IoTivity Programmer's Guide – Things Manager

1 CONTENTS

2 REVISION HISTORY

Revision	Date	Author(s)	Comments
v0.1	9/29/2014	Andy Minho Lee	Initial draft
V0.2	9/29/2014	Jihun Ha	Initial release
V0.5	11/12/2014	Andy Minho Lee	Ready to release
V0.8	12/16/2014	Andy Minho Lee	Update of the contents for SDK API in M2
V0.8	12/17/2014	Jihun Ha	Added Things configuration & Diagnostics

3 THINGS MANAGER

This guide introduces a Things Manager which provides developers with very useful primitives to manage a group of things in the network. Things Manager also provides more rich functionalities to manage multiple things. With Things Manager functionalities and its offering SDK APIs, developers can easily implement a variety of applications to find candidate devices to form a group, create a group of found devices, create a group action for the group, and execute the group action. Also configuration of multiple things and diagnostics function for multiple things can be supported by this APIs. The purpose of this guide is to provide the details for developers to understand to fully utilize a given SDK APIs and how the Things Manager works to support the APIs. Lastly, this guide introduces explanatory scenarios for a group formation, group action, group synchronization, things configuration and things diagnostics and shows a series of procedure of sample applications to describes the scenarios for a better understanding on Things Manager.

4 TERMINOLOGY

4.1 GROUP

A set of devices in an IoTivity local network and remote networks for accomplishing the specific goal. Using several kinds of criteria, devices can be a member of a specific group. However, basically those member devices don't have any information about the group. Only the device that creates this group can have and maintain the information about this group.

Currently, resource type can be used as criteria for group formation and more criteria will be provided later.

4.2 ACTIONSET

A set of action descriptions needed by remote devices as the member of a specific group. For a particular group, multiple actions set can be assigned to this group. One action set can have multiple actions and one action should be assigned to one specific member devices' characteristic. Currently only resource type can be used as device's characteristic.

To create an action set, one may need to know the Delimeter serialization. With the Delimeter, one specifies an actionset as below.

> movieTime*uri=coap://10.251.44.228:49858/a/light|power=10*uri=coap://10.251.44.228:49858/a/light|power=10

A first segment before the first asterisk(*) is an actionset name. The second segment goes before a next asterisk. In the above example, "uri=coap://10.251.44.228:49858/a/light|power=10" is the segment. This can be also divided into two sub segments by a vertical bar(|): URI and a pair of attribute key and value. The remained string from the second asterisk is same as the second segment.

4.3 THINGS MANAGER

A software service which helps to shape a specific group and maintain that group. Group action feature - creating, maintaining and executing group action related with this group also provided by Things Manager

Richer API regarding configuration of multiple things and diagnostics of multiple things are provided by Things Manager.

A more detailed description of Things Manager and its relevant components will be provided later in this guide.

4.4 THINGS CONFIGURATION

A Things Configuration class in Things Manager provides several APIs to access a Configuration resource's value to get/update a system parameter. The extent of what a Configuration Resource covers could be all system-specific parameters. In this release, a Configuration resource partially covers system parameters on time (e.g., current time), network (e.g., IP address), and security (e.g., security mode).

4.5 THINGS DIAGNOSTICS

The purpose of a Things Diagnostics class in Things Manager is to request a system command (e.g., Factory Reset, Reboot) with a diagnostic purpose to a resource server by accessing a Diagnostics resource's value from a client remote in distance.

5 SDK API

SDK API is the facet of Things Manager to applications as shown in the Figure 1.

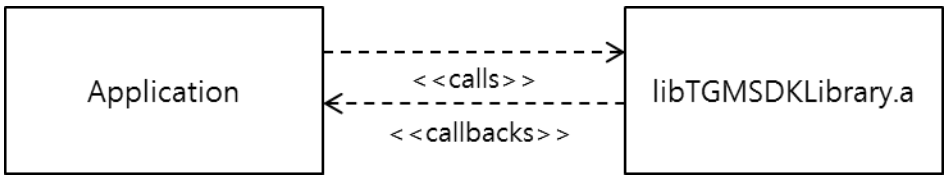


Figure 1. Things Manager SDK APIs and Application

5.1 GROUP MANAGER API

This APIs provides functions for application to find appropriate devices (i.e. things) in network, create a group of the devices, check a presence of member devices in the group, and actuate a group action in a more convenient way.

In the SDK, there are several operations provided;

- findCandidateResources

- subscribeCollectionPresence
- findGroup
- createGroup
- joinGroup
- leaveGroup
- deleteGroup
- getGroupList
- getStringFromActionSet
- getActionSetfromString
- addActionSet
- executeActionSet
- getActionSet
- deleteActionSet

findCandidateResource() is to request to Things Manager to find devices in network. The findCandidateResource() function has one major parameter to give a condition to find appropriate devices out of all devices in network. One of conditions provided is a resource type which a device has.

With the parameter, developers can indicate a list of resource types they want to find and create a group. The prototype of the function is shown as follows:

- OCStackResult **findCandidateResources**(std::vector< std::string > *resourceTypes*, std::function< void(std::vector< std::shared_ptr< OCResource > >) > *callback*, int *waitsec* = -1);

subscribeCollectionPresence() is to ask to Things Manager to subscribe the presence state of the resource. The first parameter is *resource* which is the resource pointer to check the presence state. The second parameter is *callback* which is called when the presence state is changed, for example power off. The prototype of the function is shown as follows:

- OCStackResult ThingsManager::**subscribeCollectionPresence**(std::shared_ptr< OCResource > *resource*, std::function< void(std::string, OCStackResult) > *callback*)

findGroup() is to ask a Things Manager to find a specific remote group when a resource tries to join a group. The first parameter is *collectionResourceTypes* which is the resource types of a group to find and join. The second parameter is *callback* which is called whenever a group is found or not. If a group is found, OCResource of callback is the group resource. Otherwise, OCResource is NULL. The prototype of the function is shown as follows:

- OCStackResult **findGroup** (std::vector< std::string > *collectionResourceTypes*, FindCallback *callback*)

createGroup() is to ask a Things Manager to create a group that is not existed. The only parameter is *collectionResourceType* which is the resource type of a group to create. The prototype of the function is shown as follows:

- OCStackResult **createGroup** (std::string *collectionResourceType*)

joinGroup() is to ask a Things Manager to join a existing group. This is a kind of overloaded functions and has two types. One is used when a resource that has a group tries to find a specific remote resource and makes it join a group. The first parameter is *collectionResourceType* which is the resource type of a group to join. The second parameter is *resourceHandle* which is the resource handle to join a group. The prototype of the function is shown as follows:

- OCStackResult **joinGroup**(std::string *collectionResourceType*, OCResourceHandle *resourceHandle*)

On the other hand the other is used when a resource that doesn't have a group tries to find and join a specific remote group. The first parameter is *resource* which is the group resource pointer to join. This can be the callback result of findGroup(). The second parameter is *resourceHandle* which is the resource handle to join a group. The prototype of the function is shown as follows:

- OCStackResult **joinGroup**(const std::shared_ptr< OCResource > *resource*, OCResourceHandle *resourceHandle*)

leaveGroup() is to ask a Things Manager to leave a joined group. The fist parameter is *collectionResourceType* which is the resource type of a group to leave. The second parameter is *resourceHandle* which is the resource handle to leave a group. The prototype of the function is shown as follows:

- OCStackResult **leaveGroup**(std::string *collectionResourceType*, OCResourceHandle *resourceHandle*)

deleteGroup() is to ask a Things Manager to delete a created group. The only parameter is *collectionResourceType* which is the resource type of a group to delete. The prototype of the function is shown as follows:

- OCStackResult **deleteGroup**(std::string *collectionResourceType*)

getGroupList() is to ask a Things Manager to get a list of joined groups. This has no parameter and returns the map with the resource type of a group and group resource handle. The prototype of the function is shown as follows:

- std::map< std::string, OCResourceHandle > **getGroupList**()

getStringFromActionSet() is to ask a Things Manager to translate an instance of ActionSet class into a string serialized in Delimiter format and get the string. The only parameter is a *newActionSet* which is an target instance of ActionSet class. The prototype of the function is shown as follows:

- std::string **getStringFromActionSet**(const ActionSet **newActionSet*)

getActionSetfromString() is to ask a Things Manager to translate a string serialized in Delimeter format into a instance of ActionSet class and get the instance. The only parameter is a *desc* which is an target string serialized in Delimiter format. The prototype of the function is shown as follows:

- ActionSet* **getActionSetfromString**(std::string *desc*)

addActionSet() is to ask a Things Manager to add a new actionset onto a specific resource. The first parameter is a *resource* which a new actionset will be added onto. The second parameter is a *newActionSet* which is a target instance of ActionSet class to be added. The last parameter is a *cb* which is callback function. The prototype of the function is shown as follows:

