNode)

public boolean isAvailable(NetworkNode node) {  
 return false;  
}

Next create a LocalStrategy subclass and add one RequestQueue member variable to this class (initialize it in constructor). Implement all abstract methods and have them create LocalRequest objects with the correct parameters and add these objects to the end of the RequestQueue. Override the isAvailable() method and return true when the ConnectionState is CONNECTED\_LOCALLY.

Then create a RemoteStrategy subclass and add one RequestQueue member variable to this class (initialize it in constructor). Implement all abstract methods and have them create RemoteRequest objects with the correct parameters and add these objects to the end of the RequestQueQueue. Override the isAvailable() method and return true when the ConnectionState is CONNECTED\_REMOTELY.

Create an NullStrategy subclass. Implement all abstract methods and have them call the onError() method of the ResponseHandler immediately on calling. Override the isAvailable() method and return true always.

Finally, create a CommunicationMarshal subclass and add one LocalStrategy , one RemoteStrategy and one NullStrategy as a member variable to this class. Implement all abstract methods and add a method findAvailableStrategy() to determine if the method calls should be routed to the LocalStrategy or RemoteStrategy. This method should prefer the LocalStrategy over the RemoteStrategy in the following way.

* 1. Modify DICommPort to start using the new request architecture

While the RequestQueue allows to queue requests, the request architecture currently still lacks the possibility to prioritize between requests and to combine multiple requests into one. In order to get such functionality, the DICommPort class should get an implicit queue itself.

First of all, the DICommPort needs to get a CommunicationStrategy member that will generate all requests towards the connected appliance. This member should be set in the constructor of the AirPurifier object using a setCommunicationStrategy() method.

private CommunicationStrategy communicationStrategy;

public void setCommunicationStrategy(CommunicationStrategy strategy);

Next, the DICommPort needs to know if there is an outstanding request for that port. To do this, add a new member hasOutstandingRequest and set it to true before starting a new request and to false when receiving a response (onSuccess() and onError()).

* Member private boolean hasOutstandingRequest

Then all put properties requests for all ports need to be done using the same method and multiple requests need to be able to be combined into one. For this, add a new method putProperties() that schedules the new changes into a HashMap (merge) and attempts to start performing the next request.

* Method public void putProperties(HashMap<String, String> map)
* Method private void performPutProperties()
* Member private HashMap<String, String> scheduledChanges

In order to keep the user informed about pending changes, DICommPort needs to track whenever a putProperties() request is ongoing or not. For this, add a boolean isApplyingChanges that is set to true in performPutProperties() and to false in onSuccess() and onError().

* Member private boolean isApplyingChanges
* Method public boolean isApplyingChanges()

Similarly all get properties requests for all ports also need to have the same interface. Therefore add a new method getProperties() that marks the boolean getPropertiesRequested as true and attempts to start performing the next request.

* Method public void getProperties()
* Member private boolean getPropertiesRequested

The subscribe and unsubscribe functionality also needs to become a part of the DICommPort class itself. Therefore add two methods subscribe() and unsubscribe() which first mark the booleans subscribeRequested and unsubscribeRequested as true respectively and then attempt to start performing the next request.

* Method public void subscribe()
* Member private boolean subscribeRequested
* Method public void unsubscribe()
* Member private boolean unsubscribeRequested

Note that in order to implement the subscribe() and unsubscribe() methods, the DICommPort class should define a time to live for the subscription. To do this, merge the LOCAL\_SUBSCRIPTIONTIME and CPP\_SUBSCRIPTIONTIME constants from AppConstants with the RESUBSCRIBING\_TIME from AirPurifier into the following constant of the DICommPort class:

private static final int SUBSCRIPTION\_TIMETOLIVE\_SEC = 300;

Now all that is necessary to complete the implicit queue with request prioritization and merging is to add a method tryToPerformNextRequest() which will do the following:

* Don’t start a new request when there is already an outstanding one (hasOutstandingRequest)
* Prioritize requests in the following order: 1. putProperties(), 2. subscribe(), 3. unsubscribe(), 4. getProperties()
* It should be called at the end of the onSuccess() and onError() method

private void tryToPerformNextRequest();

At this stage the DICommPort class offers all functionality to use the new request architecture, but the existing ports are still using the legacy public interface. Therefore the interface of the AirPort, FirmwarePort and ScheduleListPort must be merged with the new DICommPort methods at this stage.

