Qantom Software Pvt Ltd   Raghu Venkataramana et al.
Category: Open Proprietary Track   February 2020

IONMP : Input Output Node Messaging Protocol

Status of this Memo

This document specifies an Open standards track protocol for the IOT community, and requests discussion and suggestions for improvements. This document is work in progress and is the first version that is being circulated externally.
Distribution of this memo is unlimited.

Authors

Raghu Venkataramana
Raman K.R Iyengar

Raghu Venkataramana is the lead author of this document. He has extensive experience on working on products using the SIP Protocol. The IONMP protocol has been inspired by SIP and uses several messaging tenets of SIP.

Raman is the founder of Timing Arc, a SAP Consulting company. Prior to founding Timing Arc, he worked extensively on a SIP based enterprise unified communications system.

Copyright Notice

Abstract

This document describes the Input and Output Nodes messaging Protocol (IONMP), a messaging protocol sitting on top of Application layer protocols, designed to allow Sensors and Output Nodes to communicate with one other through a compliant server.

The protocol is agnostic of all the layers right upto the Application layer and does not stipulate or even recommend any layer for communication. Implementers are free to choose any communication protocol of their choice. The message with the format prescribed in this document MUST be communicated in the form of the payload, body or the equivalent construct of the communication protocol used. If required, implementors can use their own light weight communication protocol

The protocol allows sensor nodes and output nodes from any vendor to interoperate with one another. New devices can be plugged in after the initial deployment and relationships between nodes can be configured at run time

Table of Contents

Table of Contents(contd)

1. Introduction

In today’s scenario, for IOT, different flavours of existing protocols like HTTP, MQTT, etc., are being used. However these protocols were designed for other purposes and each implementor uses these protocols for IOT in a way that is suitable for them. The absence of a standard introduces lack of interoperability and also limits the flexibility of operation.

Entities of the IOT world have to continously communicate various types of data. The data either may be readings from sensors or may be instructions to activate output devices. In this document, the entities which are end-points are referred to as IONs - Input or Output Node.
There are numerous internet protocols that govern the communication between various entities. IONMP works in concert with these protocols by using the payload part for the messaging. Certain protocols like MQTT use a part of the payload for its own messaging. In such cases, the remaining part of the payload will be the IONMP payload.
IONMP enables the creation of an infrastructure of network hosts (called IOT gateways servers) to which IONs can send registrations and messages.

> Indented text as this paragraph will be used in this document to highlight some text.
>
> Indented text may also be used to display bullet points in an indented bullet list

2. Overview of IONMP Functionality

IONMP is a protocol that uses the message body or payload to specify the protocol rules. IONMP allows end-points (ION) to a) register with an infrastructure(gateway), b) send data to the gateway and c) receive instructions from the gateway. IONMP also provides the support for the gateway to Subscribe to events and notify subscribers about change in state.

IONMP is written, keeping in mind, the capabilities of most modern embedded computing devices. These devices are expected to have the capability to process JSON messages. The protocol has been written to allow fairly light-weight communication. This is achieved by keeping the mandatory requirements to a bare minimum. For example, IONMP specifies the flow of Registration with a secure two way authentication but does not mandate it If the gateway and the ION are in agreement, then the registration can take place even by a simple exchange of keys. It is not mandatory to have a multi step authentication.

The gateway must be acceptable to this arrangement and understand the implications of receiving messages without authentication.

3 Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 and indicate requirement levels for compliant IONMP implementations.

4 Definitions

The following terms have special significance for IONMP

Advertisement : The process of IONs propogating their capabilities and other information about themselves. The information includes Name, Location, Type of node, Input or return parameters among others.

Actuator Node : In this document, the term Actuator refers to any Output device like a fan, light, LED, Relay, etc., In its true sense, an Actuator is a device which converts electrical energy into some form of mechanical movements.

Authentication : The process where the location server verifies whether the endpoint attempting registration with the gateway has the authority to do so or not.

Controlling devices : The task of triggering an actuator node by sending a CONTROL request.

Device : An endpoint that is typically a sensor or an actuator.

Device Key : A unique key provided by the administrator (or auto generated by the infrastructure to each participating end point)

Dialog : A dialog is a set of request-response transactions required to complete a transaction between two participating ION entities. In most cases a dialog has only one pair of request-response messages. However in some cases, like two step authentication / POLLCONTROL it may be required to have multiple request-response pairs to complete a dialog

Gateway : A server or implementing the IONMP protocol

Header : The JSON Key of the IONMP Message.

Header Value : The JSON Value of a given Header for an IONMP message.

Infrastructure : Same meaning as Gateway.

ION : Abbrevation for Input or Output Node. The end point or 'Edge Node' of an IOT installation.

IOT : The Internet of Things is a system of interrelated computing devices, mechanical and digital machines are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction

JSON : Javascript Object Notation - A lightweight format for storing and transporting data.

NAT Traversal : Network Address Translation traversal is a networking tehcnique of establishing and maintaining internet protocol connections across gateways that implement NAT.

Location Server : A server entity responsible for managing registrations from endpoints

Notification : A message sent to an interested entity when an event occurs. The interested entity should have previously ‘subscribed’ with the notifying entity

Polling : The act of an Actuator Node asking the infrastructure if there is any CONTROL Message waiting for it.

Publishing : The process in which a participating sensor node sends its readings to the Infrastructure.