- OCStackResult **addActionSet**(std::shared_ptr< OCResource > *resource*, const ActionSet* *newActionSet*, PutCallback *cb*)
- typedef std::function<void(const HeaderOptions&, const OCRepresentation&, const int)> PutCallback;

executeActionSet() is to ask a Things Manager to execute a specific actionset belonging to a specific resource. The first parameter is a *resource* which is a target resource. The second parameter is a *actionsetName* which is a string indicating an actionset name to be executed. The last parameter is a *cb* which is callback function. The prototype of the function is shown as follows:

- OCStackResult **executeActionSet**(std::shared_ptr< OCResource > *resource*, std::string *actionsetName*, PostCallback *cb*);
- typedef std::function<void(const HeaderOptions&, const OCRepresentation&, const int)> PostCallback;

getActionSet() is to ask a Things Manager to get an existing actionset belonging to a specific resource. The first parameter is a *resource* which is a target resource to be retrieved. The second parameter is a *actionsetName* which is a string indicating an actionset name. The last parameter is a *cb* which is

callback function. When the callback function is called, an actionset string in Delimeter format will be returned. The prototype of the function is shown as follows:

- OCStackResult **getActionSet**(std::shared_ptr< OCResource > *resource*, std::string *actionsetName*, PostCallback *cb*);
- typedef std::function<void(const HeaderOptions&, const OCRepresentation&, const int)> PostCallback;

deleteActionSet() is to ask a Things Manager to delete an existing action set belonging to a specific resource. The first parameter is a *resource* which is a target resource. The second parameter is a *actionsetName* which is a string indicating an actionset name to be deleted. The last parameter is a *cb* which is callback function.

- OCStackResult **deleteActionSet**(std::shared_ptr< OCResource > *resource*, std::string *actionsetName*, PostCallback *cb*)
- typedef std::function<void(const HeaderOptions&, const OCRepresentation&, const int)> PostCallback;

5.2 THINGS CONFIGURATION & DIAGNOSTICS API

5.2.1 Things Configuration

There are two main usages of this class: (1) On a server side, bootstrapping requisite information (i.e. system configuration parameters) from a bootstrap server to access other IoT services, (2) On a client side, getting/updating the system configuration parameters from/to multiple remote things.

On the server side, there is only one API for bootstrapping;

- OCStackResult **doBootstrap**()

On operating at first stage, a resource server should fetch a bunch of configuration information from a bootstrap server to configure itself to access other IoT services. The information includes a system time information(e.g, time zones), network information (e.g., IP Address), and security information (i.e., access control list). After fetching, the information has been stored in forms of resources, namely Configuration resource.

On the client side, there are two APIs for getting/updating a resource value from/to resource server(s);

- OCStackResult **updateConfigurations** (std::shared_ptr< OCResource > *resource*,

std::map< ConfigurationName, ConfigurationValue > *configurations*,
ConfigurationCallback *callback*)

- OCStackResult **getConfigurations** (std::shared_ptr< OCResource > *resource*,
  std::vector< ConfigurationName > *configurations*,
  ConfigurationCallback *callback*)
- typedef std::function<void(const HeaderOptions& headerOptions, const OCRepresentation& rep, const int eCode) > ConfigurationCallback;

The first parameter, *resource*, is a pointer of resource instance of Configuration resource. The resource pointer can be acquired by performing findResource() function with a dedicated resource type, "**oic.con**". Note that, the resource pointer represents not only a single simple resource but also a collection resource composing multiple simple resources. In other words, using these APIs, developers can send a series of requests to multiple things by calling the corresponding function at once.

The second parameter, *configurations*, represents an indicator of which resource developers want to access and which value developers want to update. Basically, developers could use a resource URI to access a specific resource but a resource URI might be hard for all developers to memorize lots of URIs, especially in the case of long URIs. To relieve the problem, Things Configuration introduces a easy-memorizing name, called ConfigurationName, instead of URI. And ConfigurationValue used in updateConfiguration() function indicates a value to be updated. Note that, only one configuration parameter is supported in this release. Multiple configuration parameters will be supported in future release.

The last parameter, *callback*, is a callback function which is called when a response for the request just arrives.

As mentioned briefly, to access a resource with the above APIs, developers should know a ConfigurationName replacing a resource URI. Things Configuration provides an additional APIs for this;

- std::string **getListOfSupportedConfigurationUnits**()

When calling this function, developers get to know which Configuration Names are supported and their brief descriptions. This information is provided in JSON format.

5.2.2 Resource Model of Things Configuration

Configuration resource adopts IPSO (IP Smart Object Guidelines, http://www.ipso-alliance.org) concept. In IPSO resource model, there are two entities composing the resource model: object and resource. The notions of object and resource in IPSO concept can be naturally realized to collection resource and simple resource in IoTivity domain. All configuration parameters are stored as follows;

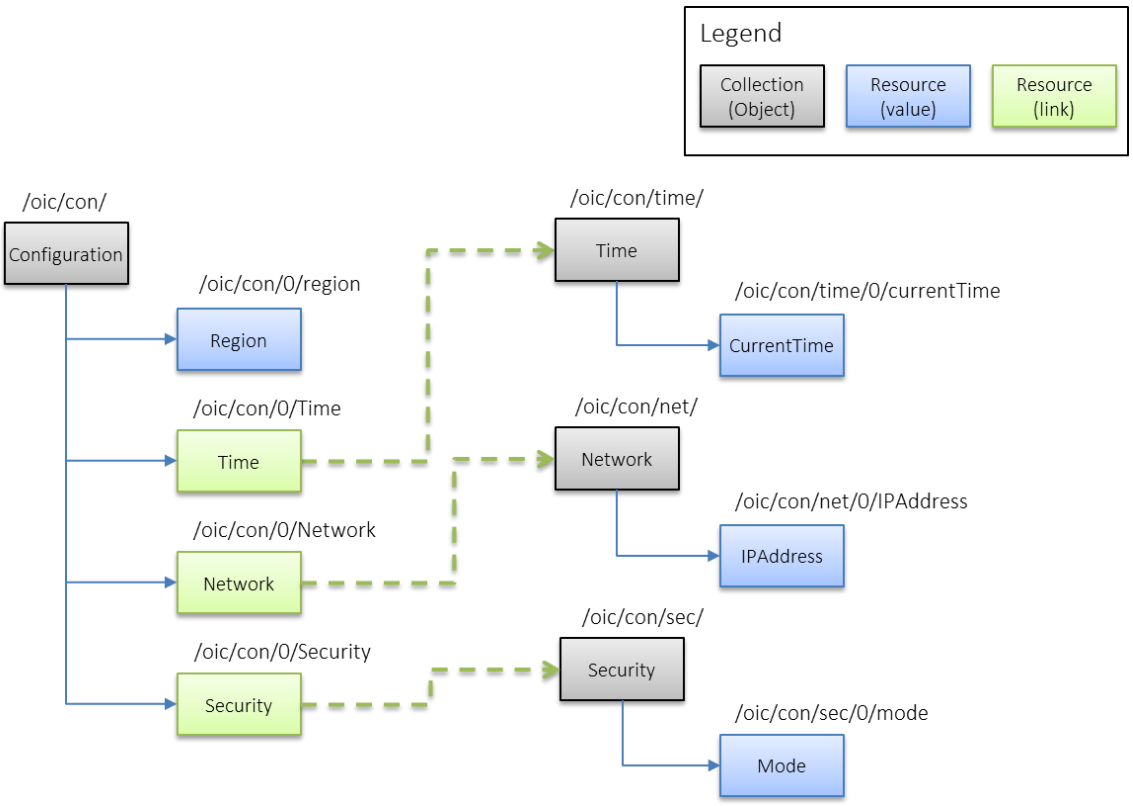


Figure 2 Resource model of Configuration Resource and its child resources(Time, Network, Security)

(Note that contents in resources are not standardized yet so it can be changed in the future.)

A distinctive point of Figure 2 is an employment of resource link (i.e. Green box). It is because IPSO resource model is flat so it has some limitations to express multiple hierarchical resource models. In resolving the problem, a resource link is very useful to connect simple resource to other collection resource in order to express the comprehensive hierarchy like Figure 2.

5.2.3 Things Diagnostics

There are two functionalities in Things Diagnostics; (1) FactoryReset to restore all configuration parameters to default one, and (2) Reboot to request a system rebooting.

The prototypes of APIs to provide these functionalities are as following;

- **factoryReset** (std::shared_ptr< OCResource > *resource*, DiagnosticsCallback *callback*)
- **reboot** (std::shared_ptr< OCResource > *resource*, DiagnosticsCallback *callback*)
- typedef std::function<void(const HeaderOptions& headerOptions, const OCRepresentation& rep, const int eCode) > DiagnosticsCallback

These APIs are very simple; developers give a pointer of found resource object and callback function. Like Things Configuration, the pointer can be for a collection resource as well as a simple resource. That is, when the pointer is for a collection resource, a request from ThingsDiagnotics will target multiple resources.

