Iotivity Programmer’s Guide – Notification Manager

1 CONTENTS

2 REVISION HISTORY

Revision	Date	Author(s)	Comments
v0.1	9/29/2014	Samir Kant Sahu	Initial Release

3 NOTIFICATION MANAGER (NM)

This document provides interaction details and the programming model between the Rich Block and Lite Block clients respectively. The purpose is to provide enough detail to a developer to understand all the relevant use cases for the client and server model of the Lite Block Hosting.

The document is divided into three main chapters.

One that will describe how the hosting resources are discovered and hosted by another smart device, a Rich Block. The SDK and the service on the Rich Block provide a rich object oriented object model.

One that will describe how a hosting resource is registered and made available and consumable on the network by another smart device as a virtual resource. The examples for these interactions will be provided in a later section.

Lastly, one that will describe the interaction model between Lite Block, Hosting Smart Device and Consumer Smart Devices. The examples for these interactions will be provided further down.

4 ENHANCEMENTS

In subsequent releases, the IOT Notification Manager Service functionalities will be enhanced to include some of other important features as follows:

4.1 HOSTING SERVICE HANDLER

It will be mainly responsible for setting hosting flag. It will also handle the race conditions for Rich devices, to host simultaneously same Lite block resources and prevent duplicate hosting. Network topology change scenarios for the Lite block and the hosting device will be taken care by the handler.

4.2 PRIORITY SUBSCRIPTION

Priority settings to select dynamically between virtual resource and physical resource giving virtual resource more priority.

4.3 NOTIFICATION MANAGER SDK

APIs will be exposed for notification manager resource discovery.

4.4 VIRTUAL RESOURCE CACHING MECHANISM

Virtual resource tree management which will dynamically check the caching policy to address the topological changes of original hosted device.

4.5 NOTIFICATION CACHING MECHANISM

Notification data will be added to the persistent storage if required for supporting the pending and sync functionalities.

4.6 Notification Persistent Storage
Database has to be implemented to store and retrieve notification data asynchronously.

4.7 Notification Synchronization
An Iotivity Device can provide the state (create/read/update/delete) synchronization of the corresponding notifications with the target Iotivity Device.

4.8 Notification QoS (Priority)
Device can deliver the prioritized notification based on the requested QoS level.

4.9 Pending Notification
Notification Manager can cache notifications and send them to the target Iotivity Devices later. Check Subscribed Consumer comes within the range of the IOT devices.

4.10 Targeted Notification
Explicit notification for unsubscribed IOT devices

4.11 Network Topology Variation Handling
Handling of different user scenarios of topological changes in the network with respect to Lite and Rich blocks.

4.12 Relative Intelligent Hosting
Intelligence during hosting resource discovery to check the memory and power capability of the devices in network vicinity. This will help in selecting the best device whose resources shall be chosen to host.

5 Terminology

5.1 Virtual Resource

A virtual resource is a resource of another Lite device or smart device which is hosted by a smart device for other devices to consume via the hosting device.

5.2 Notification Manager Operations

Operations are actions that a Rich Device performs for hosting a particular resource of a Lite device. Event driven mechanism is implemented to detect the presence of devices whose resources needs to be hosted. Then targeted discovery of hosting resource is performed.

A resource is registered as virtual in smart device base to differentiate and give priority to the hosting resource. Notification Manager will have the priority settings to select dynamically between virtual resource and physical resource giving virtual resource more priority.

6 NM Architecture and Components

6.1 NM Architecture

This chapter defines the programming model on the RICH side to interact with Thin Block devices (and other RICH devices) for hosting/servicing resources. An introduction and overview to the main classes and interfaces will be given followed by specific usage scenarios along with sample code.