Registration : The process by which a compliant endpoint becomes a part of the system. The end point first sends a message stating that it wants to be a part of this system. The registrar verifies whether the registering endpoint is allowed to do so. If yes, it accepts the registration. Otherwise, rejects the registration.

Subscription : The process where an interested entity, subscribes for certain events of another entity. When the event occurs, this interested entity receives a Notification.

Registrar : Another name for Location Server

Request : The first IONMP message sent by one ION Entity to another ION Entity. This request will have the TYPE set to one of the known types like REGISTER, ADVERTISE, CONTROL, etc.

Response : The message sent by an ION Entity in response to a previously received Request. The TYPE header of a Response message will always be a 3 digit numeric value.

Sensor Node : An ION Node which is capable of measuring attributes from its sorroundings.

5. Structure of the Protocol

IONMP is structured as a protocol that sits on top of existing application layer protocols which means that its operation is independant of all the layers below it. An implementation can use any of the transport protocols, physical layer protocols or even application layer protocols. The implementing nodes should send the data in IONMP compliant form as payload of the application layer.

The protocol definitions describes the rules for different components of the system where the components are all loosely coupled, to the maximum extent possible. The protocol does not dictate an implementation in any way.

6. Overview of Operation

This section introduces the basic operations of IONMP using simple examples. This section is tutorial in nature and does not contain any normative statements. However these examples can be understood better after understanding the complete specification.

In all the examples below, a few assumptions and variables are used. They are :

The infrastructure is aware of the below mentioned items and has its credentials stored before hand.

```
   Idev1 : Input Device. This is a motion sensor. Connected to a
      IPV4 network with IP address of 10.1.1.101
   Ikey1 : Server generated key for Idev1

   Idev2 : Input Device. This is a DHT11 sensor. Connected to a
      IPV4 network with IP address of 10.1.1.102
   Ikey2 : Server generated key for idev2

   Odev1 : Output Device. This is a relay connected to a fan.
      Connected to a IPV4 network with IP address of 10.1.1.201
   Okey1 : Server generated key for Odev1

   Odev2 : Output Device. This is a buzzer. Connected to a IPV4
      network with IP address of 10.1.1.202
   Okey2 : Server generated key for Odev2

   Server is configured to set registration expires to one hour.
   Server's IPV4 address is 10.1.1.100
```

Each of the above nodes is connected to its own microcontroller.

Example1 : Idev1 wants to Register with infrastructure. Idev1 uses https and thus wants to only use the single step process

Idev1 sends a message to the gateway infrastructure with the following payload :

```
   { “Type” : “REGISTER” , “Ver” : “0.8”, “From” : “10.1.1.101”,
  “Nid” : “Idev1”, “Mid” : “Idev1001”,
  “Time” : 55343213211, “Key” : “Ikey1” }
```

Server receives this message and verifies that authentication is fine. So it sends back a 200 OK with expires of 3600

Payload of the response is :

```
{ “Type” : 200, “RespClause” : “OK”, “Ver” : “0.8”,
“Nid” : “Idev1”, “Mid” : “Idev1001”, “Time” : 55343213281,
“From” : “10.1.1.100”, “Expires” : 3600 }
```

Registration process completes successfully

Example2 : Idev2 wants to Register with infrastructure. Idev2 uses HTTP and thus wants to use the more secure 2 step authentication process.

Idev2 sends a message to the gateway infrastructure with the following payload :

```
   { “Type” : “REGISTER” , “Ver” : “0.8”, “From” : “10.1.1.102”,
   “Nid” : “Idev2”, “Mid” : “Idev2001”, “Time” : 55343213511 }
```

Server sees that there is no Key header and sends back an authentication challenge.

```
   { “Type” : 401, “RespClause” : “Provide Authentication”,
   “Ver” : “0.8”, “Nid” : “Idev2”, “Mid” : “Idev2001”,
   “Time” : 55343213534, “From” : “10.1.1.100”,
   “Nonce” : aed1295zh }
```

Idev2 receives this message and computes a Hash using its own Device key and the received Nonce

```
   md5sum of “Ikey1|aed1295zh|55343213552”
```

Note : 55343213552 is the time stamp which the device is about to send in the next response

Idev2 resends the REGISTER message this time with the above Hash included in the message.

```
   { “Type” : “REGISTER” , “Ver” : “0.8”, “From” : “10.1.1.102”,
   “Nid” : “Idev2”, “Mid” : “Idev2001”, “Time” : 55343213552,
   “Key” : “9b43bd2d24884b4781d6dc21b3a9203e” }
```

Registrar recieves this message and this time it verifies that “Key” matches the expected value. So it sends back a 200 OK

```
  { “Type” : 200, “RespClause” : “OK”, “Ver” : “0.8”,
  “Nid” : “Idev1”, “Mid” : “Idev1001”, “Time” : 55343213562,
  “From” : “10.1.1.100”, “Expires” : 3600 }
```

Registration process completes successfully

Example3 : An unknown device tries to register

Idev5 sends a message to the gateway infrastructure with the following payload :

```
{ “Type” : “REGISTER” , “Ver” : “0.8”, “From” : “10.1.1.105”,
“Nid” : “Idev5”, “Mid” : “Idev5001”, “Time” : 55343214211,
“Key” : “Ikey1” }
```

Server receives this message and finds that it doesn't know about Idev5. It sends back a message with a 403 in the payload:

```
{ “Type” : 403, “RespClause” : “Forbidden”, “Ver” : “0.8”
“Nid” : “Idev5”, “Mid” : “Idev5001”, “Time” : 55343214224,
“From” : “10.1.1.100” }
```

Registration Process Aborts

Example4 : A 'rouge' entity posing as Idev2 tries to Register with the infrastructure.