The first function, factoryReset(), is used to restore all configuration parameters to default one. All configuration parameters refers Configuration resource, which they could have been modified for various reasons (e.g., for a request to update a value). If developers on the client want to restore the parameters, just use the factoryReset() function.

Additionally, for the purpose of storing default configuration parameters on a resource server side, the server maintains a *FactorySet* resource which stores all default configuration parameters. If the server receives a request of FactoryReset() from the client, the server refers a *FactorySet* resource to restore all parameters of Configuration Resource. Note that, a client can access a *FactorySet* resource to read default parameters but can not modify any parameter of the resource.

The second function, reboot(), is used to send a request to a server to be rebooted. On receiving the request, the server attempts to reboot itself in a deterministic time.

5.2.4 Resource Model of Things Diagnostics and FactorySet

The resource models of Things Diagnostics and FactorySet are also structured in IPSO resource model. The diagnostics resource is as follows;

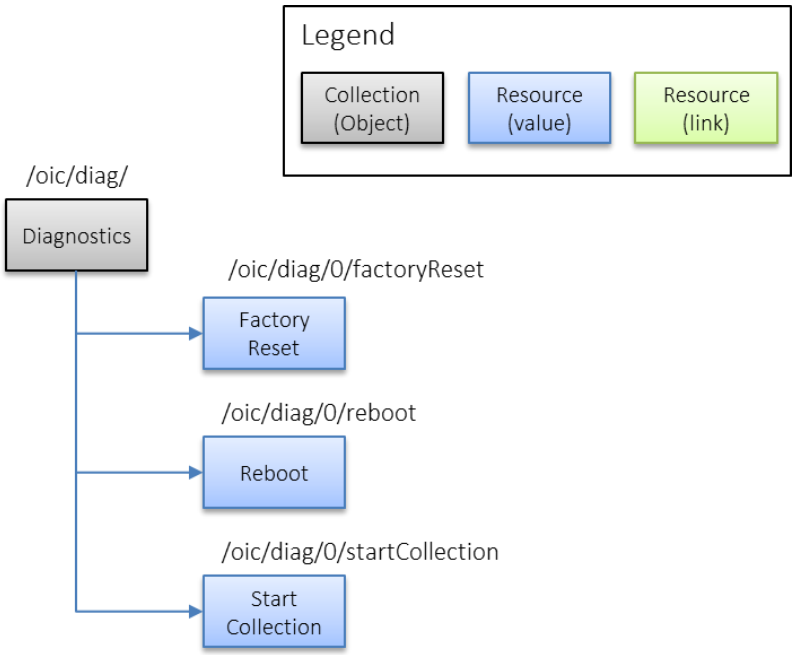


Figure 3 Resource model for Diagnostics Resource

Diagnostics resource has three simple resources:Factory Reset, Reboot, and Start Collection. The Factory Reset and Reboot resources are responsible for FactoryReset and Reboot functionalities in Things

Diagnostics, respectively. The other resource, Start Collection, is to start collecting device-related statistics itself when the resource's value is set to *TRUE*.

The FactorySet resource is as follows;

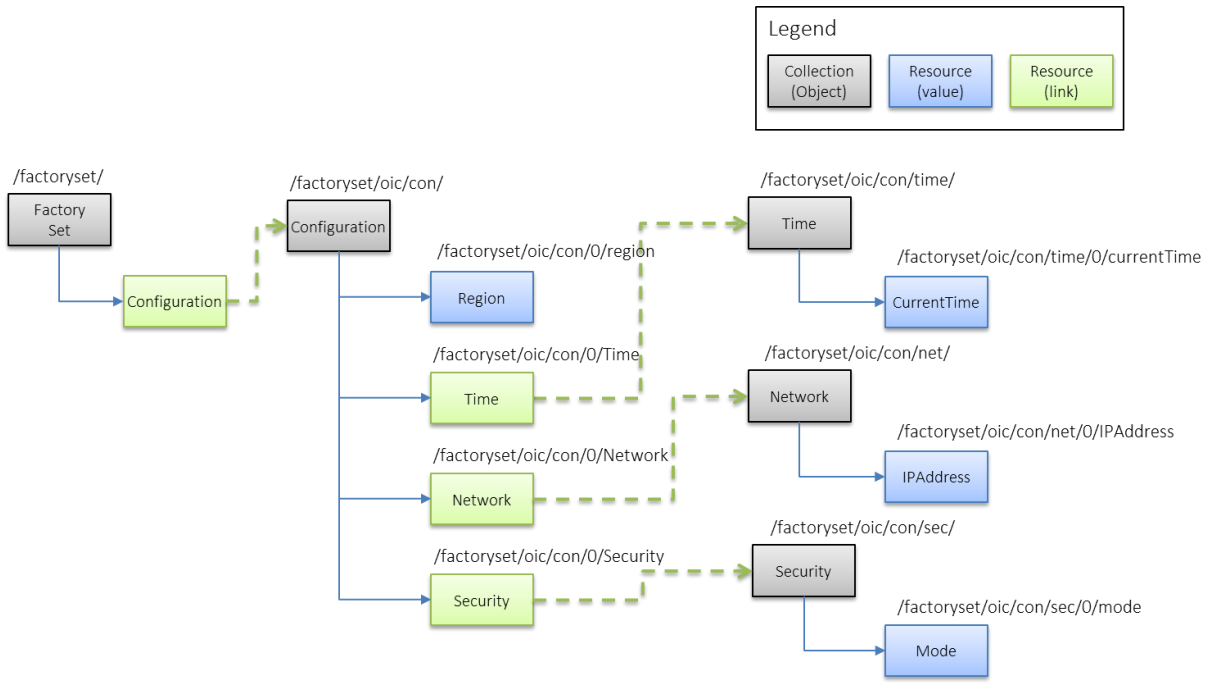


Figure 4 Resource model for FactorySet Resource

As mentioned in Section 5.2, FactorySet stores all default configuration parameters, so it embodies a copy of configuration resource. However, the copy of configuration resource in FactorySet resource has different URIs from a configuration resource's URIs described in Figure 2.

6 THINGS MANAGER ARCHITECTURE

6.1 CONTEXT DIAGRAM

Things Manager is basically operated in the IoTivity Base messaging environment as shown in the Figure 5.

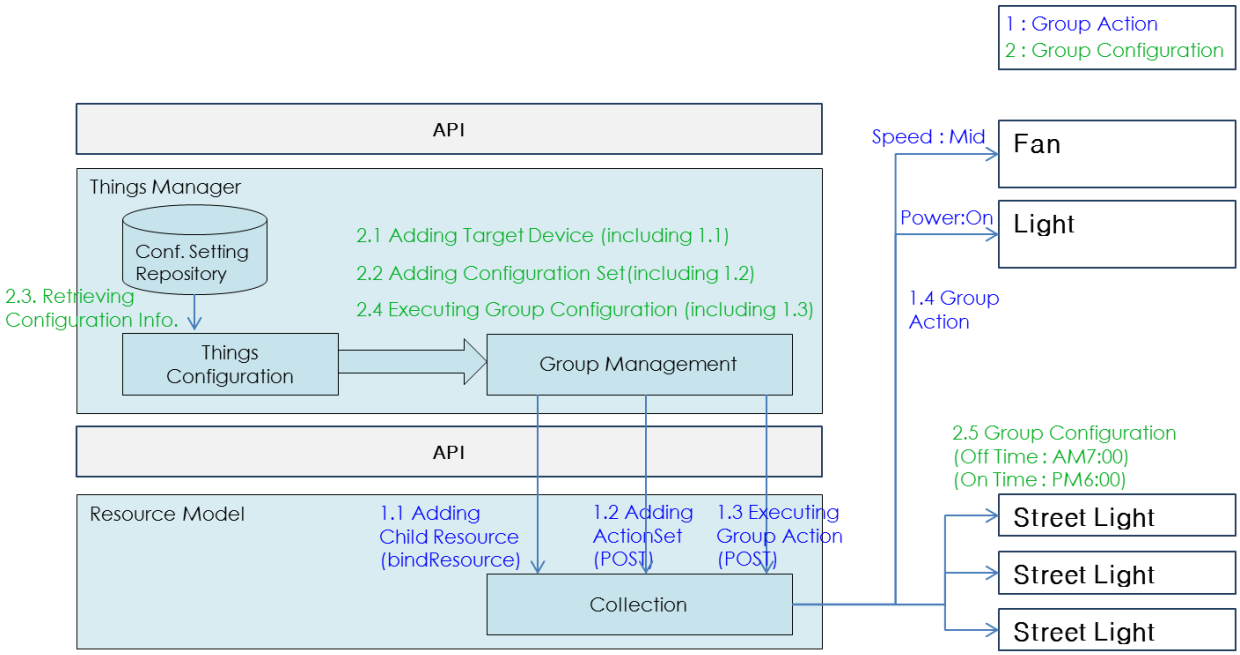


Figure 5. Things Manager Context Diagram

There are two kind of usages in the Things Manager;

- SDK API usage
- Raw API usage

For the first usage, Things Manager provides SDK API described in the previous section which hides the details of IoTivity functions and protocols and provides a simple operation set in C++.