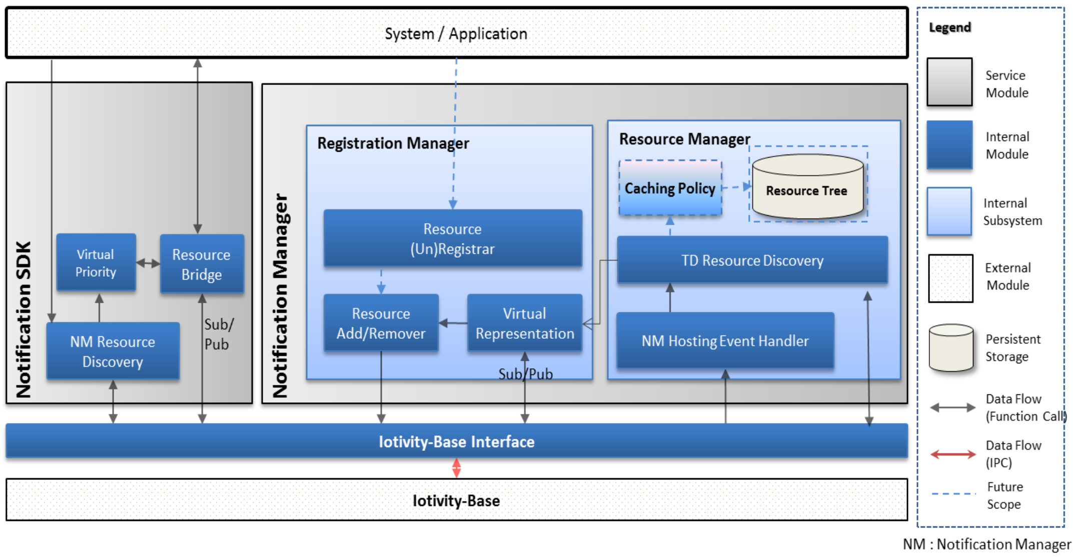

Figure 1. NM Architecture

6.2 NM Class Diagram

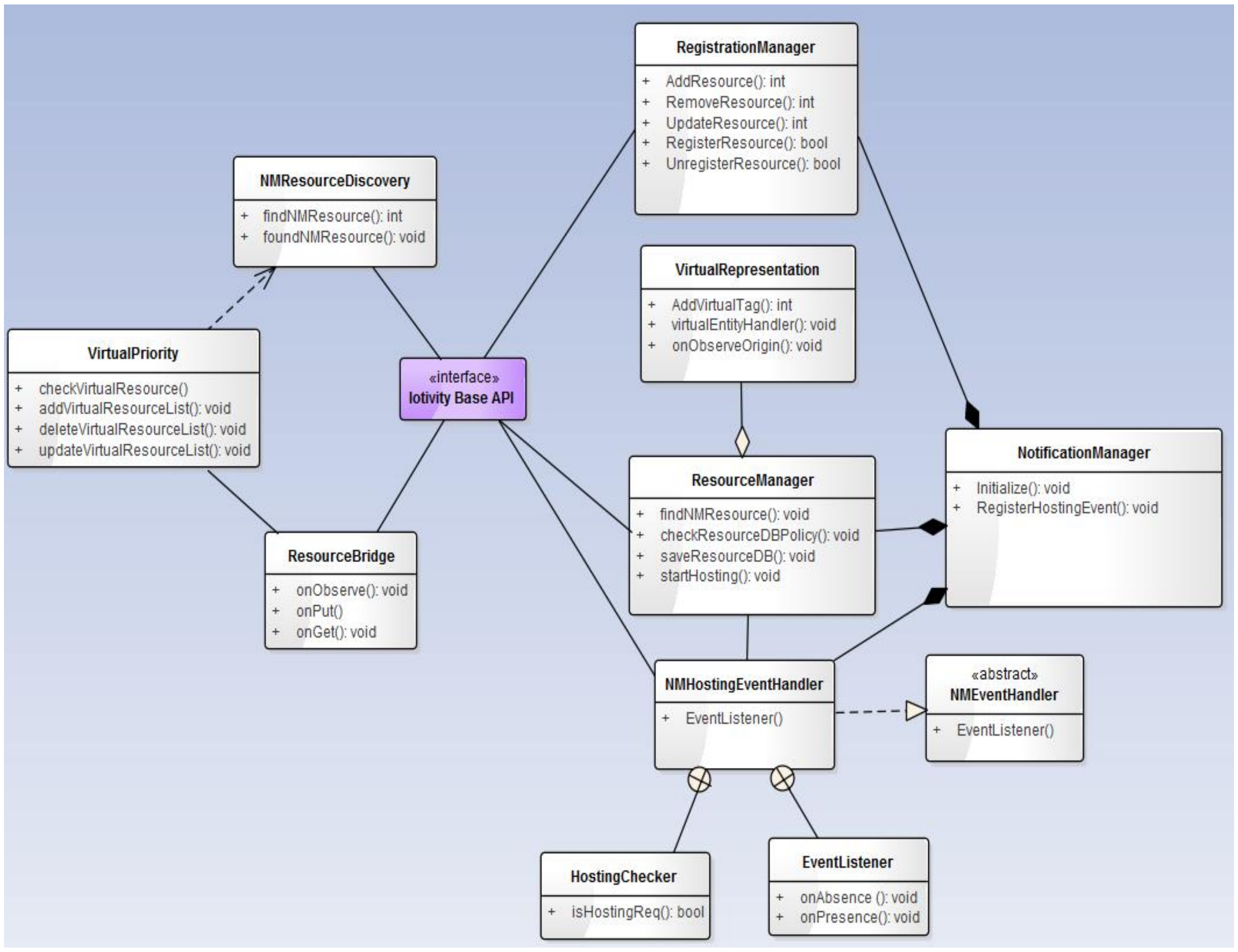

Figure 2. NM Class Diagram

6.3 NM Components

6.3.1 <class NotificationManager >

NotificationManager is the root service class which consists of two internal entities called the registration manager and resource manager. It is responsible for initializing the configuration parameters like the service operation mode, service type etc.

It also registers the event listeners for presence of devices whose resources need to be hosted.

6.3.2 <class RegistrationManager>

RegistrationManager handles the virtual representation of the Lite device resource and (un)registers virtual resource to IOT base.

6.3.3 <class ResourceManager>

ResourceManager is responsible for Lite device resource discovery. It implements the event driven mechanism of listening to the on presence of any Lite block devices in vicinity.

In future versions, resource manager will implement the caching policy and resource database for pending and sync cases.

6.3.4 <class VirtualRepresentation>

VirtualRepresentation is responsible for adding the virtual tag and virtual entity handler for the hosting resource.

6.3.5 Notification SDK

Notification SDK supports applications to find and subscribe the notification resources. It takes care of prioritizing virtual resource and mapping of the virtual resource to original resource.

7 SERVICE INITIALIZATION

Initializing the configuration parameters like the service operation mode, service type etc.

It also registers the event listeners for presence of devices whose resources need to be hosted.

NB: This event driven mechanism can be further enhanced to pre-check the power and memory capabilities of a device and dynamically decide which device needs its resources to be hosted.