Idev2 sends a message to the gateway infrastructure with the following payload :

```
{ “Type” : “REGISTER” , “Ver” : “0.8”, “From” : “10.1.1.112”,
“Nid” : “Idev2”, “Mid” : “Idev2006”, “Time” : 55343213611,
“Key” : “WrongKey” }
```

Server sees that the value for the Key header is not what it has in its database and sends back a final 401 message

```
{ “Type” : 401, “RespClause” : “Authentication Failed”,
“Ver” : “0.8”, “Nid” : “Idev2”, “Mid” : “Idev2006”,
“Time” : 55343213623, “From” : “10.1.1.100”}
```

Registration Process Aborts

Example5 : Idev2 wants to Advertise its capabilities

Idev2 is a DHT11 sensor which is already registered. During the registration process, it had computed the hash as “0a2468f2c5c6d69621afd7bedd3744d4”

```
{ “Type” : “ADVERTISE” , “Ver” : “0.8”, “From” : “10.1.1.102”,
   “Nid” : “Idev2”, “Mid” : “Idev2002”, “Time” : 55343213711,
   “Key” : “0a2468f2c5c6d69621afd7bedd3744d4”,
   “Data” :
       [{ “Name” : “TemperatureSensor”, “NodeType” : “Sensor”,
         “Location” : “Bedroom”,
         “Capabilities” :[“Read Temperature”, “Read Humidity”],
         “Parameters” : {“isFaranheit”: “Boolean”},
         “Return” : {“Temperature” : “Float”,
            “Humidity” : “Float”, “HeatIndex” : “Float” }
       }]
}
```

The server receives this message and confirms that the Device Id and the Key sent by the client matches one of its previous registrations and so sends back a 200 OK. Payload of the response is :

```
{ “Type” : “200” , “Ver” : “0.8”, “From” : “10.1.1.100”,
   “RespClause” : “Accepted”, “Time” : 55343213721,
   “Nid” : “Idev2”, “Mid” : “Idev2002”
}
```

The Advertisement process is complete

Example5 : Idev2 has to periodically publish its sensor readings. Message flow for one such reading is as follows. Idev2 is a DHT11 sensor which is already registered. During the registration process, it had computed the hash as “0a2468f2c5c6d69621afd7bedd3744d4”. Device sends out a PUBLISH message with the following payload :

```
{  “Type” : “PUBLISH” , “Ver” : “0.8”, “From” : “10.1.1.102”,
   “Nid” : “Idev2”, “Mid” : “Idev2003”, “Time” : 55343213811,
   “Key” : “0a2468f2c5c6d69621afd7bedd3744d4”,
   “Data” : [{“Name” : “TemperatureSensor”,
      {“Temperature” : 25.4, “Humidity” : 60.5,
       “HeatIndex” : 28.3 }]
    }
}
```

Server confirms that the device has previously registered and the registration key is accurate. So it sends back a 202 Accepted

```
{ “Type” : “202” , “Ver” : “0.8”, “From” : “10.1.1.100”,
   “RespClause” : “Accepted”, “Time” : 55343213821,
   “Nid” : “Idev2”, “Mid” : “Idev2003”
}
```

This set of data has been successfully received by the server and the server will process it.

Example6 : Gateway has to control the appliance connected to Odev1

For brevity sake, let's say that Odev1 has succesfully registered with the Registrar and the computed hash is “aed123”
Let's also say that the device has already advertised its capability with the data as :

```
   Data : “Name:” : “Fan#2”, “NodeType” : “Output”,
      “Location” : “Kid's Room”, “Action” : “Switch”
      “Capabilities” :[“Fan Control”],
      “Parameters” : [“ON”, “OFF”],
      “Return” : {“State” : “Boolean”}
```

When the gateway wants to turn on the fan, it sends out a CONTROL message with the following payload.

```
{  “Type” : “CONTROL” , “Ver” : “0.8”, “From” : “10.1.1.100”,
   “Nid” : “server”, “Mid” : “server004”, “Time” : 55343213911,
   “Key” : “aed123”,  “Data” : [{ “Name” : “Fan2”,
   “Action” : “Switch”, “Parameter” : “ON”}]
}
```

When the device gets this message, it would turn on the Fan. It will then send out a 202 Accepted with the following payload:

```
{ “Type” : “202” , “Ver” : “0.8”, “From” : “10.1.1.108”,
   “RespClause” : “Accepted”, “Time” : 55343213921,
   “Nid” : “server”, “Mid” : “server004”
}
```

This completes the CONTROL operation of turning on the Room Fan.