The raw API also can be used to use main function of group management (i.e. group action) and other functionalities of group management can be emulated by using raw API. The raw API introduces Resource based concept and matched with CoAP's basic functionalities (i.e. GET/PUT/POST/DELETE). This API now provided as a simple operation set in C language.

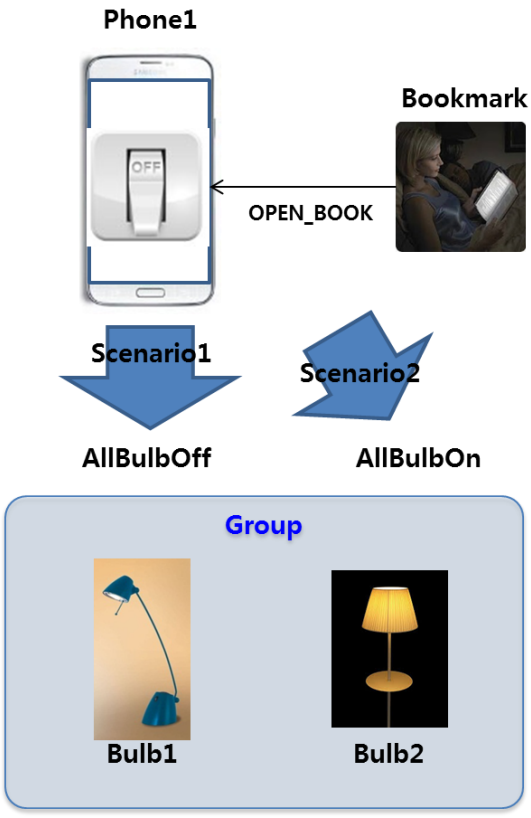
7 EXAMPLE : GROUP FORMATION & GROUP ACTION

This section is to show the functionality of group management and group action. With example, it can be understood how group can be made and how can we use it.

A Group can have homogeneous or heterogeneous member things. Multiple groups can be handled simultaneously.

Group Action can be triggered by user's action (i.e. switch control) or by registered action. User application can subscribe this kind of action to particular group. Subscription function will be provided by next release

7.1 EXAMPLE SCENARIO



Role		Scenario1	Scenario2
Actuator	Bulb1	On->Off	Off->On
	Bulb2	On->Off	Off->On
Sensor	Bookmark	N/A	Open
Command UI & TM	Phone1	Things Graph Manager working	
		Switch On->Off	N/A

- Scenario 1: Bulb Control Service

There are mainly two entities composing "Bulb Control System": one mobile phone and two bulbs. This scenario shows that users can easily manage their at-home light bulbs with their mobile phone. One of convincing situations for the scenario is the moment that users leave home for work; they may want to turn all bulbs at home off with their phone.

① At initial stage, we assume that two bulbs are already on.

② After an application to control the bulbs executed on the phone, , the phone discovers the bulbs around itself and sets them as “Group” for Bulb Control Service.

③ After “Group” made, user can see a switch to control.

④ Once clicking the switch to off, the bulbs are going to turning off.

- Scenario 2: Help Your Reading Service (using bookmark)

There are mainly three entities composing “Help Your Reading System”: One bookmark with Arduino platform, two bulbs with Arduino platform. This scenario shows that when a specific user is about to read a book, the bookmark put in the book detects users’ intention and seamlessly the bulbs produce a comfortable mood for reading.

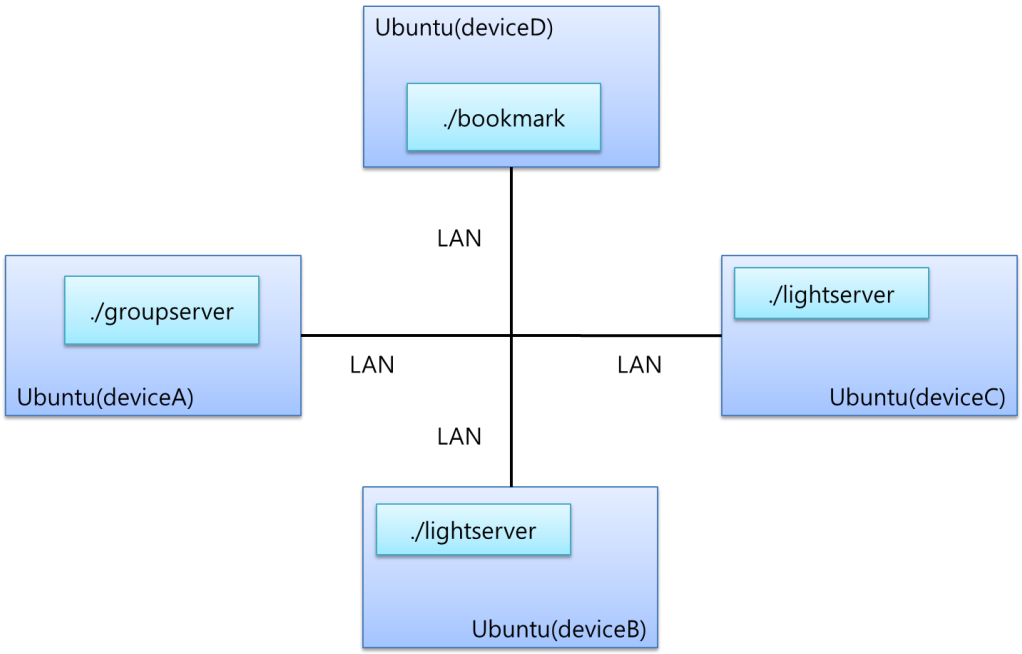
① At initial stage, we assume that the bulbs are already off and the bookmark is put in a book.

② The phone discovers the bookmark and bulbs. First, when all bulbs are found, “Group” can be made. And when the bookmark is found, the phone requests the bookmark to observe a book’s openness. After that, the bookmark will notice its status whenever it changes. (open→close, close→open)

③ A book reader opens his/her book.

④ Then the bulbs are going to turn.

7.2 WORKING FLOW



[Example Topology]

This section introduces an experimental example of the scenarios run on Ubuntu platform. This section describes how Things (i.e. bulbs and bookmark) can make group communication and what kind of features can be provided by Things Manager, specifically Group Manager.

① Run all light bulb applications and a bookmark application

First of all, users execute light bulb applications corresponding to actuators (called resources from here). Create the Ubuntu device topology according to "Example Topology" and run executables

To execute all bulb applications, enter as follows:

```
~/deviceB/service/things-manager/build/linux/release$ ./lightserver
Resource URI : /a/light
     Resource Type Name : core.light
     Resource Interface : oc.mi.def
     Resource creation is successful with resource handle : 0x1dd1de0
…
~/deviceC/service/things-manager/build/linux/release$ ./lightserver
…
```

As seeing above, users can know details of the registration for the bulb resource including resource URI, resource type.

Next, users execute a bookmark. To do, enter as follows:

```
~/deviceD/service/things-manager/build/linux/release$ ./bookmark
Resource URI : /core/bookmark
     Resource Type Name : core.bookmark
     Resource Interface : oc.mi.def
     Resource creation is successful with resource handle : 0xfd6da0
```

② Run a groupserver application to control a group

Lastly, users execute a user application, called a "groupserver". Once the application is executed, it automatically attempts to discover all candidate resources in a network. In this example, the candidate resource is a light bulb resource with a resource type "core.light".

After a few seconds (e.g., 5 seconds), the application creates a group composed of the found light blub resources. If the creation is successful, users can see a menu to ask users' input.

To run the application, enter as follows:

```
~/deviceA/service/things-manager/build/linux/release $ ./groupserver


DISCOVERED Resource:
     URI of the resource: /a/light
     Host address of the resource: coap://10.251.42.143:45631
     List of resource types:
            core.light
            core.brightlight
     List of resource interfaces:
            oc.mi.def
            oc.mi.ll


(additional resources can be founded)


CHILD RESOURCE OF GROUP
     URI :: coap://10.251.42.143:45631/a/light
     URI :: coap://10.251.44.228:52997/a/light
     …


1 :: CREATE ACTIONSET 2 :: EXECUTE ACTIONSET(ALLBULBON) 3 :: EXECUTE ACTIONSET(ALLBULBOFF)
4 :: GET ACTIONSET 5 :: DELETE ACTIONSET 6 :: QUIT
0 :: OBSERVE TO BOOKMARK
_(prompt)
```

③ Create group actionsets,"AllBulbsOn" and "AllBulbsOff"

The next step is to define group actionsets for the group. In this example, there are two predefined group actionsets: "AllBulbsOn" and "AllBulbsOff".

