IoTivity

The **IoTivity** is an open source project.[1] The IoTivity project is hosted by the Linux Foundation,[2] and sponsored by the OIC[3] that is a group of technology companies such as Samsung Electronics and Intel who will be developing a standard specification and certification program to enable the Internet of Things.[4] This project is independent from the OIC. Any individual or company can contribute to the project, and this may influence OIC standards indirectly. However, being a member of the OIC can benefit from patent cross-licensing protection.

The IoTivity architectural goal is to create a new standard by which billions of wired and wireless devices will connect to each other and to the internet. The goal is an extensible and robust architecture that works for smart and thin devices.[5]

IoTivity 1.1.0 Features

General

- Core functionality written in C for deployment to constrained devices
- Most functionality available from C and C++
- Other bindings available:
    - Java (Android)
    - JavaScript (in progress)

Discovery & Connectivity

- Direct Device-to-Device, Local Network
- Messaging Connectivity
- Supports information exchange and control based on a messaging/CoAP Model
- Supports IPv4 and IPv6 on IP-based networks (all OS)
- Supports Bluetooth GATT profile on Linux, Android and Arduino targets
- Supports Bluetooth Serial RFCOMM (Android)
- Message switching between heterogeneous connectivity types supported
- Manages radio connections among devices (Wi-Fi*, LAN) and across any available transport, whether it's device-to-device or across the same network
- The SDK abstracts all the OS APIs for radio connections into simpler APIs
- Discovery mechanisms for devices and resources in proximity
- Allows presence subscription, un-subscription, and announcement from the device, and based on a newly created resource
- Provides device discovery mechanism to find devices based upon specific device-level attributes
- Supports secure connections (DTLS)
- Supports high QoS (RESET/ACK CoAP responses)
- On-boarding support by multi-phy EasySetup

Resource Management

- Provides platform initialization, discovery of resources and registration/creation of resources
- Resource model based operations: GET, PUT, OBSERVE, Cancel observation, and notifications sent by the resource
- Entity handler support to receive requests from a client for processing
- Client, server and client-server mode support with In-Proc model, CBOR encoding and decoding, CBOR serialization and deserialization
- Allows a root resource to point to other resources
- Default, link list and batch operations on the collection resource

- Enables support for a server to indicate a 'slow response' to a client request
- Provides mechanism for completing operations on a secure resource

Services

Resource Encapsulation API:

- Simplified data-driven API to create an IoTivity server and client

Scene Manager:

- Group creation and finding appropriate resources in network
- Member presence management (check status regarding connectivity/resource change)
- Taking a single action on a group to affect all resources
- Supports configuration and diagnostic commands

Resource Offloading (Resource Hosting Service):

- Responsible for hosting the resources of a Lite device or less power/memory capable IOT device by another smart device
- Event driven mechanism to detect the presence of devices whose resources needs to be hosted
- Targeted discovery of hosting resource
- Registering a resource as virtual in a smart device base to differentiate and give priority to the hosting resource

Resource Directory:

- Responsible for hosting a list of services not always discoverable with multicast queries (e.g., devices that disable their radios when idle)
- Client-side code to enable discovery of the RD itself and to publish services in it

Resource Container:

- Dynamic loading of plugins (resource bundles), including bundle templates and a common configuration
- Bundles can be loaded dynamically and abstract the interaction with the IoTivity base
- Virtual soft sensors: takes input from one or more resources (physical sensor data listening), adds processing, translation, and aggregation capabilities to present as a virtual resource
- Provides mechanism to represent non-OIC protocols within the OIC framework

IoTivity Services

IoTivity Services, which are built on the IoTivity base code, provide a common set of functionalities to application development. IoTivity Services are designed to provide easy, scalable access to applications and resources and are fully managed by themselves.

There are four IoTivity Services, each with its own unique functionality: Protocol Plugin Manager, Soft Sensor Manager, Things Manager, and Notification Manager.

If you want to know the necessary development environment and setup information to work with each of the services, see **IoTivity Services: Getting Started for Linux, Tizen and Android**.

Protocol Plugin Manager

Protocol Plugin Manager makes IoTivity applications communicate with non-IoTivity devices by plugging protocol converters.
It provides several reference protocol plugins and plugin manager APIs to start/stop plugins.

For more details, see **Programmer's Guide: Protocol Plugin Manager for Linux, Tizen and Android**.

Soft Sensor Manager

Soft Sensor Manager provides physical and virtual sensor data on IoTivity in a robust manner useful for application developers. It also provides a deployment and execution environment on IoTivity for higher level virtual sensors. Soft Sensor has two main components:

1. Soft Sensor Manager: A service component that 1) collects physical sensor data, 2) manipulates the collected sensing data by aggregating and fusing it based on its own composition algorithms, and 3) provides the data to applications.
2. Soft Sensor (Logical Sensor, Virtual Sensor): A software component that detects specific events or changes in a given context by applying its predefined process model with required data.

For more details, see **Programmer's Guide: Soft Sensor Manager for Linux, Tizen and Android**.

Things Manager

Things Manager creates Groups, finds appropriate member things in the network, manages member presence, and makes group action easy. It benefits 3rd party application developers in three ways:

1. Application can easily collect things for a specific service by the service characteristics, not by each thing's identification.
2. Application does not require handling, tracing, or monitoring many things.
3. Application does not require managing to send control messages to several things. Also, configuration and diagnostics of multiple things can be supported by this service.

For more details, see **Programmer's Guide: Things Manager for Linux, Tizen and Android**.

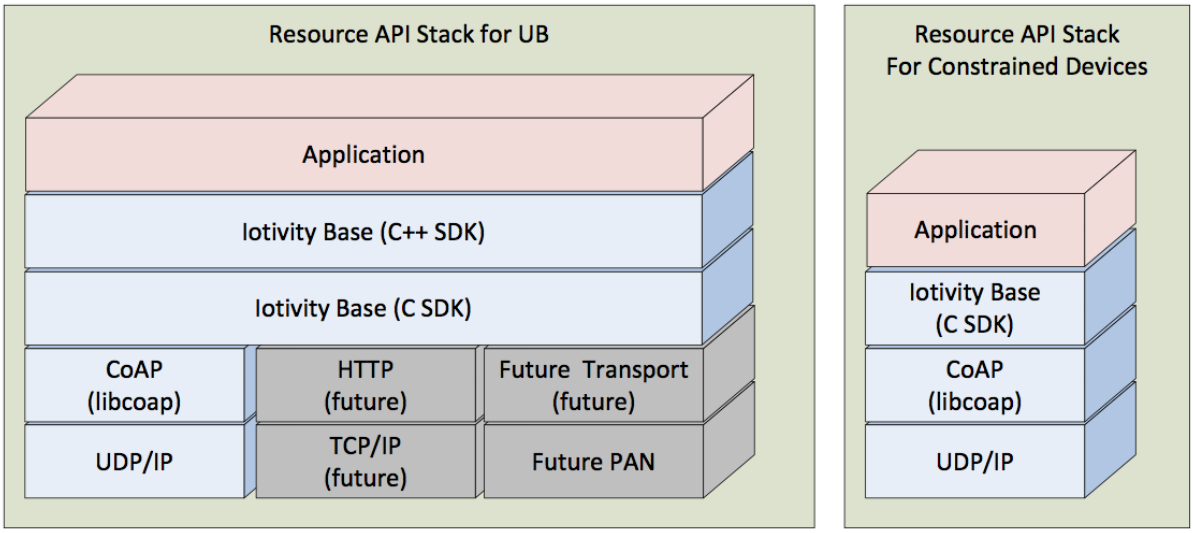
Notification Manager

Notification Manager provides resource hosting function. The resource hosting is a feature which stores only resource data with new address:port information. The goal of this feature is to off-load the request handling works from the resource server where original resource is located.