Example7 : The Event Processor wants to SUBSCRIBE with the Message Processor and receive Notifications about an event Payload of the message :

```
{  “Type” : “SUBSCRIBE” , “Ver” : “0.8”, “From” : “10.1.1.100”,
   “Nid” : “EventProc”, “Mid” : “EventProc001”,
   “Time” : 55343214011,“Key” : “EventProc123”, “Expires” : 3600,
   “Events” : { “Nid” : “Idev1”, “Name” : “Light Sensor#1”,
   “Parameter” : “LightLevel”, “Condition” : “<”,
   “CondValue” : 3}
}
```

In the above SUBSCRIBE, the Event Processor component of the server is trying to subscribe to changes from the Light sensor when the light level is less than 3

In response to this the Message processor component (recepient of subscribe message),sends back a 202 payload as under :

```
{  “Type” : “202” , “RespClause” : “Accepted”, “Ver” : “0.8”,
   “From” : “10.1.1.100”, “Nid” : “EventProc”,
   “Mid” : “EventProc001”, “Time” : 55343214021,
}
```

Let’s say that at some point in time, the Light Sensor in question sends its sensor readings which is 2.

```
{  “Type” : “PUBLISH” , “Ver” : “0.8”, “From” : “10.1.1.101”,
   “Nid” : “Idev1”, “Mid” : “Idev1004”, “Time” : 55343264021,
   “Key” : “0a2468f2c5c6d69621afd7bedd3744d4”,
   “Data” : [{“Name” : “Light Sensor#1”,
      {“LightLevel” : 2 }]
    }
}
```

As the above value matches the requirements of the subscription, the PUBLISHEE (Message processor component), sends out a NOTIFY to the subscriber:

```
{  “Type” : “NOTIFY” , “Ver” : “0.8”, “From” : “10.1.1.100”,
   “Nid” : “MsgProc”, “Mid” : “MsgProc_Idev1004”,
   “Time” : 55343214031,“Key” : “MsgProc123”,
   “Data” : { “LightLevel”: 2 , “SubscribeId” : “EventProc001”}
}
```

In lieu of the “Nid” parameter nested within the “Data” key, a combination of the Device Name and Location can be used.

In response to this the Subscriber component sends back a 202 payload as under :

```
{  “Type” : “202” , “RespClause” : “Accepted”, “Ver” : “0.8”,
   “From” : “10.1.1.100”, “Nid” : “MsgProc”,
   “Mid” : “MsgProc_Idev1004”, “Time” : 55343214041 }
```

Example 8 : An Actuator node which is not in the same network as the server, has to be controlled based on an Event. As the node is behind NAT(Network Address Translation), the server cannot directly send the control message to the node. Instead the node sends a request to the server and asks the server whether there are any control messages pending.

For brevity sake, let's say that Odev1 has succesfully registered with the Registrar and the computed hash is “aed123”
Let's also say that the device has already advertised its capability with the data as :

```
Data : “Name:” : “Fan#2”, “NodeType” : “Output”,
       “Location” : “Room”, “Action” : “Switch”
       “Capabilities” :[“Fan Control”],
       “Parameters” : [“ON”, “OFF”],
       “Return” : {“State” : “Boolean”}
```

Case1 : An Event has actually occured due to which Fan#2 has to be turned on.
The Device first sends out a POLLCONTROL message to the server:

```
{ "Type":"POLLCONTROL", "Ver":"0.8", "From":"10.1.1.201",
  "Time":55343214021,  "Nid":"Odev1", "Key":"aed123",
  "Mid":"Odev1POLLCONTROL177" }
```

In Response to this message, the server sends the response:

```
{ "Type": "CONTROL", "Mid": "Odev1POLLCONTROL177",
  "Time": 55343214025, "From": "10.1.1.100", "Ver": "0.8",
  "Nid": "Odev1",  "Data": [ {"Name": "Fan#2",
  "Parameters": {"State": true}, "Action": "Switch"}] }
```

Based on this responses, the Fan#2 knows that it has to turn itself ON.

The control message was succesfully sent to the FAN to ask it to switch ON. This status has to be communicated back to the server. So the end point sends the message

```
{  “Type” : “CTLRESPONSE”, "Mid": "Odev1POLLCONTROL177",
   "Time": 55343214425, "From": "10.1.1.201", "Ver": "0.8",
   “Nid” : “Odev1”, “Status” : “202” }
```

In this case, the status has been set to “202” because the device did not try to check if the actual actuator turned or not. If it had checked for this and found that the actuator did work, then the status would have been set to 200.

On receipt of this message, the server sends back a 202 Accepted message back to the device.

Case2 : The Event which would have caused Fan#2 to be controlled has not occured. Inspite of this, the Fan#2 device sends a POLLCONTROL message

The Device first sends out a POLLCONTROL message to the server:

```
  {  "Type":"POLLCONTROL", "Ver":"0.8", "From":"10.1.1.201",
     "Time":55343218021,  "Nid":"Odev1", "Key":"aed123",
     "Mid":"Odev1POLLCONTROL178" }
```

This time the server checks and finds that there is no control message for the entity. It sends back a 404 response

```
{ "Type": 404, "Mid": "Odev1POLLCONTROL178", "Time": 55343218323,
  "From": "10.1.1.100", "Ver": "0.8", "Nid": "Odev1"}
```

From this response, the device understands that there is no CONTROL message for it and it does not do anything.

7 IONMP Messages

IONMP is a text-based protocol and uses the UTF-8 charset (RFC 2279). IONMP does not define the rules for the establishment of communication between the participants. Neither does it recommend any of the existing protocols as the preferred choice. It relies on the belief that the existing protocols for these layers of communication are already mature and there is no need to reinvent another protocol for this. IONMP only requires the underlying protocol to have the capability to communicate json data in the form of payload.