To create a group actionset, users need to specify a new ActionSet class instance. How to specify is shown as follows:

~/deviceA/service/things-manager/sampleapp/linux/groupaction/groupserver.cpp 177

```
…
void createActionSet_AllBulbOff()
{
    ActionSet *allBulbOff = new ActionSet();
    allBulbOff->actionsetName = "AllBulbOff";

    for(auto iter = lights.begin(); iter != lights.end(); ++iter)
    {
        Action *action = new Action();
        action->target = (*iter);

        Capability *capa = new Capability();
        capa->capability = "power";
        capa->status = "off";

        action->listOfCapability.push_back(capa);
        allBulbOff->listOfAction.push_back(action);
    }

    g_ thingsMgr->addActionSet(resource, allBulbOff, callback);
```

First, a name of group actionset is specified in *actionsetName* variable of ActionSet instance. Next, for each light bulb resource, users need to create a Action class instance and specify the light bulb resoure's URI in *target* variable of Action instance. And another class instance, called Capability, is need to be created to specify an attribute key and value of the target resource. The attribute key and value are written in *capability* and *status* variables of Capability instance class, respectively. After filling the Capability instance, store it to *listOfCapability* vector variable of Action instance and store the Action instance to *listOfAction* vector variable of ActionSet instance.

On the ActionSet instance for "AllBulbsOff" is specified, users just use addActionSet() function provided in ThingsManager class.

To execute two group actionsets to turn off/on all bulbs, enter a digit '1'. If the creation is successful, users can see the below log message:

```
In execution of "./groupserver" on deviceA
1 :: CREATE ACTIONSET 2 :: EXECUTE ACTIONSET(ALLBULBON) 3 :: EXECUTE ACTIONSET(ALLBULBOFF)
4 :: GET ACTIONSET 5 :: DELETE ACTIONSET 6 :: QUIT
9 :: FIND GROUP 0 :: FIND BOOKMARK TO OBSERVE

1
ActionSet :: AllBulbOff*
ActionSet :: AllBulbOn*
```

④ Execute a group action

To execute a group actionset, users just call a executeActionSet() function in ThingsManager class as following:

~/deviceA/service/things-manager/sampleapp/linux/groupaction/groupserver.cpp 257

```
void allBulbOff()
{
    g_ thingsMgr->executeActionSet(resource, "AllBulbOff", callback)
}
```

To execute a group action set to turn off all bulbs, enter a digit '3' as follows:

```
In execution of "./groupserver" on deviceA
1 :: CREATE ACTIONSET 2 :: EXECUTE ACTIONSET(ALLBULBON) 3 :: EXECUTE ACTIONSET(ALLBULBOFF)
4 :: GET ACTIONSET 5 :: DELETE ACTIONSET 6 :: QUIT
9 :: FIND GROUP 0 :: FIND BOOKMARK TO OBSERVE

3
```

Then, users can notice that all light bulb applications have a "Off" event as follows:

```
In execution of "./lightserver" on device and deviceC
…
In entity handler wrapper:


     In Server CPP entity handler:
            requestFlag : Request
                   requestType : PUT
                          power: off
```

⑤ Request an observe to a bookmark application

For the second scenario mentioned in Section 7.1, the groupserver application needs to monitor how a bookmark's status changes. To do this, the groupserver application utilizes an observe option of CoAP specification.

To find the bookmark resource and request an observe to it, enter a digit "0" as follows:

```
In execution of "./groupserver" on deviceA
1 :: CREATE ACTIONSET 2 :: EXECUTE ACTIONSET(ALLBULBON) 3 :: EXECUTE ACTIONSET(ALLBULBOFF)
4 :: GET ACTIONSET 5 :: DELETE ACTIONSET 6 :: QUIT
9 :: FIND GROUP 0 :: FIND BOOKMARK TO OBSERVE


0
FOUND RESOURCE
OBSERVE RESULT:
     SequenceNumber: 0
     level: 0
```

If the bookmark resource is found and the request is successful, users can see a log message like the above. At the same time, users can also notice that the bookmark application has a new prompt to get users' input as follows:

```
In execution of "./bookmark" on deviceD


Input a integer(0:opened, 5:close) :
…
```

⑥ Change a status of bookmark resource

If users want to change a status of the bookmark resource, simply put a digit “0” or “5” as follows:

```
In execution of “./bookmark” on deviceD


Input a integer(0:opened, 5:close) : 0

…
```

Once the status changes, the bookmark application sends the notification to the groupserver application. If the notification informs that a book is opened, just execute an “AllBulbOn” actionset and send a request for light bulb resource to turn on.

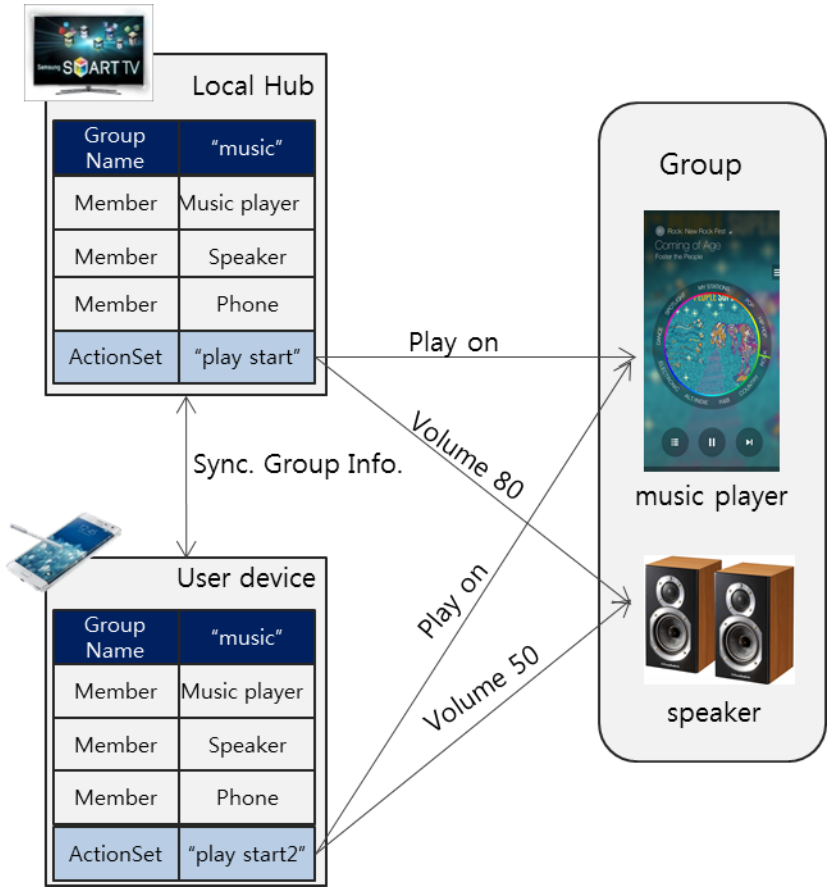
After that, users can notice that all light bulb applications have a “On” event as follows:

```
In execution of “./lightserver” on deviceB and deviceC

…

In entity handler wrapper:


     In Server CPP entity handler:

            requestFlag : Request

                   requestType : PUT

                           power: on
```

8 EXAMPLE : GROUP SYNCHRONIZATION AND GROUP ACTION

This section is to show the functionality of group synchronization and group action. With example, it can be understood how group information is shared within all members of a group, action set is made by one member of group and it is executed to control other members of a group.

8.1 EXAMPLE SCENARIO



- Scenario 1: Music Control Service

There are mainly four entities composing "Music Control System": one local hub, one user device like mobile phone, one music player and one speaker. This scenario shows that users can easily manage a music group with their mobile phone.

① At the initial stage, we assume that a music player and a speaker joined a music group in a local hub already.