Finally, delete the DeviceHandler completely as it is currently no longer used.

* 1. Make DICommPort easier to subclass

Because DICommPort will likely need to get subclasses for every project that will reuse the DI Comm Android architecture, it makes sense to make this process as smooth as possible.

Add a new generic member portInfo and methods setPortInfo() and getPortInfo() to the DICommPort class. These method should replace the already existing less generic methods to set and get the \*\*\*PortInfo object. Make sure to set the generic parameter <V> to the appropriate \*\*\*PortInfo class for every DICommPort subclass.

private <V> portInfo;

These new methods now allow two different optimizations to be add:

* getProperties() should trigger a getProperties() update the \*\*\*PortInfo object is null. This makes sure that clients don’t have to worry about updating properties themselves.

public <V> getPortInfo() {  
 if (portInfo == null) {  
 getProperties();  
 }  
 return portInfo;  
}

* setPortInfo() should discard any outstanding getProperties() request to minimize the amount of performed network calls.

protected setPortInfo(<V> portInfo) {  
 this.portInfo = portInfo;  
 getPropertiesRequested = false;  
}

Since not all DI Comm ports are equally interesting to subscribe to, it makes sense to have a mechanism to detect whether or not the app should subscribe to that port. Therefore add a new abstract method to DICommPort that returns if a client should subscribe to that port.

public abstract boolean supportsSubscription();

All subclass must implements this method accordingly. (Note that the main application for this method is to allow the DICommAppliance to bulk subscribe to all its ports).

* 1. Make it easier to add ports to DICommAppliance

A very common use case for DI Comm users will be to subscribe to all relevant ports of the DICommAppliance. In order to make this possible, the DICommAppliance should get a list of all DICommPorts that should be initialized with all ports in its constructor.

private List<DICommPort> ports;

Next add two convenience methods to subscribe() and unsubscribe() to all ports. They should loop over all ports, check to see if the port supportsSubscription() and only then call through to that port.

public void subscribe() {  
 for (DICommPort port : ports) {   
 if (port.supportsSubscription() ) {  
 port.subscribe();  
 }  
 }  
}

public void unsubscribe() {  
 for (DICommPort port : ports) {   
 if (port.supportsSubscription() ) {  
 port.unsubscribe();  
 }  
 }  
}

* 1. Move the SubscriptionHandler into the CommunicationStrategy

Because subscriptions are now started and stopped via the CommunicationStrategy, the resources required to have an active subscription also need to be managed by those components.

In order to do this, introduce a new SubscriptionEventHandler interface that will process all subscription events received from an appliance. This interface should only have one single method:

public void onSubscriptionEventReceived(String data)

Implement this interface in the AirPurifier class and in that method, loop over all ports pass the response to the right port. For instance:

if(getFirmwarePort().isResponseForThisPort()) {  
 getFirmwarePort().processResponse();  
}

Next, add a two new methods to the CommunicationStrategy to manage this listener namely setSubscriptionEventHandler() and removeSubscriptionEventHandler().

Then, split the subscriptionHandler from the AirPurifier into two and move one SubscriptionHandler into the LocalStrategy and one into the RemoteStrategy. After, override the setSubscriptionEventhandler() method in LocalStrategy and RemoteStrategy so they start all resources for the local and remote connection respectively. Similarly, the removeSubscriptionEventHandler() should be overridden to stop all resources.

public void setSubscriptionEventHandler(SubscriptionEventHandler handler);

public void removeSubscriptionEventHandler();

Please make sure to verify that the SubscriptionEventHandler is properly set and removed during the app lifecycle (fore/background).

Finally, note that the SubscriptionHandler should be split at this point in a remote and local variant and a generic superclass should be extracted. This is left as a future improvement.

* 1. Merge the subscription logic into DICommPort

Move all logic to keep a subscription alive from the AirPurifier class into the DICommPort class. This will allow subscriptions to individual ports.

* 1. Clean up operation

Review the existing code to find for unused classes, enumerations and constants. All these should be remove at this stage.