Resource API

Stack Blocks

The Resource API stack consists of several thin layers of software. In unconstrained environments such as Android*, iOS*, or Microsoft* Windows*, the stack provides APIs in C and C++ that allow developers to talk to both constrained and unconstrained devices via IP networks, with potential support for additional network protocols and wireless technologies. In the first release, the key technologies for connectivity include UDP/IP and the Constrained Application Protocol (CoAP).



Terminology

Device A constrained device that has the Thin Block stack installed which enabled one or more services for other Thin Block or Unified Block devices to consume.

Resource A resource is a component in a server that can be viewed and controlled by another Thin Block or Unified Block device. There are different resource types, for example a temperature sensor, a light controller etc.

Resources can be arranged in a hierarchal manner to form a resource tree. This generic method of structure enables modeling of many different resource topologies.

- Example: A light controller is a resource.
- Example: A light array is a set of resources organized in a flat (non-hierarchical) manner.
- Example: A garage door opener is a resource; it could host two resources - light and lock.

A more detailed description of resources and resource management along with code snippets is provided later in this document.

Operations Operations are actions that a Thin Block or Unified Block can perform on attributes associated with a particular resource. Resource attributes can have different operations based on the nature of the resource type. Fundamentally, these are GET and PUT operations. Additionally, attributes can be declared observable to enable remote devices to subscribe to changes.

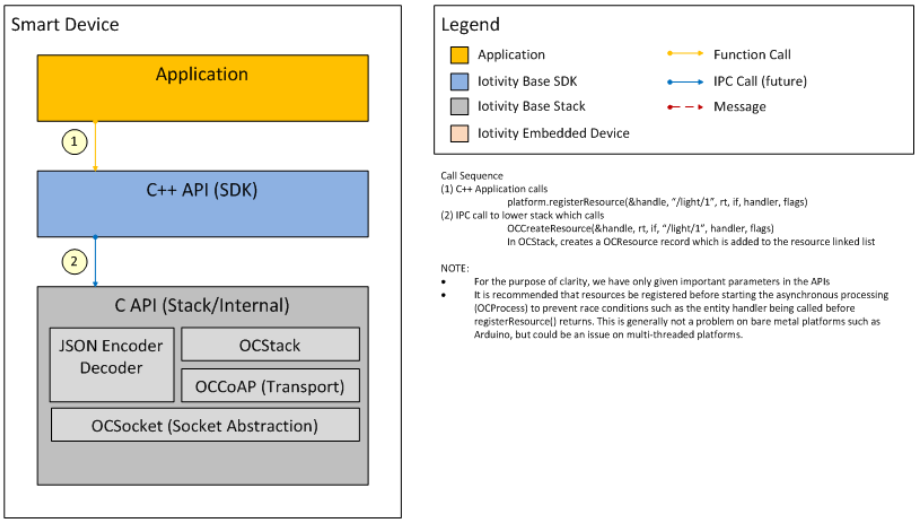
- Example: One of the child resources on the garage door opener is the light control; it has a GET operation that allows a device to get the current light state (on/off).
- **Registering a Resource**

Registering a Resource

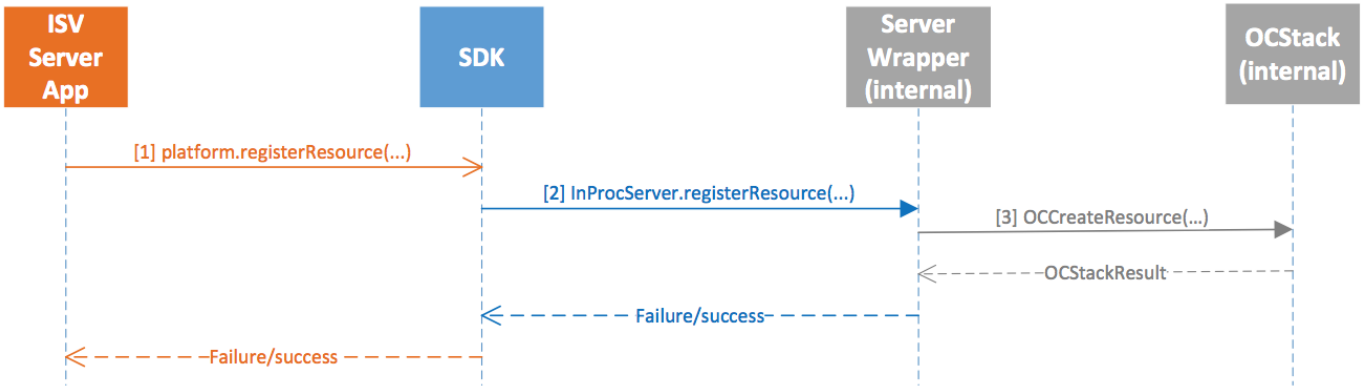
Registering a resource requires two basic items:

- A handler to process requests from the stack, and
- A URI path to register the resource. The URI path should be rooted (in other words, start with a slash). The stack will construct the fully qualified URI by adding the URI authority to the provided URI path. For example, given a service running on port 5683 in a device at IP address 192.168.1.1, if the application registers a resource with a URI path "/light/1", the resulting fully qualified URI is "oc://192.168.1.1:5683/light/1", which uniquely identifies the resource's location (IP address port and path).

  Note: Only one resource can be registered at a given URI.



Sequence Diagram
The following call sequence diagram outlines the operations performed in the stack when a resource is registered:



Step 1

Assuming the application has created a valid OCPlatform object, the application registers a new resource with the stack by calling OCPlatform::registerResource(...).

In this example, the call would take the form:

```
 platform.registerResource(&handle, "/light/1", "light", "oc.mi.def",
handler,
```

```
    OC_DISCOVERABLE);
```

The handle is a reference to the resource that is used on other APIs. The URI path ("/light/1") is where the resource can be located on this server. The URI path is unique; this call will fail if the application attempts to register another resource using an existing URI. The resource type ("light") and interface ("oc.mi.def") are properties of the resource used in the discovery process. The handler is a function called from the stack to process requests. The flags control how the stack should handle the resource. The OC_DISCOVERABLE flag indicates that the resource should be reported if a client performs a resource discovery on this server.

Step 2:

The OCPlatform::registerResource(...) method delegates the call to the appropriate instance of the stack (in-process or out-of-process via IPC).

Step 3:

The internal registerResource(...) method constructs a C++ entity handler and registers it with the C SDK using OCCreateResource(...).

In this example, the call would take the form:

```
OCCreateResource(&handle, "light", "oc.mi.def", "/light/1", handler,
    OC_DISCOVERABLE);
```

Many of these parameters are passed through to the C SDK directly. However, the entity handler is a proxy function for the handler passed from OCPlatform::registerResource(...).