```
void NotificationManager::initialize()

{

        cfg.ipAddress = "134.134.161.33";

        cfg.port = 5683;

        cfg.mode = ModeType::Both;

        cfg.serviceType = ServiceType::InProc;

        nmOCPlatform = new OCPlatform(cfg);

        //Register Hosting Event

        //Listen to on presence of Lite device

        //Check if Hosting Required

}
```

8 DISCOVERY

The advertisement/discovery mechanism for Lite Block clients is event driven. Discovery of hosting resource is done by listening to the on presence of the Lite devices. Querying or probing for all resource for the particular discovered Lite device is followed.

Once target Lite device is found, Lite device IP address can be used as host IP to direct resource discovery query.

Specific resource or all resources of Lite device can be hosted based on which the query parameters are varied and hence the mode of discovery.

(@param host, @param resource URI)	Mode of Discovery
(Not Empty, NULL/Empty)	Performs ALL resource discovery on a particular service
(Not Empty, Not Empty)	Performs query for a filtered/scoped/particular resource(s) from a particular service

```
void NotificationManager::findHostingCandidate()

{

        try

        {

            ResourceManager::getInstance()->findNMResource("",
            "coap://224.0.1.187/oc/core",true);

        }

        catch(OCException e)

        {

                std::cout << "Fail!!" << endl;

        }

}
```

9 Registering as Virtual Resource

In the preceding sections all the interaction has been to discover a resource from a Lite Block device. This section wraps up local resource management functionality by illustrating how a Rich Block device can register a hosting resource as virtual and make it available to other Rich Block devices.