A non-exhaustive list would be:

* DeviceHandler class
* PurifierEvent enum
* Remove all subscription logic from the SubscriptionHandler (requests)

1. Component interface
   1. DICommApplianceFactory

Create a new abstract class DICommApplianceFactory, which specifies the following methods:

public abstract boolean canCreateDICommAppliance(NetworkNode node);

public abstract DICommAppliance createDICommAppliance(NetworkNode node);

Create a new class AirPurifierApplianceFactory which extends from this class and provides an implementation for these methods.

* In the canCreateDICommAppliance() return true when the modelName equals AirPurifier
* In the createDICommAppliance() do the following:
  + Create a new DISecurity()
  + Create a new CommunicationMarshal() with that security
  + Create a new AirPurifier using the NetworkNode and security
* In networkNode, ensure that the default ConnectionState is disconnected
  1. DiscoveryManager

Add two static methods for the initialization, which will store all necessary information for the component initialization.

public void createSharedInstance(DICommApplianceBuilder applianceBuilder)

public void createSharedInstance(DICommApplianceBuilder applianceBuilder, ApplianceDatabase applianceDatabase)

Create a new class DICommAppContext, which is a static class that will provide the DIComm component access to the Context at all times. This class should contain the following static methods:

Public static void initialize(ApplicationContext appContext)

Public static ApplicationContext getContext()

Ensure to initialize this component in the createSharedInstance() method.

Generalize AirPurifier specific fields to use the DICommAppliance:

* Change mDevicesMap to mAllAppliancesMap
  + Change AirPurifier type to DICommAppliance
* Change storedDevices to mAddedAppliances
  + Change AirPurifier type to DICommAppliance
* Change addNewPurifierListener to mNewApplianceDiscoveredListener
  + Change the classname to NewApplianceDiscoveredListener
  + Change method to onNewApplianceDiscovered()
* Add a new member for the DICommApplianceBuilder mApplianceBuilder
  + Initialize the member in the createSharedInstanceMethod
* Add a new member for the DICommApplianceSettings mApplianceSettings
  + Initialize the member in the createSharedInstanceMethod
* Ensure the ApplianceDatabase is properly initialized inside the constructor

Generalize AirPurifier specific methods to use the NetworkNode instead:

* Change isValidPurifier() to isValidNetworkNode()
  + Drop the check for the USN
* Change getPurAirDevice() to getNetworkNode()
  + Change the return type to NetworkNode
  + Only add the EncryptionKeyUpdatedListener when the NetworkNode is valid
  + Note: Building a device will happen in addNewDevice()
  + Remove usn completely
* onDeviceDiscovered() should only work with NetworkNode
* Change updateExistingDevice()
  + It should take NetworkNode as a parameter
  + Update Location should be moved to AirPurifier and called when someone requests location and the location is null
  + Rename to updateExistingAppliance()
* Change addNewDevice()
  + It should take NetworkNode as a parameter
  + Rename to addNewAppliance()
  + It should use the builder to build an Appliance and add that to the mDevicesMap
* Change onDeviceLost()
  + Change USN logic to CppID logic -> remove device when USN matches
  + Rename to onApplicanceLost()
  + Replace all AirPurifier references to DICommAppliance references
* Replace AirPurifier to DICommAppliance, and replace all references to device/purifier in method names, variable names and logs in the following methods:
  + markLostDevicesInBackground()
  + markOtherNetworkDevicesRemote()
  + markNonDevicesRemote()
  + markAllDevicesRemote()
  + markAllDevicesOffline()
  + updateConnectedStateViaCppAllPurifiers()
  + updateConnectedStateViaCppReturnedPurifiers()
  + updateConnectedStateOnlineViaCpp()
  + updateConnectedStateOfflineViaCpp()
  + connectViaCppAfterLocalAttemptDelayed()
  + cancelConnectViaCppAfterLocalAttempt()
  + syncLocalDevicesWithSsdpStackDelayed()
  + cancelSyncLocalDevicesWithSsdpStack()
  + printDiscoveredDevicesInfo()
  + setAddNewPurifierListener()( -> setNewApplianceDiscoveredListener
  + removeAddNewPurifierListener() -> clearNewApplianceDiscoveredListener
  + getDiscoveredDevices() -> getAllDiscoveredAppliances
  + getStoreDevices() -> getAddedAppliances()
  + updateStoreDevices -> updateAddedAppliances()
  + getDeviceByEui64 -> getApplianceByCppId()
  + getNewDevicesDiscovered()
  + onDiscoverEventReceived()
* Change removeFromDiscoveredList()
  + Add a todo that this method should be removed
  + Change AirPurifier to DICommAppliance
* Change updatePairingStatus()
  + Add a todo that this method should be removed
  + Change AirPurifier to DICommAppliance
* Change onSignedOnViaCpp() and onSignedOffViaCpp()
  + Add a TODO that the commented code should be fixed
  + Change purifier in logs to appliance