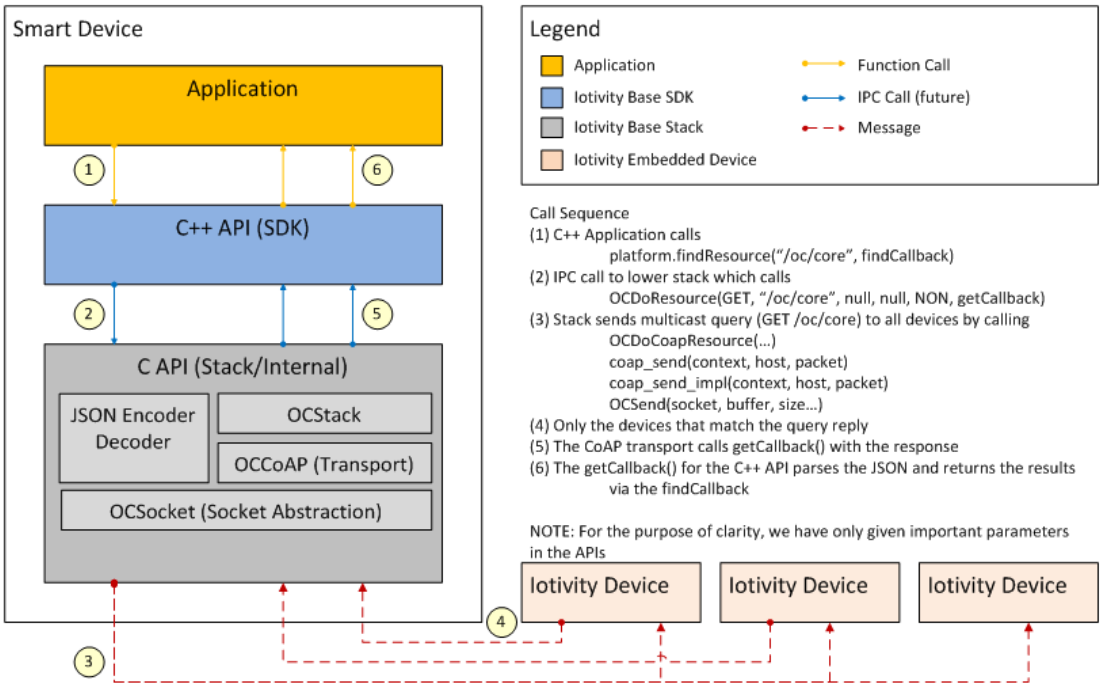
Register Resource in C++ [Server]

```
OCResourceHandle resourceHandle;
std::string resourceURI = "/light/1";
std::string resourceTypeName = "alpha.light";
std::string resourceInterface = DEFAULT_INTERFACE;
uint8_t resourceProperty = OC_DISCOVERABLE | OC_OBSERVABLE;

OCStackResult result = platform.registerResource(resourceHandle,
resourceURI,
    resourceTypeName, resourceInterface, &entityHandler, resourceProperty);
if (OC_STACK_OK == result)
{
    //Successfull
}
```

Finding a Resource

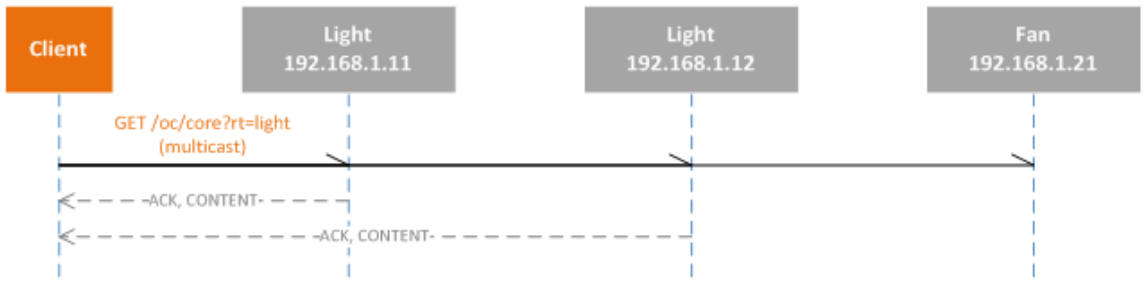
This operation returns all resources of given type on the network service. This operation is sent via multicast to all services. However, the filter limits the responders to just those that support the resource type in the query. Currently only exact matches are supported.



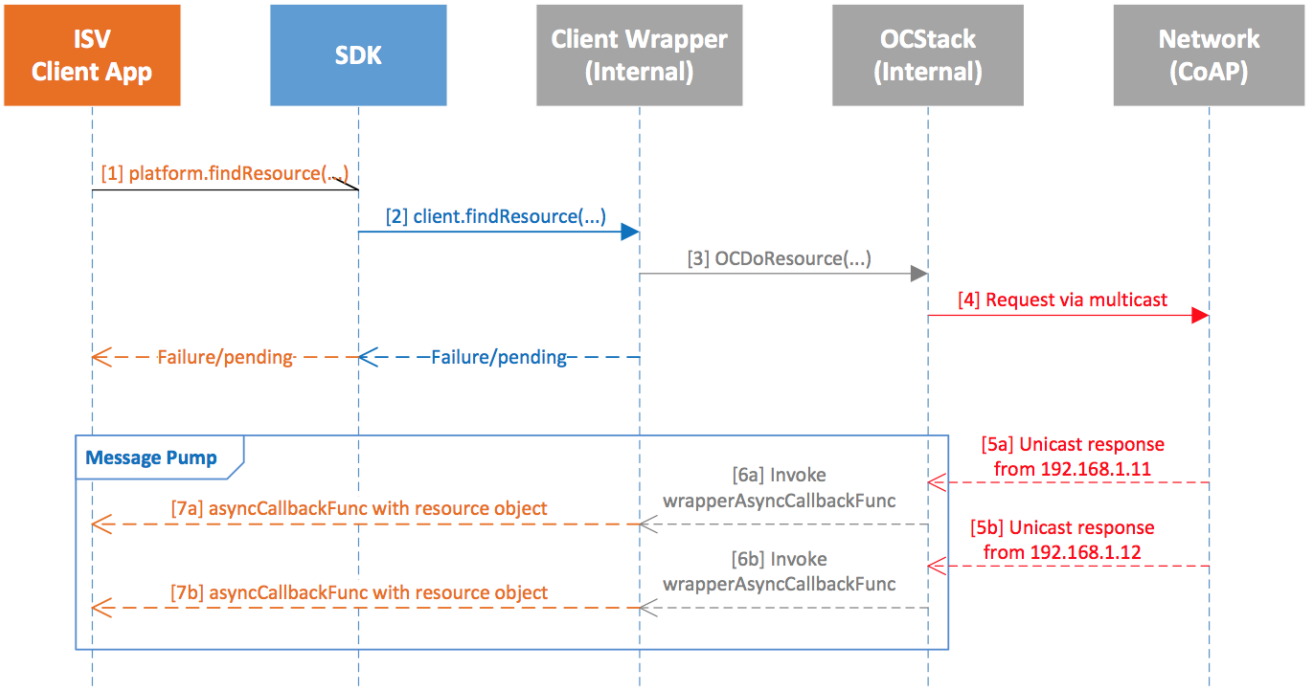
Sequence Diagram

The following sequence diagram illustrates the resource discovery process over the network when using CoAP. The mechanism is different for Bluetooth, SSDP/HTTP, etc. In the case of CoAP, a 'get' request is sent via multicast to all IoTivity devices. Each device processes the query and responds if the request filter is satisfied.