The payload data, which forms the protocol compliant message is a JSON text. The JSON message format takes advantage of the key-value notation specificed by JSON to seperate the header key and header value.

IONMP messages follow the request-response communication model. An IONMP message is either a request from a client to a server, or a response from a server to a client. IONMP chooses the request response model, in order to faciliate implementors to have a wider variety of choices for the underlying communication protocols. For example, HTTP, a widely used messaging protocol, use the request-response model. By adopting this model, implementors can build an IONMP compliant server by using an existing communication implementation. They are not forced to re-write a completely new server implementation.

IONMP is inspired by SIP and derives several features and philosophy from SIP.

However IONMP is not an extension or derivative of SIP.

7.1 Header keys and Values

IONMP messages are JSON messages which are sent in the form of the message payload. The JSON keys and values denotes various header keys and values.

Keys in IONMP are case-insensitive, while Values are case-sensitive.

```
“Name” : “Raghu”
is the same as
“name” : “Raghu”

“Name” : “Raghu”
is NOT the same as
“Name” : “raghu”
```

7.2 Requests

A message is a Request if it is the first message being sent out in a Request-Response dialog.

The Json key-value syntax for a request is :

“Type” : MSGTYPE

MSGTYPE has one of the following values :

REGISTER – for registering IONs with infrastructure

ADVERTISE – When endpoints want to advertise their capability

PUBLISH – When an entity wants to publish some data. The data is typically a sensor reading.

CONTROL – When an entitiy wants to send control information to ION. It is usually sent when the ION is an output device like an actuator.

SUBSCRIBE – When an IONMP entity wants to subscribe to messages that are of interest to it. This is typically done by one of the IONMP servers to communicate with other servers For example, when the Event Processor component wants to be notified about changes in a sensor value, it would SUBSCRIBE to this sensor with the Message Processor

NOTIFY – When an IONMP entity wants to notify its subscribers about messages which they are subscribed to. This is typically done by one of the IONMP servers to communicate with other servers. For example, if the Event Processor has subscribed for temperature values, whenever the temperature sensor sends data, the Message Processor will send a NOTIFY message to the Event Processor

QUERY – When an entity wants to find out about the capabilities of another node, it sends a QUERY message to the infrastructure with the TargetId field set to the DeviceId of the other node. The infrastructure then verifies if the requesting entity is authorized to make a Query. If yes, then it sends back the capabilities of the TargetId. If TargetId is specified as an asterix(*), then the infrastructure returns the capabilities of all registered entities.

POLLCONTROL – When an entity behind NAT has to receive CONTROL messages in response to other events, it sends out a POLLCONTROL message. If there is a control message waiting for this entity, the server sends back a response with the Type field set to CONTROL. If there is no CONTROL message waiting for this entity, then the server sends back a 404.

CTLRESPONSE - The CTLRESPONSE method is used by the output node to communicate the status of the previous CONTROL message. A CTLRESPONSE message MUST be sent only after an output node receives a CONTROL message in response to its previous POLLCONTROL message. It SHALL NOT be sent in any other circumstances

The entity making an IONMP request MUST send a message in which one of the JSON keys is 'Type' and the value is one of the above.

7.3 Responses

Every Response message in IONMP will have the JSON 'Type' key value pair. In most cases, the value is a 3 digit number. However it can also be a String. In this version, the only case where the Type header can be a string is when the server sends back a CONTROL message in response to a POLLCONTROL request. In ths case the Type header is set to “CONTROL”.

The Json key-value syntax for a response is :

```
   “Type” : 3digitNumber
       or
   “Type” : “CONTROL”
```

The server MAY optionally send a textual Reason clause to provide a human readable textual message for this response. The Json key-value for the reason clause is :

```
   “RespClause” : “A textual description about the response”
```

A server MUST NOT send the RespClause header if the message being sent out is NOT a response.

8 Common Header Keys

Irrespective of whether the message is a Request or a Response, all messages must contain values for certain keys. These keys and their corresponding values are described in the next sections.

8.1 Message Type (Type)

Every IONMP entity MUST send the message type in all messages. The JSON Syntax is

```
   “Type” : “Type val”
```

The “Type val” MUST be one of the types described in Sections 7.2 and 7.3

8.2 Version (Ver)

Ever IONMP entity MUST send the IONMP version in all messages. The JSON Syntax is

```
   “Ver” : “Version string”
```

For this version the string is “0.8”

8.3 Target (Target)

An IONMP entity MAY optionally send a Target URL. The “Target” parameter is used when the message is sent to one entity but the actual recepient of the message is another entity. The JSON Syntax is

```
      “Target” : “URL of Target”
```

This is an optional header and is particulary relevant for responses to QUERY requests.