Registering a resource as virtual is done by adding a virtual tag and making it available for other entities to discover and consume. A resource is made available by registering onto a CBServer object.

In the example below a light resource is registered.

Example: Registering a resource as virtual

```
bool RegistrationManager::registerNMResource(VirtualRepresentation &resourceObject,
std::shared_ptr<OCResource> resource)

{

        std::string uri               = resourceObject.getUri();

        std::string type              = resourceObject.getResourceTypeName();

        std::string interface= resourceObject.getResourceInterface();


        OCResourceHandle resourceHandle;

        OCStackResult result;

        result = NotificationManager::getOCPlatform()->registerResource(

                resourceHandle,

                uri,

                type,

                interface, std::function<void(std::shared_ptr<OCResourceRequest> request,

                std::shared_ptr<OCResourceResponse> response)>

                        (std::bind(&VirtualRepresentation::entityHandler,
                        resourceObject,

                        std::placeholders::_1,

                        std::placeholders::_2)),

                        resourceObject.getResourceProperty());

        resourceObject.setResourceHandle(resourceHandle);

        // On successful registration

        // Observe to the changes of the parameters of the original Lite

        // resource
```

10 OBSERVING A HOSTING PROPERTY

Once a resource of a Lite device is registered as virtual resource in the hosting device, it monitors that resource property for any state change. The hosting device would report back any change in state of the resource to any of the subscribers of that property.

```
bool RegistrationManager::registerNMResource(VirtualRepresentation &resourceObject,
std::shared_ptr<OCResource> resource)

{

        // Get the resource URI, interface, type and property flag


        // Register as virtual resource


        // On success, observe on the resource parameters of the original

        // Lite resource


        if(OC_STACK_OK != result)

        {

                std::cout << "Resource : " << uri << "Register Fail\n";

                return false;

        }

        else

        {

                QueryParamsMap queryParmaMap;

                resource->observe(ObserveType::Observe, queryParmaMap,

                        std::function<void(const OCRepresentation& rep, const int& eCode, const
                        int& sequenceNumber)>

                        (std::bind(&VirtualRepresentation::onObserve,

                        resourceObject,

                        std::placeholders::_1,

                        std::placeholders::_2,

                        std::placeholders::_3)));

        }

        return true;

}
```

11 NOTIFY ALL CONSUMERS

The hosting device notifies all consumers who are observing the hosted resource property of any change in state.

```
void VirtualRepresentation::onObserve(const OCRepresentation& rep, const int& eCode, const
int& sequenceNumber)

{

    if(eCode == SUCCESS_RESPONSE)

    {

        AttributeMap tmp = rep.getAttributeMap();

        if(NotificationManager::getInstance()->getOnObserve())

        {

                NotificationManager::getInstance()->getOnObserve()(tmp);

        }

        std::string targeturi = addVirtualTag(rep.getUri());

        OCStackResult result = OC_STACK_OK;

        VirtualRepresentation tmpObj = *this;

        if(!tmpObj.getUri().empty())

        {

                AttributeMap tmpAttMap = ResourceManager::getInstance()-
                >copyAttributeMap(tmp);

                {

                        // lock attribureMap

                        std::unique_lock<std::mutex> ck(mutexAttributeMap);

                        while(!isReadyAttributeMap)
        { conditionAttributeMap.wait(lck); }

                                isReadyAttributeMap = false;

                                attributeMap = tmpAttMap;

                                // unlock attributeMap

                                isReadyAttributeMap = true;

                                conditionAttributeMap.notify_all();

                        }

                        result = OCPlatform::notifyObservers(tmpObj.getResourceHandle());

                }

    }

}
```

12 Finding Hosting Resource By Consumer

The Rich block acting as a consumer can find the resource using the Notification Manager tag.

Example: Finding a light resource hosted by Notification Manager.

```
void findResourceCandidate()
{
        try
        {
                nmfindResource("", "coap://224.0.1.187/oc/core?rt=core.NMlight");
                while(true)
                {
                        // some operations
                }
        }catch(OCException& e)
        {
                //log(e.what());
        }
}
```