In the following example, the client requests all of the light resources with a resource type (rt). Both lights respond to the request, but the fan does not.



The following sequence diagram describes the call sequence for discovery from the client side.



Notes:

- Assuming that the client has a properly initialized OCPlatform object, a C++ SDK client can discover resources by calling OCPlatform::findResources(...).

In this example, the call would take the form:

```
platform.findResources("", "coap://224.0.1.187/oc/core?rt=alpha.light",
    findHandler)
```

The first parameter is the URI authority (target host), which, when empty, indicates that this is for all nodes. The second parameter ("/oc/core?rt=alpha.light") is the URI path and URI query. The URI path ("/oc/core") indicates the resource and the URI query ("rt=alpha.light") is the filter.

- The SDK call findResources(...) internally delegates the call directly to the in-process or to the out-of process stack via IPC based on the stack configuration.
- Within the stack, findResource(...) calls the C API function OCDoResource(...). In this example, the call is OCDoResource(&handle, OC_REST_GET, "/oc/core?rt=alpha.light", 0, 0, OC_NON_CONFIRMABLE, ...)
- OCDoResource determines which transport is needed to dispatch the request and delegates the call. In the case of CoAP, the following calls are made:
    - Calls OCDoCoapResource(OC_REST_GET, OC_NON_CONFIRMABLE, token, "/oc/core?rt=alpha.light", 0). The token in this example is a nonce that ties a CoAP response back to the CoAP request. Internally, this method creates the CoAP PDU for dispatching.
    - Calls coap_send(context, host, pdu), which is a wrapper for the implementation below.
    - Calls coap_send_impl(context, host, packet), which dispatches the packet to the socket and does the appropriate CoAP bookkeeping.
    - Calls OCSend(socket, buffer, size...), which is a wrapper for the socket implementation as the functions for dispatching a UDP packet can vary in the embedded systems.
- Servers that offer the resource on the network will reply to the query. The message pump evoked from the OCProcess(...) function in the C SDK receives these response packets and dispatches

them to the callback associated with the original request based on the CoAP message ID. These responses will come back at the timing defined by their servers. The client stack has timeouts for these responses that are listed in the appendices.

- As previously mentioned, the stack matches the response to the original request using the message ID and send the results to the callback associated with the request. At this level, the raw payload is presented in JSON format. It is the responsibility of the callback passed to OCDoResource(...) to parse this result.
- The C++ SDK provides a callback to OCDoResource(...) that will parse the results, construct collections of OCResource objects from the response, and pass them to a C++ client using the handler passed to the platform.findResource(...) method. The handler will be called once for each resource server that responds to the query.

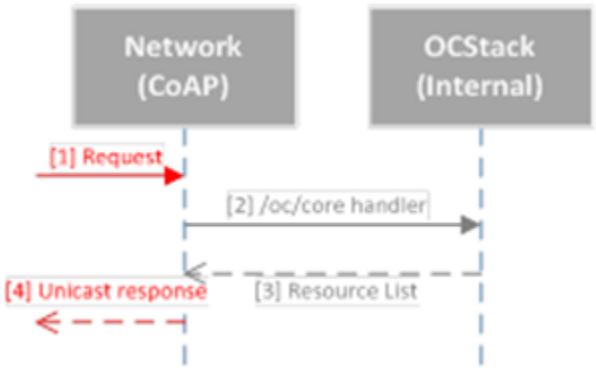
Notes:

- Some of the API call parameters have been omitted for brevity.
- The findResource() method can be used in the following ways:
  - Find all resources on the network that match the provided criteria
  - Query a specific (single) server for the resources that it provides matching the provided criteria
- The findResource() method may be used multiple times to find a resource
- The findResource() callback is called from the message pump thread in multithreaded environments
- Blocking in the findResource() callback will block other stack processing including servicing the network I/O which can cause delays and missed packets.

Detailed server call sequence diagram The following sequence diagram illustrates the call sequence for discovery from the server side.

Note

When the request is sent to all nodes, all nodes will run through this sequence.



Notes:

- The discovery request under CoAP is handled like any other resource GET request. The request can be received via unicast or multicast, but the response, if any, is always sent via unicast.
- The stack dispatches the request to an entity handler defined by the stack.
- The handler for "/oc/core", processes the URI query, if any, and builds a list of resources that match the criteria and returns the result in JSON to the network transport.

- In the case of CoAP, if the request is made to all nodes (multicast) and the resource list is empty, no response is sent to the clients. If the request is directed (unicast) or the resource list has results, the response is sent unicast back to the client.

Register Resource in C++ [Client]

```
// Callback to found resources
  void foundResource(std::shared_ptr<OCResource> resource)
  {

  std::string resourceURI;
  std::string hostAddress;
  try
  {
    // Do some operations with resource object.
    if(resource)
    {
      std::cout<<"DISCOVERED Resource:"<<std::endl;
      // Get the resource URI
      resourceURI = resource->uri();
      std::cout << "\tURI of the resource: " << resourceURI << std::endl;

      // Get the resource host address
      hostAddress = resource->host();
      std::cout << "\tHost address of the resource: " << hostAddress <<
std::endl;

      // Get the resource types
      std::cout << "\tList of resource types: " << std::endl;
      for(auto &resourceTypes : resource->getResourceTypes())
      {
        std::cout << "\t\t" << resourceTypes << std::endl;
      }

      // Get the resource interfaces
      std::cout << "\tList of resource interfaces: " << std::endl;
      for(auto &resourceInterfaces : resource->getResourceInterfaces())
      {
        std::cout << "\t\t" << resourceInterfaces << std::endl;
      }

      if(resourceURI == "/a/light1")
      {

        // Found interested resource
      }
    }
    else
    {
      // Resource is invalid
      std::cout << "Resource is invalid" << std::endl;
    }
  }
  catch(std::exception& e)
  {
    //log
  }
  }
```

```
try
{
  OCPlatform platform(cfg);

  // Find all resources
  platform.findResource("", "coap://224.0.1.187/oc/core?rt=alpha.light",
      &foundResource);

}catch(OCException& e)
{
  //Handle Error
}
```

Over the air Request

The request is sent to all nodes on the network:

Field	Value	Note(s)
Address	224.0.1.187:5683	Multicast packet
Header	NON, GET, MID=0x7d40	Multicast discovery request should be non-confirmable
URI-Path	oc	"/oc/core?rt=alpha.light"
URI-Path	core	
URI-Query	rt=alpha.light	
Accept	application/json	

Over the air Response(s)

Assuming that all of the representative devices (see Representative Devices) are online, three responses are expected. Only the three devices with light resources respond; the list of resources has been filtered to contain just the resources that match the criteria.