8.4 Sender Address (From)

An IONMP entity MUST send its own address, either as an IP Address or a Fully Qualified Domain Name(FQDN) in all messages. The JSON Syntax is

```
      “From” : “Sender's ip address”
```

8.5 Node ID (Nid)

Every IONMP entity MUST send its unique ID in all messages. The JSON Syntax for NodeID is

```
      “Nid” : “unique id”
```

8.6 Message ID (Mid)

An IONMP entity MUST send a Message ID in all its messages. The value of the Message ID takes a different value depending on the type of message and will be discussed in the appropriate sections. The JSON Syntax for Message ID is

```
      “Mid” : “message id”
```

8.7 Time (Time)

An IONMP entity MUST send its current time in all its messages. All the participants in an IONMP system MUST have a common view of time. The protocol doesn't specify what mechanism must be used for Time synchronization. It just mandates that the times of all the entities in the system has to be the same. JSON Syntax is :

```
       “Time” : current_time
```

9 Registrations

9.1 Overview

IONMP offers a discovery capability. If an event on one ION has to trigger an action on another ION, then there must be a mechanism through which the devices can be discoverable. To allow this IONMP gateway elements have an abstract service known as location service, which provides an address binding for the participating nodes and end-points. Registration creates bindings in a location service that associates a Device ID with a contact address.

IONMP provides a mechanism for an end-point like ION to create a binding explicitly. This mechanism is known as registration.

Registration entails sending a REGISTER request to a special type of Gateway service known as a registrar. A registrar acts as the front end to the location service for a domain, reading and writing mappings based on the contents of REGISTER requests.
This location service is then consulted by services that need to access the ION.

The term Registrar and Location Server are used interchangeably in this document to refer to the entity which is responsible for managing registrations.

9.2 Authorized Entities

The Infrastructure entity should be configured to maintain a list of authorized devices. Each device must also be assigned a unique key.

9.3 Constructing a REGISTER Request

Registration of an ION may either be done in a single step process or a two step process.

In the single step process, the entity which wishes to REGISTER with the Registrar sends out a REGISTER request along with its Device ID and the server assigned Key. If the Key matches the value stored in the Registrar, the server accepts the registration and sends back a 200 OK. Otherwise the registrar sends back a 401 Authentication Failed

In a double step process, the entity which wants to REGISTER with the gateway first sends out a REGISTER request ONLY with its device id. The registering entity, does not send the KEY header. The gateway sends back a “Provide Authentication” message along with a unique nonce value. The entity sends back the REGISTER request, this time with its key value hashed with a combination of the nonce value and the value of the “Time” header which the entity will be sending back in its response.

The gateway verifies whether the received Hash matches the Hash value based on the key stored in its database and nonce. If yes, then it accepts this registration and sends back a 200 OK. If it fails, it sends back a 401 Authentication Failed response.

After successful registration, when any endpoint wants to communicate with the infrastructure using any of the Request messages, it MUST include the “Key” header. The value of the “Key” header is either the device key itself or the computed Hash depending on how the registration was done.

The double step process is extremely. In this case, all communication between the entities is authenticated using a key which is unique for every message. As each message is hashed using a different secret (end point’s time + nonce), even if there is man in the middle attack, it becomes virtually impossible to crack the code.

End-points that choose to implement single step authentication process must be aware of the security implications. They may still opt to use this, if they are confident that other measures have been put in place to ensure that security is not compromised (Example : If the communication between the endpoint and server uses TLS)

9.3.1 Flow of REGISTER Message – 1 step process

An ION or any other entity that wants to REGISTER with the gateway must set its Type key to REGISTER.
“Type” : “REGISTER”

An ION or any other entity that wants to REGISTER with the gateway creates a JSON string which includes all the mandatory common keys, “Ver”, “From”, “Nid”, “Mid”, “Time”
It MAY send the optional keys based on the configuration
The value for each of the key should be as per the description in Section 8

“Mid” MUST contain a value which is a unique string across the entire installation. The protocol does not specify or recommend any technique to generate this unique sequence. It is left upto the choice of the implementer.

The registering entity MUST define a header called “Key” and set its value to the Unique key given to it by the administrator

The JSON string in the payload message for a REGISTER request will have the following syntax:

```
    {“Type” : “REGISTER” , “Ver” : “0.8”, “From” : “Device_Address”,
     “Nid” : “DEVICE_ID”, “Mid” : “A_UNIQUE_VALUE”,
     “Time” : node's_time, “Key” : “DEVICE_KEY” }
```

In response to this request, the gateway checks its credentials database to verify if the device id exists its database. It also checks if the key sent by the device matches the key stored in the credentials database. Based on this check it does one of the following :

Case1 : Authentication matches :
The Registrar MUST send back a response with the following headers:

```
   “Type” : 200
   “Mid” : Same ID sent by the request
   “Nid” : Same Device ID sent by request
   “Ver” : Version String - currently 0.8
   “Time” : Registrar's current time
   “From” : Address of registrar
   “Expires” : Time in seconds for which registration is valid
```

The Registrar MUST add an Expires key which gives the amount of seconds for which the registration is valid. If the end point wishes to still be a part of the arrangement even after this period, it must send back another REGISTER request before the expiry period.

The Registrar SHOULD send back a Reason clause for this response using the header key “RespClause”. The RespClause could be any String which the Registrar chooses but a common reason clause for 200 is “OK”. The Registrar is free to ignore the “RespClause” header if it has a valid reason to do so . A sample reason class string would be

```
   “RespClause” : “OK”
```

Case2 : Device is not in Registrar's database:
The Registrar MUST send back a response with the following headers:

```
   “Type” : 403
   “Mid” : Same ID sent by the request
   “Nid” : Same Device ID sent by request
   “Ver” : Version String - currently 0.8
   “Time” : Registrar's current time
   “From” : Address of registrar
```

The Registrar SHOULD send back a Reason clause for the failure using the header key “RespClause”. The RespClause could be any String which the Registrar chooses but a common reason clause for 403 is “Forbidden”. The Registrar is free to ignore this header if it has a valid reason to do so .