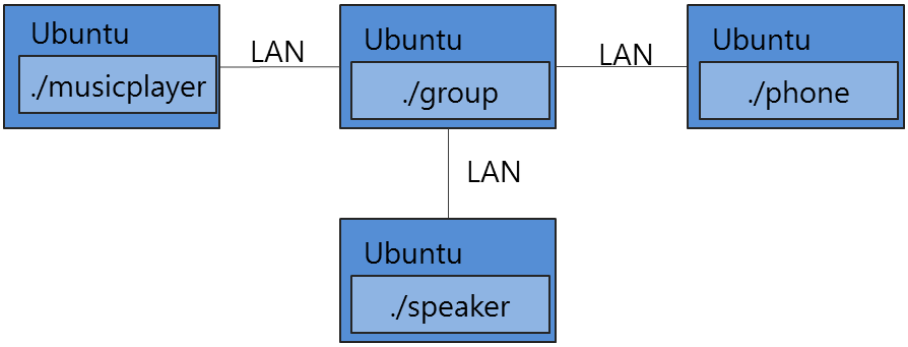
② A user device finds and joins a "music group" in the local hub and saves the members of a "music group".

③ The user device makes new ActionSet “play start2” that controls the music player and the speaker.

④ The user device adds ActionSet “play start2” to the local hub.

⑤ When the user device requests to execute ActionSet “play start2” to the local hub, the local hub sends some commands to the music player and the speaker.

8.2 WORKING FLOW



[Example Topology]

This section introduces an experimental example of the scenarios to run on Ubuntu platform. This section describes how Things (i.e. music player, speaker, group and phone) can make group and execute some ActionSet.

① Run all requisite applications composing the scenarios

First of all, users execute a bunch of applications corresponding to actuators (called resources from here) such as a music player, a speaker, a group and a phone in this example. Create the Ubuntu device topology according to “Example Topology” and run executables

For example, to execute a music player application, enter as follows:

```
~/oic-resource/service/things-manager/sampleapp/linux/groupsyncaction$./musicplayer
(1) CREATE MUSIC PLAYER | (2) DELETE MUSIC PLAYER
1
Resource creation (music player) was successful
(1) CREATE MUSIC PLAYER | (2) DELETE MUSIC PLAYER
```

Please enter digit “1” to create music player resource.

Also to execute a speaker application, enter as follows:

```
~/oic-resource/service/things-manager/sampleapp/linux/groupsyncaction$./speaker
(1) CREATE SPEAKER } (2) DELETE SPEAKER
1
To register speaker resource was successful
(1) CREATE SPEAKER } (2) DELETE SPEAKER
```

Please enter digit “1” also to create speaker resource.

Next, users execute a group. To do, enter as follows:

```
~/oic-resource/service/things-manager/sampleapp/linux/groupsyncaction$./group
(1) CREATE GROUP
(11) FIND MUSIC PLAYER & JOIN GROUP | (12) FIND SPEAKER & JOIN GROUP
(21) LEAVE GROUP – MUSIC PLAYER | (22) LEAVE GROUP – SPEAKER
(31) DELETE GROUP
1
GroupSynchronization::createGroup - The created group is added.
GroupSynchronization::createGroup : collection uri - /core/group, type - core.group
Group creation was successful
(1) CREATE GROUP
(11) FIND MUSIC PLAYER & JOIN GROUP | (12) FIND SPEAKER & JOIN GROUP
(21) LEAVE GROUP - MUSIC PLAYER  | (22) LEAVE GROUP - SPEAKER
(31) DELETE GROUP
```

Please enter digit “1” to create group.

Lastly, users execute a phone application. To run the application, enter as follows:

```
~/oic-resource/service/things-manager/sampleapp/linux/groupsyncaction$./phone
(1) CREATE PHONE
(11) FIND & JOIN GROUP | (12) ADD GROUP ACTION | (13) PLAY START | (14) PLAY STOP
(15) DELETE GROUP ACTION | (16) LEAVE GROUP
(21) DELETE PHONE
1
To register phone resource was successful
(1) CREATE PHONE
(11) FIND & JOIN GROUP | (12) ADD GROUP ACTION | (13) PLAY START | (14) PLAY STOP
(15) DELETE GROUP ACTION | (16) LEAVE GROUP
(21) DELETE PHONE
```

Please enter digit “1” to create phone resource.

② Make a music player and a speaker join a group

The first step for the scenarios is to discover a music player and a speaker in a network.

To find them enter two digit “11” and “12” in group application. Then the updated group information is displayed in group application.

```
Collection Resource Handle List
        1. collection resource type - core.group
          details
          uri - /core/group
          resource type - core.group
          resource interface - oc.mi.def

                1. child resource details
                  uri - coap://10.251.44.67:34143/core/musicplayer
                  resource type - core.musicplayer
                  resource interface - oc.mi.def

                2. child resource details
                  uri - coap://10.251.44.67:59317/core/speaker
                  resource type - core.speaker
                  resource interface - oc.mi.def
```

③ Join a group

To manage ActionSet for the members of a group phone has to join a group.

Please enter digit “11” in phone application. Then the updated group information is displayed in both group application and phone application.

```
Collection Resource Handle List
        1. collection resource type - core.group
          details
          uri - /core/group
          resource type - core.group
          resource interface - oc.mi.def

                1. child resource details
                  uri - /core/phone
                  resource type - core.phone
                  resource interface - oc.mi.def

                2. child resource details
                  uri - coap://10.251.44.67:34143/core/musicplayer
                  resource type - core.musicplayer
                  resource interface - oc.mi.def

                3. child resource details
                  uri - coap://10.251.44.67:59317/core/speaker
                  resource type - core.speaker
                  resource interface - oc.mi.def
```

④ Create a new ActionSet and add it to a group

Users can create a new ActionSet in phone and add it to a group to control a music player and a speaker by phone.

To create and add ActionSet, enter digit “12” in phone application. At that time two ActionSets are created and added. The first is “playstart” and the second is “playstop”. When “playstart” is executed, a music player starts playing and the volume of a speaker is set to 50. When “playstop” is executed, a music player stops playing and the volume of a speaker is set to 0. In group application the two ActionSet information is displayed.

```
   DESC ::
playstart*uri=coap://10.251.44.67:34143/core/musicplayer|play=on*uri=coap://10.251.44.67:59317/core/speaker|
volume=50

   DESC Copied ::
playstart*uri=coap://10.251.44.67:34143/core/musicplayer|play=on*uri=coap://10.251.44.67:59317/core/speaker|
volume=50

   ACTION SET NAME :: playstart

   ACTION SET NAME :: playstart

   ACTION SET NAME :: playstart

   ATTR Copied :: uri=coap://10.251.44.67:34143/core/musicplayer|play=on

   ATTR Copied :: uri=coap://10.251.44.67:34143/core/musicplayer|play=on

   uri :: coap://10.251.44.67:34143/core/musicplayer

   play :: on

   ATTR Copied :: uri=coap://10.251.44.67:59317/core/speaker|volume=50

   ATTR Copied :: uri=coap://10.251.44.67:59317/core/speaker|volume=50

   uri :: coap://10.251.44.67:59317/core/speaker

   volume :: 50


           RESPONSE ::

   {

           "href": "/core/group",

           "rep":  {

   }

   }

   DESC ::
playstop*uri=coap://10.251.44.67:34143/core/musicplayer|play=off*uri=coap://10.251.44.67:59317/core/speaker|
volume=0

   DESC Copied ::
playstop*uri=coap://10.251.44.67:34143/core/musicplayer|play=off*uri=coap://10.251.44.67:59317/core/speaker|
volume=0

   ACTION SET NAME :: playstop

   ACTION SET NAME :: playstop

   ACTION SET NAME :: playstop

   ATTR Copied :: uri=coap://10.251.44.67:34143/core/musicplayer|play=off

   ATTR Copied :: uri=coap://10.251.44.67:34143/core/musicplayer|play=off

   uri :: coap://10.251.44.67:34143/core/musicplayer

   play :: off

   ATTR Copied :: uri=coap://10.251.44.67:59317/core/speaker|volume=0

   ATTR Copied :: uri=coap://10.251.44.67:59317/core/speaker|volume=0

   uri :: coap://10.251.44.67:59317/core/speaker

   volume :: 0


           RESPONSE ::
```

⑤ Execute group ActionSet

The next step is to execute some ActionSet. If digit “13” is selected in phone application, “playstart” is executed. And if digit “14” is selected, “playstop” is executed.