From 192.168.1.11:

Field	Value	Explanation
Address	192.168.1.1:5683	Client Address
Header	ACK, CONTENT, MID=0x7d40	Success w/content

Content Format	application/json	
Payload	[{ "href" : "/light/1", "rt":["alpha.light"], "if":["oc.mi.def"], "obs":1}]	

From 192.168.1.12:

Field	**Value**	**Explanation**
Address	192.168.1.1:5683	Client Address
Header	ACK, CONTENT, MID=0x7d40	Success w/content
Content Format	application/json	
Payloa	[{ "href" : "/light/2", "rt":["alpha.light"], "if":["oc.mi.def"], "obs":1}]	

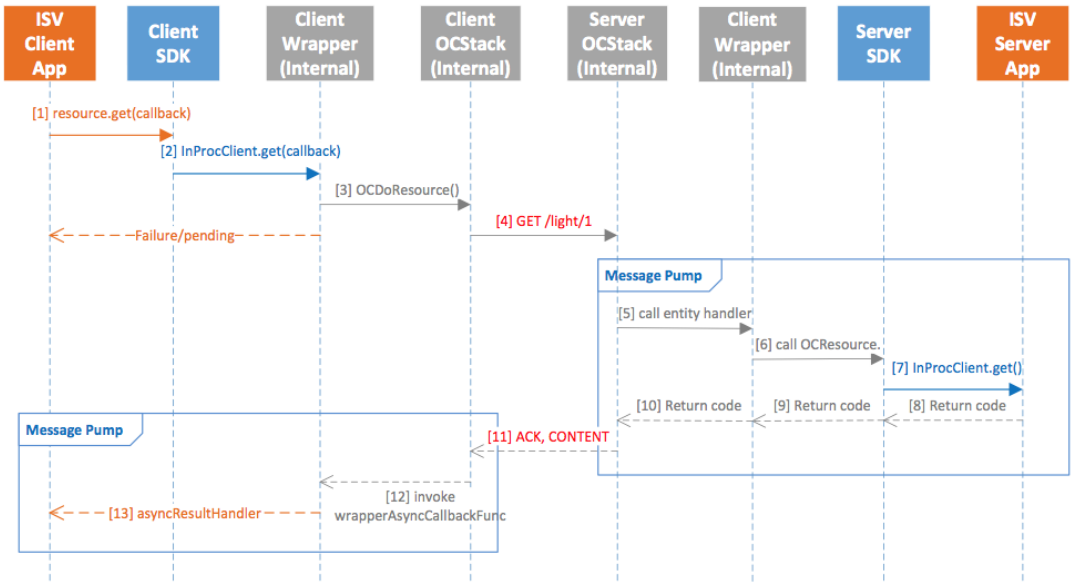
From 192.168.1.13:

Field	**Value**	**Explanation**
Address	192.168.1.1:5683	Client Address
Header	ACK, CONTENT, MID=0x7d40	Success w/content
Content Format	application/json	
Payload	[{ "href" : "/light/1", "rt":["alpha.light"], "if":["oc.mi.def"], "obs":1}, { "href" : "/light/2", "rt":["alpha.light"], "if":["oc.mi.def"], "obs":1}]	

Querying Resource State [GET]

This operation fetches the value of a simple resource. In this example, we fetch the state from the light resource.

Sequence Diagram



Steps:

1. The client application calls resource.get(...) to retrieve a representation from the resources.
2. The call is marshalled to the stack which is either running in-process or out-of-process (daemon).
3. The C API is called to dispatch the request. The call may look like the following: OCDoResource(OC_REST_GET, "//192.168.1.11/light/1, 0, 0, OC_CONFIRMABLE, callback);
4. Where CoAP is used as a transport, the lower stack will send a GET request to the target server.
5. On the server side, the OCProcess() function (message pump) receives and parses the request from the socket, then dispatches it to the correct entity handler based on the URI of the request.
6. Where the C++ API is used, the C++ entity handler parses the payload and marshals it to the client application depending on if the server stack is running in-process or out-of-process (daemon).
7. The C++ SDK passes it up the C++ handler associated with the OCResource.
8. The handler returns the result code and representation to the SDK.
9. The SDK marshals the result code and representation to the C++ entity handler.
10. The entity handler returns the result code and representation to the CoAP protocol.
11. The CoAP protocol transports the results to the client device.
12. The results are returned the OCDoResource callback.
13. The results are returned to the C++ client application's asyncResultCallback.

Querying resource State [GET] in C++ [Client]

```
// Local function to get representation of light resource
void getLightRepresentation(std::shared_ptr<OCResource>
resource)
{
```

```
  if(resource)
  {
    std::cout << "Getting Light
Representation..."<<std::endl;
    // Invoke resource's get API with the callback parameter
    QueryParamsMap test;
    resource->get(test, &onGet);
  }
}
// callback handler on GET request
void onGet(const OCRepresentation& rep, const int eCode)
{
  if(eCode == SUCCESS_RESPONSE)
  {
    std::cout << "GET request was successful" << std::endl;
    AttributeMap attributeMap = rep.getAttributeMap();
    std::cout << "Resource URI: " << rep.getUri() <<
std::endl;
    for(auto it = attributeMap.begin(); it !=
attributeMap.end(); ++it)
    {
      std::cout << "\tAttribute name: "<< it->first << "
value: ";
      for(auto valueItr = it->second.begin(); valueItr != it-
>second.end(); ++valueItr)
      {
        std::cout <<"\t"<< *valueItr << " ";
      }
      std::cout << std::endl;
    }
    std::vector<OCRepresentation> children =
rep.getChildren();
    for(auto oit = children.begin(); oit != children.end();
++oit)
    {
      std::cout << "Child Resource URI: " << oit->getUri() <<
std::endl;
      attributeMap = oit->getAttributeMap();
      for(auto it = attributeMap.begin(); it !=
attributeMap.end(); ++it)
      {
        std::cout << "\tAttribute name: "<< it->first << "
value: ";
        for(auto valueItr = it->second.begin(); valueItr !=
it->second.end(); ++valueItr)
        {
          std::cout <<"\t"<< *valueItr << " ";
```

```
        }
        std::cout << std::endl;
      }
    }
    putLightRepresentation(curResource);
  }
  else
  {
    std::cout << "onGET Response error: " << eCode <<
std::endl;
    std::exit(-1);
  }
}
```

Querying resource State [GET] in C++ [Server]

```
// Handling GET request in Entity handler
if(requestType == "GET")
  {
    cout << "\t\t\trequestType : GET\n";
    // Check for query params (if any)
    QueryParamsMap queryParamsMap = request-
>getQueryParameters();
    cout << "\t\t\tquery params: \n";
    for(QueryParamsMap::iterator it = queryParamsMap.begin();
it != queryParamsMap.end(); it++)
    {
      cout << "\t\t\t\t" << it->first << ":" << it->second <<
endl;
    }
    // Process query params and do required operations ..
    // Get the representation of this resource at this point
and send it as response
    // AttributeMap attributeMap;
    OCRepresentation rep;
    rep = myLightResource.getRepresentation();
    if(response)
    {
      // TODO Error Code
      response->setErrorCode(200);
      auto findRes = queryParamsMap.find("if");
      if(findRes != queryParamsMap.end())
      {
        response->setResourceRepresentation(rep, findRes-
>second);
      }
      else
```

```
        {
          response->setResourceRepresentation(rep,
DEFAULT_INTERFACE);
        }
      }
}
```

Over the air request

In this example, we are querying state from one of the lights. At this point, the resource was discovered by its type, and we understand its interface and the attributes that the resource exposes.