A sample reason class string would be

```
   “RespClause” : “Forbidden”
```

When an entity receives a 403 response it should not attempt to REGISTER with the server again till the entity knows that it has the right credentials.

Case3 : Device is found in Registrar's database but key mismatch:

The Registrar MUST send back a response with the following headers:

```
   “Type” : 401
   “Mid” : Same ID sent by the request
   “Nid” : Same Device ID sent by request
   “Ver” : Version String - currently 0.8
   “Time” : Registrar's current time
   “From” : Address of registrar
```

The Registrar SHOULD send back a Reason clause for the failure using the header key “RespClause”. The RespClause could be any String which the Registrar chooses but a common reason clause for 401 is “Authentication Failed”. The Registrar is free to ignore this header if it has a valid reason to do so .

A sample reason class string would be
   “RespClause” : “Authentication Failed”

9.3.2 Flow of REGISTER Message - 2 step process

An ION or any other entity that wants to REGISTER with the gateway must set its Type key to REGISTER.
   “Type” : “REGISTER”

An ION or any other entity that wants to REGISTER with the gateway creates a JSON string which includes all the mandatory common keys,
   “Ver”, “From”, “Nid”, “Mid”, “Time”
For the two step secure registration, the gateway MUST NOT send the “Key” header.

“Mid” MUST contain a value which is a unique string across the entire installation. The protocol does not specify or recommend any technique to generate this unique sequence. It is left upto the choice of the implementer.

The Registering entity MAY optionally choose to include the “Target” Header if its scenario so requires

In response to this request, the gateway checks its credentials database to verify if the device is in its database.

Case 1 : Device not found in the credentials database
If the credentials database does not contain any reference to this device, then the registrar sends back a 403 Forbidden message in exactly the same way as the response for the similar scenario described in Section 9.3.1 - Case 2. The syntax of the message for the response is exactly similar to Case2 of Section 9.3.1.

Case 2 : Device Found in the credentials database

If the Registrar finds that the registering entity is present in its database, it then sends a response challenging the device to provide authentication. This is done by sending a message to the entity with certain headers.

Case2 - Sequence 1 :

The Registrar MUST send back a response with the following headers:

“Type” : 401
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String - currently 0.8
“Time” : Registrar's current time
“From” : Address of registrar
“Nonce” : A cryptic string/number which is generated by the Registrar and will only be used for this transaction

The Nonce MUST be a long string. This Nonce becomes one part of the 'Hasher' for the registring entity. The end-point will create a hash using a combination of its own device key, this Nonce and the value of the “Time” header that it sends with the following response. This mechanism makes the communication very secure . Even if a hacker gets hold of the Nonce or the computed hash, using a man in the middle attack, (s)he will not be able to construct the original key. Furthermore, each dialog will have a different value for the KEY header and thus (s)he will not even be able to impersonate further messages.

The Registrar SHOULD send back a Reason clause for the failure using the header key “RespClause”. The RespClause could be any String which the Registrar chooses but a common reason clause for 401 is “Authentication Failed”. The Registrar MAY also use this header to send a Reason Clause which indicates the exact type of authorization mismatch. For example, for the current REGISTER message, it could send the reason clause as “Provide Authentication” The Registrar is free to ignore the “RespClause” header if it has a valid reason to do so .

A sample reason class string would be
“RespClause” : “Provide Authentication”

Case2 - Sequence 2 :

When the requesting entity receives the above response from the server, it MUST generate a md5sum hash using the following string:

Device_Key|Nonce|Time
Device_Key : Is the Registring unique key that in the Registrar's database
| : The Pipe (vertical bar) keystroke
Nonce : Is the string sent by the Registrar in the response
Time : Is the value of the “Time” header which the registering entity sends back in its response.

The Registring entity MUST then send a followup REGISTER request with the following headers :

“Type” : “REGISTER”
“Ver” : “Version String”
“From” : “Device_Address”,
“Nid” : “DEVICE_ID”
“Mid” : “The same message ID which was in the original request”
“Time” : node's_time
“Key” : “Hash generated in previous step”

If the initial message had any optional headers, the entity MUST include those headers with the same values as in the initial message

When the Registrar receives this message, it compares the hash value sent by the Registering entity and the message-id. If the message id of the request matches the initial REGISTER request and if the hash matches the value computed by the registrar, it sends out a 200 OK. Otherwise it sends out a 401 message. The detailed message flow is described in Scenario1 and Scenario2 below

Case2 Scenario1 – when authenticated succesfully:

If the authentication is succesful, then the Registrar MUST send back a response with the following headers:

“Type” : 200
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String – currently 0.8
“Time” : Registrar's current time
“From” : Address of registrar
“Expires” : Time in seconds for which registration is valid

The Registrar MUST add an Expires key which gives the amount of seconds for which the registration is valid. If the end point wishes to still be a part of the arrangement even after this period, it must send back another REGISTER request before the expiry period.
The Registrar MAY send back a Reason clause for the response using the header key “RespClause”. The RespClause could be any String which the Registrar chooses but a common reason clause for 200 is “OK”. The Registrar is free to ignore this header if it has a valid reason to do so . A sample reason class string would be

“RespClause” : “OK”

The client which receives this response will have to start using these information for all future messaging till the registration 'expires'. In every subsequent message, it MUST send the “Key” header with the same value as it did in the authenticated REGISTER request.