Please enter digit “13” in phone application. Then some command is received in music player application and speaker application.

```
In entity handler wrapper:


mpEntityHandler:
        requestFlag : Request
                requestType : PUT
                        play : on
```

Music player application receives the command “play on”.

```
In entity handler wrapper:


speakerEntityHandler:
        requestFlag : Request
                requestType : PUT
                        volume : 50
```

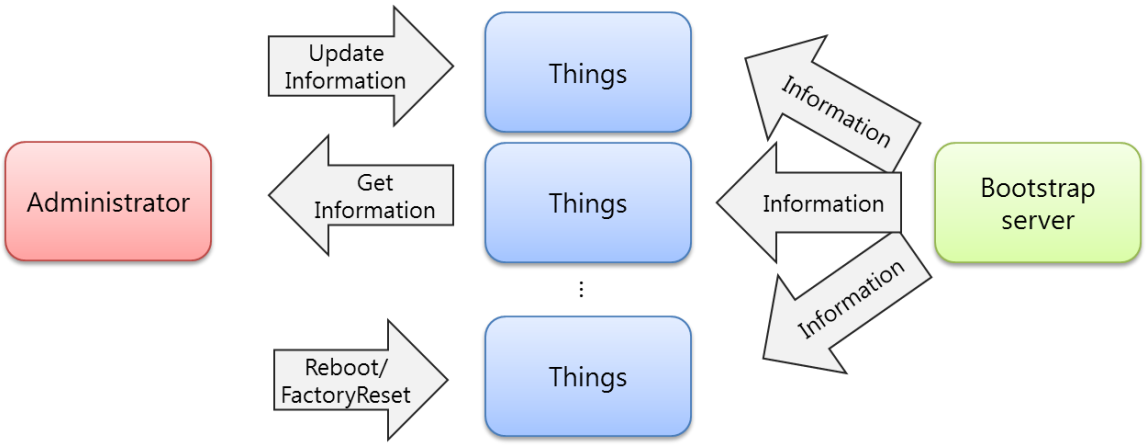
Speaker application receives the command “volume 50”.

.

9 EXAMPLE : THINGS CONFIGURATION & DIAGNOSTICS

This section is to show the functionalities of things configuration and diagnostics. As mentioned in Section 5.2, a things configuration is to get/update system configuration parameters and a things diagnostics is to ask a set of specified functions with diagnostic purpose such as a factory reset and system reboot.

9.1 EXAMPLE SCENARIO



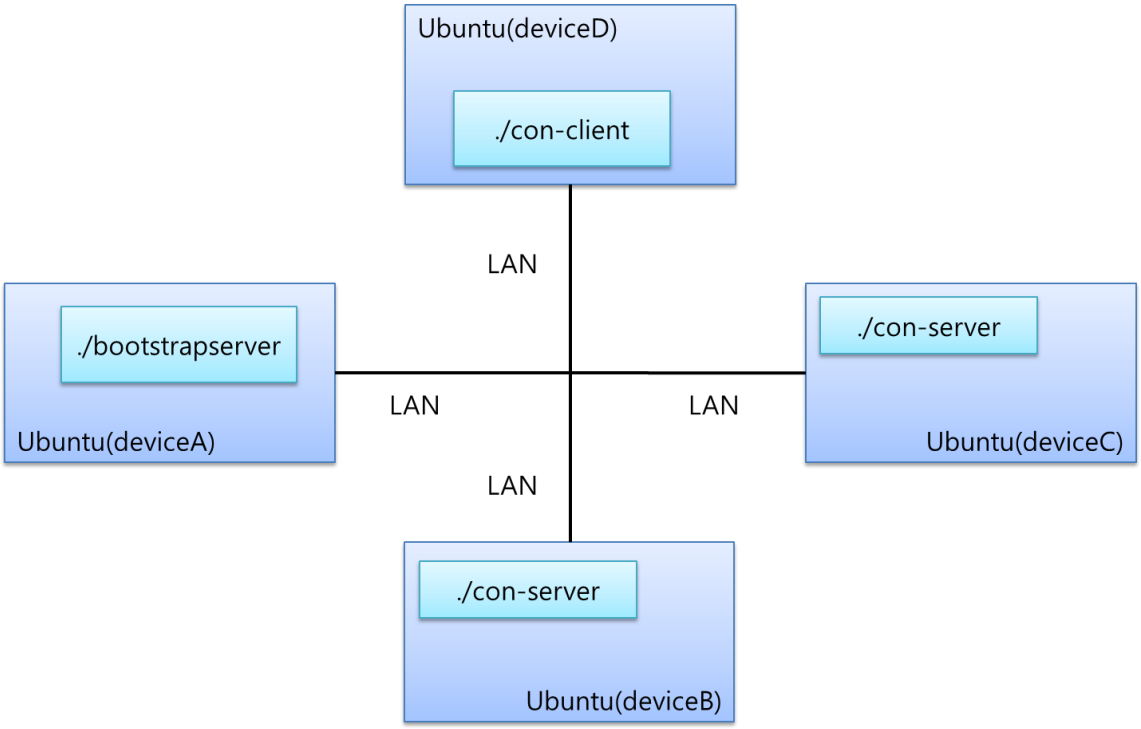
- Scenario 1: Things Configuration (Get/Update)

We assume that there are two things(i.e. resource servers) to be managed by the administrator(i.e. a resource client). Additionally, we also assume that a simple bootstrap server is installed in purpose that things can fetch essential configuration parameters to access other IoT service. This scenario shows a bootstrap procedure between things and a bootstrap server and retrieval and update procedures between things and an administrator.

- Scenario 2: Things Diagnostics (Factory Reset/Reboot)

This scenario shows two requests from an administrator to things: First is a factory reset and second is a system reboot. A factory reset is to restore all system configuration parameters to default one. And a system reboot is just to let the system reboot.

9.2 WORKING FLOW



[Example Topology]

Note that this section introduces an experimental example of the scenarios run on Ubuntu platform.

① Run all relevant applications: a bootstrapserver, two con-servers, and con-client applications.

First of all, users execute all relevant applications. A bootstrapserver application represents a bootstrap server, a con-server application does a Thing, and a con-client application does an administrator. Create the Ubuntu device topology according to “Example Topology” and run executables

To execute the applications, enter as follows:

```
~/deviceA/service/things-manager/build/linux/release$ ./bootstrapserver
…
```

```
~/deviceB/service/things-manager/build/linux/release$ ./con-server
~/deviceC/service/things-manager/build/linux/release$ ./con-server
…
(0) Quit
(1) Bootstrap
(2) Create Configuration Resources
```

```
~/deviceD/service/things-manager/build/linux/release$ ./con-client
…
(0) Quit
(1) Find all resources(URI: /oic/con, /oic/diag, /factoryset)
(2) Make collections with the found resources
(3) Get a new value (of "Configuration" Collection)
(4) Update a value (of "Region" Resource)
(5) Get a value (for "Region" Resource)
(6) FactoryReset (for the group)
(7) Reboot (for the group)
(10) Show Configuration Units
```

② Bootstrapping between things and a bootstrap server

First step for the con-server application is to bootstrap all system configuration parameters from the bootstrapserver application. To do this, users simply use a doBootStrap() function as following code:

~/deviceA/service/things-manager/sampleapp/linux/configuration/con-server.cpp 324

```
int main()
{
    …
    g_thingsmanager->doBootstrap(&onBootstrap);
```

After a successful bootstrapping, a callback function will be called. All configuration parameters are carried as a pair of attribute key and value in a form of OCRepresentation instance. So users need to use an attribute key to retrieve a corresponding attribute value. In this release, users should manually retrieve these configuration parameters. This process is shown as following code:

~/deviceA/service/things-manager/sampleapp/linux/configuration/con-server.cpp 249

```
void onBootstrap(const HeaderOptions& headerOptions, const OCRepresentation& rep, const
int eCode)

{

    if (eCode == SUCCESS_RESPONSE){

        std::cout << "\n\nGET request was successful" << std::endl;

        std::cout << "\tResource URI: " << rep.getUri() << std::endl;


        defaultRegionValue = rep.getValue < std::string > ("regionValue");

        defaultTimeValue = rep.getValue < std::string > ("timeValue");

        defaultCurrentTimeValue = rep.getValue < std::string > ("currentTimeValue");

        defaultNetworkValue = rep.getValue < std::string > ("networkValue");

        defaultIPAddressValue = rep.getValue < std::string > ("IPAddressValue");

        defaultSecurityValue = rep.getValue < std::string > ("securityValue");

        defaultModeValue = rep.getValue < std::string > ("modeValue");

        defaultConfigurationValue = rep.getValue < std::string > ("configurationValue");

        defaultFactorySetValue = rep.getValue < std::string > ("factorySetValue");
```

Based on the codes, users can execute the con-server application to bootstrap by entering a digit “1” as follows:

```
In execution of "./con-server" on deviceB and deviceC
…


(0) Quit
(1) Bootstrap
(2) Create Configuration Resources
1
Finding Bootstrap Server resource...
DISCOVERED Resource:
     URI of the resource: /bootstrap
     Host address of the resource: coap://10.251.42.143:33472
     List of resource types:
            bootstrap
     List of resource interfaces:
            oc.mi.def
Getting bootstrap server representation on: oc.mi.def


GET request was successful
     Resource URI: /bootstrap
     regionValue : Seoul, Korea
     timeValue : Time Collection
     currentTimeValue : 00:00:00
     networkValue : Network Collection
     IPAddressValue : 192.168.0.2
     securityValue : SecurityValue
     modeValue : NoSec
     configurationValue : Configuration Collection
     factorySetValue : FactorySet Value
```

③ Create a Configuration resource and Diagnostics resource

Next step for the con-server application is to create a Configuration resource and Diagnostics resource with the retrieved configuration parameters from a bootstrap server. Please note that an implmementation and management of the Configuration resource and Diagnostics resource is up to developers. For this example scenario, this guide provides sample files (i.e., ConfigurationCollection.h/.cpp and DiagnosticsCollection.h/.cpp) for the resources so users can refer the files.