Field	Value	Note(s)
Address	192.168.1.11:5683	Unicast packet
Header	CON, GET, MID=0x7d42	Confirmation is requested
URI-Path	light	"/light/1"
URI-Path	1	
Accept	application/json	

Over the air request

Assuming that the request is valid, we expect the following reply from the resource.

From 192.168.1.11:

Field	Value	Explanation
Address	192.168.1.1:5683	Client Address

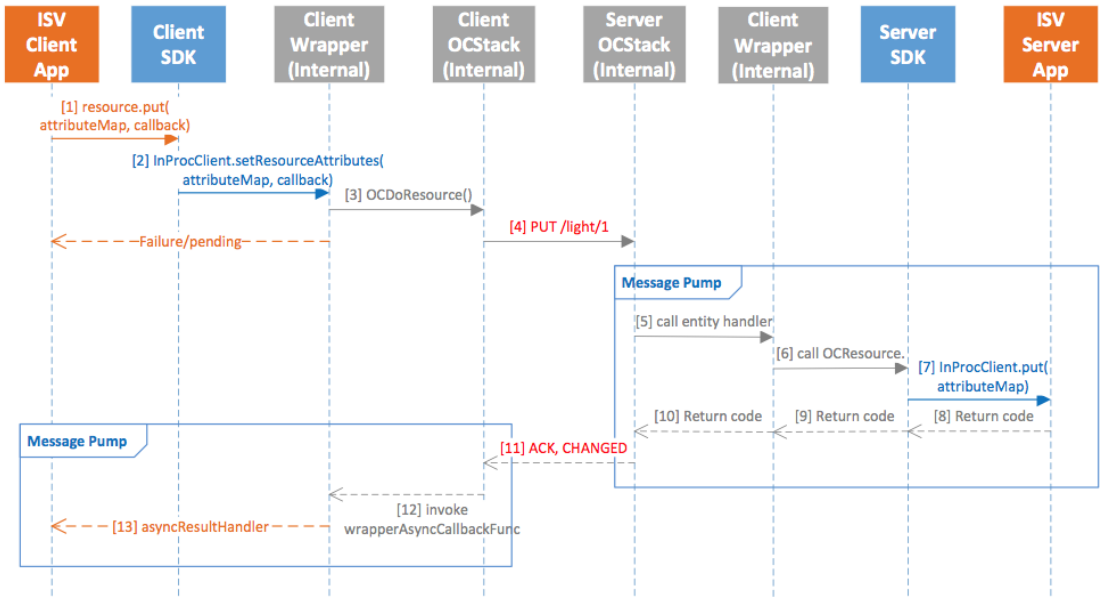
Header	ACK, CONTENT, MID=0x7d42	Success w/Content
Content Type	application/json	

Payload	<pre>{
	  "power" : 0,
	  "level" : 10
	}</pre>	

Setting a resource state [PUT]

This operation sets the value of a simple resource. In this example, we turn on a light resource and set the brightness to 50%.

Sequence Diagram



Steps:

1. The client application calls resource.put(...) to set representation of resource. Example call: resource.put(attributeMap, queryParamsMap, &onPut);
2. Client SDK internally calls the setResourceAttributes function of the client wrapper. Example call: OCDoResource(OC_REST_PUT, "//192.168.1.11/light/1, 0, 0, OC_CONFIRMABLE, callback);
3. Send PUT request to remote device
4. The OCProcess() service function (server-side message pump) reads the packet from the socket and dispatches the request to the entity handler for the provided URI.
5. The entity handler, which was provided by an upper layer when the resource was registered, parses the representation and in the case of the C++ API passes the results to the upper layer handler. In a C only environment, the results would also be processed in the entity handler.
6. The upper layer entity handler written by the app developer/vendor is invoked, and the response is formed accordingly.
7. The upper layer entity handler returns success or failure with a response.
8. Returns success or failure to lower layer for transmission to client.
9. Returns success or failure to lower layer for transmission to client.
10. Returns success or failure to lower layer for transmission to client.
11. Result is formatted and sent over network to client
12. The OCProcess() service function (client-side message pump) reads results and passes the results back to the client application via the callback passed to OCDoResource

Set Resource's State [PUT] in C++ [Client]

```
void putLightRepresentation(std::shared_ptr<OCResource>
resource)
{
  if(resource)
  {
    OCRepresentation rep;

    std::cout << "Putting light
representation..."<<std::endl;
    // Create AttributeMap
    AttributeMap attributeMap;
    // Add the attribute name and values in the attribute map
    AttributeValues stateVal;
    stateVal.push_back("true");

    AttributeValues powerVal;
    powerVal.push_back("10");

    attributeMap["state"] = stateVal;
    attributeMap["power"] = powerVal;
```

```
    // Create QueryParameters Map and add query params (if
any)
    QueryParamsMap queryParamsMap;

    rep.setAttributeMap(attributeMap);

    // Invoke resource's put API with attribute map, query
map and the callback parameter
    resource->put(rep, queryParamsMap, &onPut);
  }
}

// callback handler on PUT request
void onPut(const OCRepresentation& rep, const int eCode)
{
  if(eCode == SUCCESS_RESPONSE)
  {
    std::cout << "PUT request was successful" << std::endl;

    AttributeMap attributeMap = rep.getAttributeMap();

    for(auto it = attributeMap.begin(); it !=
attributeMap.end(); ++it)
    {
      std::cout << "\tAttribute name: "<< it->first << "
value: ";
      for(auto valueItr = it->second.begin(); valueItr != it-
>second.end(); ++valueItr)
      {
        std::cout <<"\t"<< *valueItr << " ";
      }
      std::cout << std::endl;
    }

    std::vector<OCRepresentation> children =
rep.getChildren();

    for(auto oit = children.begin(); oit != children.end();
++oit)
    {
      attributeMap = oit->getAttributeMap();

      for(auto it = attributeMap.begin(); it !=
attributeMap.end(); ++it)
      {
        std::cout << "\tAttribute name: "<< it->first << "
value: ";
```

```
        for(auto valueItr = it->second.begin(); valueItr !=
it->second.end(); ++valueItr)
        {
          std::cout <<"\t"<< *valueItr << " ";
        }
        std::cout << std::endl;
      }
    }
  }
  else
  {
    std::cout << "onPut Response error: " << eCode <<
std::endl;
std::exit(-1);
  }
```