Finally, the DiscoveryManager should define a generic type <T>, which can be defined whenever the DiscoveryManager is created. This generic type is only necessary for the getter and setter methods of DiscoveryManager, internally it should use DICommAppliance everywhere.

1. Next steps

* Investigate database update requirements for this architectural change
* DISecurity object should be refactored so it can handle only one single appliance. This allows for the rest of the application to reuse the DISecurity object present in the AirPurifier object.
* Remove the parseAirPurifierEventData() method from the DataParser class, which is currently not yet possible due to its usage in the DeviceHandler class.
* Smarter enabling and disabling of remote subscription. (for instance via registering and unregistering listeners)
* Start and stop the Threads in the queue according to app lifecycle
* Clarify the interface between the callback interface DICommAppliance and the rest of the architecture.
* Clean up the getStoredDevices() and getDevices() -> all and added
* Describe how to add DICommApplianceBuilder and DICommApplianceSettings
* Describe how to move everything to a separate package: com.philips.cl.di.dicomm
* Add Proguard configuration file for DIComm
* Enable and disable logging
* PairingHandler, SchedulerHandler and NotificationRegisteringManager should be better blended into the architecture and not directly exposed to the outside of DI Comm.
* Add UI performance considerations from [DIPRA]:
  + Ignore incoming property updates while processing a put request
  + Don’t hide spinner or update incoming property updates when a new put request needs to be performed after receiving the result from the first one
  + Add a resubscribe and stopResubscribe mechanism to ports and expose that also via the DICommAppliance
* Prevent CPPController from signing in when remote control disabled
* Describe how to add device connection difficulties (exclamation mark in UI)
* Specify how to add the model type to differentiate between Jaguar and Powercube
* Describe how to generalize the AirPurifierDatabase to NetworkNodeDatabase
* Review how the Subscriptions are started and stopped (dcs and udp)
* Reuse the UDP subscription Thread between multiple devices.
* DISecurity should no longer be injected into the AirPurifier
* Only notify the UI when something actually changed -> filter out duplicate updates (for example for local UDP packages)

Document data and history

|  |  |  |  |
| --- | --- | --- | --- |
| **Document data** | | | |
| **Project name** | CDP common components | | |
| **Document name** | CDP common components – DI Comm client | | |
| **Document id** | C07S02 | **Date** | 2015-07-22 |
| **Version** | 0.8 | **Status** | Draft |

|  |  |  |  |
| --- | --- | --- | --- |
| **Document history** | | | |
| **Version** | **Date** | **Author** | **Reason** |
| 0.1 | 2015-01-14 | Jeroen Mols | Moved reuse instructions for Android from iOS architecture reference document. |
| 0.2 | 2015-01-20 | Jeroen Mols | Added reference architecture description of DiscoveryManager, PurifierManager and NetworkNode |
| 0.3 | 2015-02-02 | Jeroen Mols | Added new request architecture - steps |
| 0.4 | 2015-02-05 | Jeroen Mols | Updated reference architecture |
| 0.5 | 2015-02-05 | Jeroen Mols | Add design in diagrams |
| 0.6 | 2015-02-10 | Jeroen Mols | Added review feedback Stefan and instructions to disable remote control |
| 0.7 | 2015-02-16 | Jeroen Mols | Added feedback Matthijs |
| 0.8 | 2015-04-21 | Jeroen Mols | Added ListPort description |

**Open issues.**

|  |  |
| --- | --- |
| **Date** | **Subject** |
| 14-01-2015 | Add instructions to refactor component into separate package |
| 27-02-2015 | Split DiscoveryManager into separate component managing added devices |

1. Note that some building blocks are still allowed to directly communicate to the CPPController (e.g. CppDiscoveryHelper). [↑](#footnote-ref-1)