To create such resources, enter a digit “2” as follows:

```
In execution of “./con-server” on deviceB and deviceC
…
(0) Quit
(1) Bootstrap
(2) Create Configuration & Diagnostics Resources
2
Configuration Collection is Created!(URI: /oic/con)
Time Collection is Created!(URI: /oic/con/time)
Network Collection is Created!(URI: /oic/con/network)
Security Collection is Created!(URI: /oic/con/security)
Configuration Collection is Created!
Diagnostics Collection is Created!
```

Note that log messages on a FactorySet resource are missing in the above. The usage of FactorySet resource will be discussed in Step 7.

④ Discover a Configuration resource and Diagnostics resource and Make a group

Next step for the con-client application is to discover a Configuration resource and Diagnostics resource in a network. As often as a resource is found, users can check its URI to be used to categorize it into a Configuration resource (with /oic/con) and Diagnostics resource (with /oic/diag). Then, using categorized resources, users can make two groups: a group for a Configuration resources and a group for a Diagnostics resources. To make a group, users can use a *bindResource*() provided by OCPlatform class.

To discover the resources and make a group, enter a digit “1” as follows:

```
In execution of "./con-client" on deviceD
…
(0) Quit
(1) Find all resources(URI: /oic/con, /oic/diag, /factoryset)
(2) Find all groups
…
1

Finding Configuration Resource...
Finding Diagnostics Resource...
DISCOVERED Resource:
     URI of the resource: /oic/diag
     Host address of the resource: coap://10.251.42.143:53773
     List of resource types:
             oic.diag
     List of resource interfaces:
             oc.mi.def
             oc.mi.b
             oc.mi.ll
DISCOVERED Resource:
     URI of the resource: /oic/con
     Host address of the resource: coap://10.251.42.143:53773
     List of resource types:
             oic.con
     List of resource interfaces:
             oc.mi.def
             oc.mi.b
             oc.mi.ll

(The other Configuration/Diagnostics Resources are found)
```

Note that a group can be represented to a collection resource. Thus, users need to find the collection resources as to find groups. To do this, enter a digit “2” as follows:

```
In execution of "./con-client" on deviceD
…
(0) Quit
(1) Find all resources(URI: /oic/con, /oic/diag, /factoryset)
(2) Find all groups
…
2

Finding Collection resource...
DISCOVERED Resource:
     URI of the resource: /core/a/diagnostics/resourceset
     Host address of the resource: coap://10.251.42.143:39071
     List of resource types:
             core.diagnostics.resourceset
     List of resource interfaces:
             oc.mi.b
             oc.mi.c
             oc.mi.def
DISCOVERED Resource:
     URI of the resource: /core/a/configuration/resourceset
     Host address of the resource: coap://10.251.42.143:39071
     List of resource types:
             core.configuration.resourceset
     List of resource interfaces:
             oc.mi.b
             oc.mi.c
             oc.mi.def
```

⑤ Update a Configuration resource

Next step for the con-client application is to update a specific value of a Configuration resource. Here, a target value is a value of Region resource which is a child resource of Configuration resource.

First, users need to know a Configuration Name indicating the target resource, which is introduced in Section 5.2.1. In this release, the Configuration Name is “region”. And users need to specify a new value to be updated. After that, users store them in form of std::map structure and then use a *updateConfigurations*() function. The code of this procedure is shown as follows:

~/deviceA/service/things-manager/sampleapp/linux/configuration/con-client.cpp 395

```
ConfigurationName name = "region";
ConfigurationValue value = "U.S.A (new region)";

std::cout << "For example, change region resource's value" << std::endl;

std::map< ConfigurationName, ConfigurationValue > configurations;
configurations.insert(std::make_pair(name, value));

if (g_thingsmanager->updateConfigurations(g_configurationCollection, configurations,
        &onUpdate) != OC_STACK_ERROR)
    isWaiting = 1;
```

Based on the above code, to update a value of Configuration resource, enter a digit “4” as follows:

```
In execution of “./con-client” on deviceD
…
 (0) Quit
(1) Find all resources(URI: /oic/con, /oic/diag, /factoryset)
(2) Make collections with the found resources
(3) Get a new value (of "Configuration" Collection)
(4) Update a value (of "Region" Resource)…
4
```

Then, users can notice that all con-server applications have received a request to update the value as follows:

```
In execution of "./con-server" on deviceB and deviceC
…
In entity handler wrapper:

     In Server CPP (entityHandlerForResource) entity handler:
     In Server CPP prepareResponseForResource:
            requestFlag : Request
                   requestType : PUT
                          value: U.S.A (new region)
```

⑥ Get a Configuration resource

Next step for the con-client application is to get a value of a Configuration resource. Here, a target value is a value of Configuration resource which has four child resources such as a Region, Time, Network, and Security resources. More details are described in Section 5.2.

To get a value, users need to know a Configuration Name indicating the target resource, which is introduced in Section 5.2. In this release, the Configuration Name is "configuration". An update value is not needed. After that, users store them in form of std::vector structure and then use a *getConfigurations*() function. The code of this procedure is shown as follows:

~/deviceA/service/things-manager/sampleapp/linux/configuration/con-client.cpp 377

```
  ConfigurationName name = "configuration";

  std::cout << "For example, get configuration collection's value" << std::endl;

  std::vector< ConfigurationName > configurations;
  configurations.push_back(name);

  if (g_thingsmanager->getConfigurations(g_configurationResource, configurations,
&onGet)
          != OC_STACK_ERROR)
      isWaiting = 1;
```

⑦ Factory Reset for diagnostics

One functionality of Thing Diagnostics is a factory reset to restore all system configuration parameters to default one. As described in Section 5.2, all default parameters are stored in FactorySet resource.

For a factory reset functionality, users need to just update a value of FactoryReset resource to “true”. After the value is updated to “true”, the con-server application checks the value and executes a factory reset by itself. The logic how to do a factor reset is not provided by Things Diagnotics. This con-server application just shows one of example to do a factory reset using with FactorySet resource. Please refer to the con-server application for this.

To sum up, users need to just update a value of FactoryReset resource by using *factoryReset*() function. The code of this procedure is shown as follows:

~/deviceA/service/things-manager/sampleapp/linux/configuration/con-client.cpp 425

```
// factory reset
if (g_thingsmanager->factoryReset(g_diagnosticsCollection, &onFactoryReset)
        != OC_STACK_ERROR)
    isWaiting = 1;
```

Based on the code, to request a factory reset to all con-server applications, enter a digit “6” as follows:

```
In execution of “./con-client” on deviceD
…

(0) Quit
...
(6) FactoryReset (for the group)
(7) Reboot (for the group)
(10) Show Configuration Units

6
```

Then, users can notice that all con-server applications have received a request to do a factory reset as follows:

```
In execution of "./con-server" on deviceB and deviceC
…
In entity handler wrapper:


     In Server CPP (entityHandlerForResource) entity handler:
     In Server CPP prepareResponseForResource:
             requestFlag : Request
                    requestType : PUT
                           value: true
Factory Reset will be soon...
```

⑧ System reboot for diagnostics

One functionality of Thing Diagnostics is a system reboot to let the system reboot. Like a factory reset, all users need to do for the system reboot is to just update a value of Reboot resource to "true". The logic how to let the system reboot is not provided by Thing Diagnostics. This con-server application just shows one of example to reboot an Unbuntu system. Please refer to the con-server application for this.

The code to request a reboot to the con-server applications is shown as follows:

~/deviceA/service/things-manager/sampleapp/linux/configuration/con-client.cpp 432

```
// reboot
if (g_thingsmanager->reboot(g_diagnosticsCollection, &onReboot) != OC_STACK_ERROR)
    isWaiting = 1;
```

Based on the code, to update a value of Reboot resource, enter a digit "7" as follows:

```
In execution of "./con-client" on deviceD
…


(0) Quit
...
(6) FactoryReset (for the group)
(7) Reboot (for the group)
(10) Show Configuration Units

7
```

Then, users can notice that all con-server applications have received a request to reboot a system as follows:

```
In execution of "./con-server" on deviceB and deviceC
…
In entity handler wrapper:


     In Server CPP (entityHandlerForResource) entity handler:
     In Server CPP prepareResponseForResource:
             requestFlag : Request
                     requestType : PUT
                             value: true
Reboot will be soon...
```

Please note that an actual system reboot will happen.