Set Resource's State [PUT] in C++ [Server]

```
//Entity handle sample for PUT
  if(requestType == "PUT")
  {
    cout << "\t\t\trequestType : PUT\n";

    // Check for query params (if any)
    QueryParamsMap queryParamsMap = request-
>getQueryParameters();

    cout << "\t\t\tquery params: \n";
    for(auto it = queryParamsMap.begin(); it !=
queryParamsMap.end(); it++)
    {
      cout << "\t\t\t\t" << it->first << ":" << it->second <<
endl;
    }

    // Get the representation from the request
    OCRepresentation rep = request-
>getResourceRepresentation();

    myLightResource.setRepresentation(rep); // See code
snippet below

    // Do related operations related to PUT request // See
code snippet below
    rep = myLightResource.getRepresentation();

    if(response)
```

```
    {
      response->setErrorCode(200);

      auto findRes = queryParamsMap.find("if");

      if(findRes != queryParamsMap.end())
      {

        response->setResourceRepresentation(rep, findRes-
>second);
      }
      else
      {
        response->setResourceRepresentation(rep,
DEFAULT_INTERFACE);
      }
    }
  }
  void setRepresentation(OCRepresentation& light)
  {
    AttributeMap attributeMap = light.getAttributeMap();

    if(attributeMap.find("state") != attributeMap.end() &&
attributeMap.find("power") != attributeMap.end())
    {
      cout << "\t\t\t" << "Received representation: " <<
endl;
      cout << "\t\t\t\t" << "power: " <<
attributeMap["power"][0] << endl;
      cout << "\t\t\t\t" << "state: " <<
attributeMap["state"][0] << endl;

      m_state = attributeMap["state"][0].compare("true") ==
0;
      m_power= std::stoi(attributeMap["power"][0]);
    }
  }

  OCRepresentation getRepresentation()
  {
    OCRepresentation light;

    light.setUri(m_lightUri);

    std::vector<std::string> interfaces;
    //interfaces.push_back(m_lightInterface);
```

```
    light.setResourceInterfaces(interfaces);

    std::vector<std::string> types;
    //types.push_back(m_lightType);

    light.setResourceTypes(types);

    AttributeMap attributeMap;
    AttributeValues stateVal;
    if(m_state)
    {
      stateVal.push_back("true");
    }
    else
    {
      stateVal.push_back("false");
    }

    AttributeValues powerVal;
    powerVal.push_back(to_string(m_power));

    attributeMap["state"] = stateVal;
    attributeMap["power"] = powerVal;

    light.setAttributeMap(attributeMap);

    return light;
  }
```

Over the air request

In this example, we are pushing state to one of the lights. At this point, the resource was discovered by its type, and we understand its interface and the attributes exposed by the resource.

Field	Value	Note(s)
Address	192.168.1.13:5683	Unicast packet
Header	CON, PUT, MID=0x7d41	Confirmation is requested

URI-Path	light	"/light/1"
URI-Path	1	
Content-Type	application/json	

Payload	<pre>{
	  "power" : 1,
	  "level" : 5
	}</pre>	

Over the air request

Assuming that the request is valid and the resource is able to complete the transition, the following represents a successful change in state.

From 192.168.1.13:

Field	**Value**	**Explanation**
Address	192.168.1.1:5683	Client Address
Header	ACK, CHANGED, MID=0x7d41	Success (changed)

Observing resource state [Observe]

This operation fetches and registers as an observer for the value of a simple resource. In this example, we fetch the state of the light resource. For more implementation details, see "Observing Resources in CoAP" listed in the referenced documents. (https://datatracker.ietf.org/doc/draft-ietf-core-observe/?include_text=1)
The handling of observation registration is application specific. It should not be assumed that a resource is observable, or a resource can handle any specific number of observers. If the server responds with a success (2.xx) code, the registration is considered successful.

Notifications from the server to the client may be confirmable or non-confirmable. If the client returns a RST message, the observation registration should be dropped immediately. If the client fails to acknowledge a number of confirmable requests, the server should assume that the client has abandoned the observation and drop the registration.

If the observed resource is removed, the server sends a NOTFOUND status to all observers.

If an observed resource fails to notify a client before the max-age of a resource value update, the client should attempt to re-register the observation.

Observing resource state [Observe] in C++ [Client]

```
if (OBSERVE_TYPE_TO_USE == ObserveType::Observe)
  std::cout << endl << "Observe is used." << endl << endl;
  else if (OBSERVE_TYPE_TO_USE == ObserveType::ObserveAll)
    std::cout << endl << "ObserveAll is used." << endl <<
endl;

    QueryParamsMap test;

    curResource->observe(OBSERVE_TYPE_TO_USE, test,
&onObserve);

// callback
void onObserve(const OCRepresentation& rep, const int& eCode,
const int& sequenceNumber)
{
  if(eCode == SUCCESS_RESPONSE)
  {
    AttributeMap attributeMap = rep.getAttributeMap();

    std::cout << "OBSERVE RESULT:"<<std::endl;
    std::cout << "\tSequenceNumber: "<< sequenceNumber <<
endl;
    for(auto it = attributeMap.begin(); it !=
attributeMap.end(); ++it)
    {
      std::cout << "\tAttribute name: "<< it->first << "
value: ";
      for(auto valueItr = it->second.begin(); valueItr != it-
>second.end(); ++valueItr)
      {
        std::cout <<"\t"<< *valueItr << " ";
      }

      std::cout << std::endl;
    }

    if(observe_count() > 30)
    {
```

```
      std::cout<<"Cancelling Observe..."<<std::endl;
      OCStackResult result = curResource->cancelObserve();

      std::cout << "Cancel result: "<< result <<std::endl;
      sleep(10);
      std::cout << "DONE"<<std::endl;
      std::exit(0);
    }
  }
  else
  {
    std::cout << "onObserve Response error: " << eCode <<
std::endl;
    std::exit(-1);
  }
}
```

Observing resource state [Observe] in C++ [Server]

```
// Handling observe in server's entity handler
if(requestFlag == RequestHandlerFlag::ObserverFlag)
  {
    pthread_t threadId;

    cout << "\t\trequestFlag : Observer\n";
    gObservation = 1;

    static int startedThread = 0;

    // Observation happens on a different thread in
ChangeLightRepresentation function.
    // If we have not created the thread already, we will
create one here.
    if(!startedThread)
    {
      pthread_create (&threadId, NULL,
ChangeLightRepresentation, (void *)NULL);
      startedThread = 1;
    }

// ChangeLightRepresentaion is an observation function,
// which notifies any changes to the resource to stack
// via notifyObservers
void * ChangeLightRepresentation (void *param)
{
  // This function continuously monitors for the changes
  while (1)
```

```
  {
    sleep (5);

    if (gObservation)
    {
      // If under observation if there are any changes to the
light resource
      // we call notifyObservors
      //
      // For demostration we are changing the power value and
notifying.
      myLightResource.m_power += 10;

      cout << "\nPower updated to : " <<
myLightResource.m_power << endl;
      cout << "Notifying observers with resource handle: " <<
myLightResource.getHandle() << endl;

      OCStackResult result =
OCPlatform::notifyObservers(myLightResource.getHandle());

      if(OC_STACK_NO_OBSERVERS == result)
      {
        cout << "No More observers, stopping notifications"
<< endl;
        gObservation = 0;
      }
    }
  }
  return NULL;
}
```

Over the air request

The following observation request is basically a GET request with the observation option set.

Fields	Value	Notes
Address	192.168.1.11:5683	Unicast packet
Header	CON, GET, MID=0x7d44,	Confirmation requested

	TOK=0x3f	
Observe	Register (0)	This indicates registration
URI-Path	Light	"/light/1 "
URI-Path	1	
Accept	application/json	Requesting result in JSON

Over the air response(s)

A successful observe request would be similar to the following:

Field	**Value**	**Explanation**
Address	192.168.1.1:5683	Client Address
Header	ACK, CONTENT, MID=0x7d44, TOK=0x3f	Success w/content
Observe	12	Sequence number for ordering
Max-Age	30	Indicates that the value is fresh for 30 seconds. It also indicates that the server should send an update within this time period.

Content Type	application/json	

Payload	<pre>{
	  "power" : 0,
	  "level" : 10
	}</pre>	

Subsequent Notifications from 192.168.1.1

If the light resource is being observed and the light transitions from an off state to an on state, a notification is sent to the client from the server. The following is an example of such a notification:

Field	**Value**	**Explanation**
Address	192.168.1.1:5683	Client Address
Header	CON, CONTENT, MID=0x7D45, TOK=0x3f	Content, Can be confirmable or non-confirmable
Observe	15	Monotonically increasing until overflow
Max-Age	30	Indicates that the value is fresh for 30 seconds. It also indicates that the server should send an

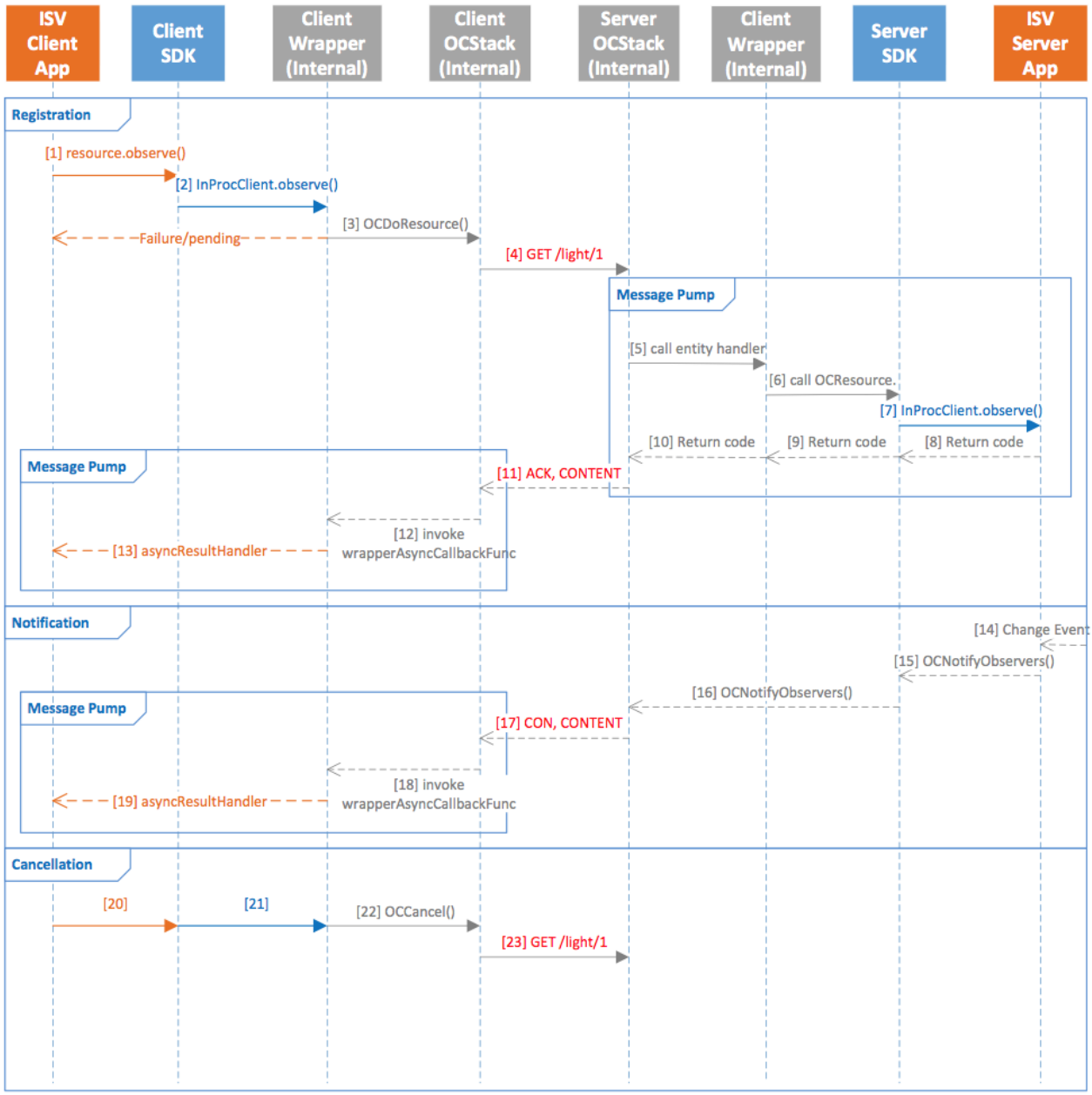
		update within this time period.

Payload	{
	"power" : 1,
	"level" : 10
	}	

Since the above notification was marked confirmable, the client should acknowledge the notification with a packet such as the following:

Field	Value	Explanation
Address	192.168.1.11:5683	Unicast packet
Header	ACK, MID=0x7D45, TOK=0x3f	Success

Sequence Diagram



Steps:

1. The client application calls resource.observe(...) to retrieve a representation from the resources.
2. The call is marshalled to the stack which is either running in-process or out-of-process (daemon).
3. The C API is called to dispatch the request. The call may look like this: OCDoResource(OC_REST_GET | OC_REST_OBSERVE, "//192.168.1.11/light/1, 0, 0, OC_CONFIRMABLE, callback);
4. Where CoAP is used as a transport, the lower stack will send a GET request to the target server. The primary difference between a GET request and an observe request is that the observe request contains an observe option indicating that, in addition to querying this resource, the client wishes to get notifications if/when the resource state changes.

5. On the server side, the OCProcess() function (message pump) receives and parses the request from the socket, then dispatches it to the correct entity handler based on the URI of the request. The request to the entity handler will indicate that the request is both a query and subscription request. The entity handler MAY take note of this, but it is not responsible to tracking the observers. The stack tracks the observers of record.
6. Where the C++ API is used the C++ entity handler parses the payload and marshals it to the client application depending on if the server stack is running in-process or out-of-process (daemon).
7. The C++ SDK passes it up the C++ handler associated with the OCResource.
8. The handler returns the result code and representation to the SDK.
9. The SDK marshals the result code and representation to the C++ entity handler.
10. The entity handler returns the result code and representation to the CoAP protocol.
11. The CoAP protocol transport the results to the client device.
12. The results are returned to the OCDoResource callback.
13. The results are returned to the C++ client application's asyncResultCallback.
14. If the entity handler has registered observers, it will periodically be called with the observe flag set so that it may sample or poll underlying hardware to determine if the state has changes.
15. When the application has deemed that the resource state has changed either via polling (entity handler observe) or via external signal, the application should call OCNotifyObservers(). This tells the stack the observers need updating.
16. For each observer of a changed resource, the entity handler is called to generate a representation that is transmitted to the observing clients.
17. Where CoAP is used as a transport, a packet with content is sent to the devices that have observing clients. The packets may be confirmable or non-confirmable based on application needs.
18. The client-side OCProcess function (message pump) receives the message and matches it to the original request based on the CoAP token ID and dispatches the appropriate C API callback.
19. The C API callback passes the final results to the C++ client application's asyncResultCallback.
20. When the C++ client no longer desires to receive notifications from the server, it calls observation cancellation method cancelObserve().
21. The C++ cancellation method calls the OCCancel() function from the C API.
22. OCCancel() finds the observation that is associated with the operation and sends an observe deregistration request to the server.