Case2 Scenario2 – when authentication fails:
When the client sends a REQUEST in response to the first 401 message, the Registrar compares the Key with its own computed hash. If it doesn't match, it means that the authentication details which the client has is wrong or that an unauthorized device is trying to register.
The Registrar MUST send back a response with the following headers:

“Type” : 401
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String – currently 0.8
“Time” : Registrar's current time
“From” : Address of registrar

The Registrar SHOULD send back a Reason clause for the failure using the header key “RespClause”. The RespClause could be any String which the Registrar chooses but a common reason clause for 401 is “Authentication failed”. The Registrar MAY also use this header to send a Reason Clause which indicates the exact type of authentication mismatch.
The Registrar is free to ignore this header if it has a valid reason to do so .
A sample reason class string would be

“RespClause” : “Authentication Failed”

When a client recieves this response it MUST NOT re-attempt registrations until it is sure that the issue has been resolved.

10 Advertisement

10.1 Overview

IONMP offers a way in which endpoints can advertize their capabilities to the infrastructure. This feature allows new devices to be added after deployment, existing devices to be reconfigured and devices to be moved around without having to modify the internal working code.

Only REGISTERED devices of a network will be allowed to Advertise their capabilities.

10.2 When to send Advertise Messages

The End points MUST send an ADVERTISE message on these conditions

a) Immediately afer getting a 200 OK for Registration
b) After every succesful RE-REGISTER (after expires) Request
c) If some capability of the node has changed

10.3 Constructing an Advertisement Message

The entity which wishes to ADVERTISE its capability with the gateway sends out an ADVERTISE request along along with its Device ID and the key that was computed during the Registration process. The key may either be the actual key given to it by the administrator or a hashed value. This depends on whether the registration was done using the single step process or the two step process. If the Registration was done using a single step process (Section 9.3.1), then the Key is nothing else but the device key issued by the administrator. If the Registration was done using a two step authentication method (Section 9.3.2), then the key must be set to the Hash value computed in the two step authentication process.

The entity which wishes to ADVERTISE its capability MUST send out a request message to the gateway with the following keys

```
   “Type” : “ADVERTISE”
   “Ver” : “0.8”
   “From” : “Device_Address”
   “Nid” : “DEVICE_ID”
   “Mid” : “The same Mid of Register”
   “Time” : node's_time
   “Key” : “A Key depending on how the registration was done”
   “Data” : Explained below
   “ControlMethod” : “Poll|Deamon”
```

Data : In addition to the regular headers, the client MUST also send a Header whose key is “Data” and value is a JSON array which explains the operations that this ION can do. The Name header for each item of this array MUST be unique. The value will be a list of items that contains different headers as explained below

The JSON representation is :

```
    [
          { “Name” :“NodeName1” ,
            “NodeType” : “Sensor|Output|Hybrid”,
            “Location” : “Device’s location string”,
            “Action” : Only for Output or Hybrid type nodes,
            “Capabilities” :[Array of capabilities],
            “Parameters” : JSON|Array,
            “Return” : [JSON]
          },
          { “Name” : “NodeName2”,  ... },
             ...
         { “Name” : “NodeName()n”,  ... }
     ]
```

NodeType : Mandatory. A string which can either be Sensor, Output or Hybrid

Location : Mandatory. String representing location

Action : Mandatory for output nodes. Not there for sensor nodes.

Capabilities : An array of String representing the capabilities which this device has. For eg - Temperature, Humidity, etc.,

Parameters : Either JSON or Array.

If JSON, then key of each item is the name of the parameter while the value is a string denoting the type of the parameter (string, int,float, etc.,)

If Array, then it is an array of strings which specifies the possible values that can be passed as the only parameter.

Return : A JSON Object of return values. The key of each item is a label of what the return value signifies (eg., Temperature) while the value is the type of the value.

ControlMethod :

The endpoint MAY send the ControlMethod header. The value of this can either be Poll, Deamon or Auto.

This parameter is only applicable for output device. It has no meaning for sensors / other input devices.

If the endpoint is an output device, then it MUST send this header.

If this value is DEAMON, it means that whenever this output device has to be controlled, the infrastructure sends out a CONTROL message to the device. The endpoint SHALL listen for CONTROL messages on the default port

If set to POLL, then the infrastructure will only send the CONTROL message to the device, when the device itself sends out a POLLCONTROL message.

If set to AUTO, the gateway is free to choose whichever method it finds appropriate. By setting this value to AUTO, the endpoint stipulates that it is capable of receiving both CONTROL messages and will also be periodically send POLLCONTROL messages. By setting it to AUTO, endpoints give the server a free hand to determine whether the server and the endpoint are within the same network or they are on different networks.

By advertising these values, an output node opens itself up to being controlled at run time. Whenever an Event occurs either from a sensor node or from other extraneous changes in the server parameters, this output node can be controlled.

Likewise, if a sensor is continously streaming data, the gateway can analyze the data based on the advertised capabilities. The gateway does not have to know about the return values or format of each sensor before hand.

In a conventional system, these values must have already been known to other entities or to the gateway. In such systems, all nodes of the system are tightly coupled with each other.

When the Server receives the Advertisement request from the entity it returns a 200 OK message. The Server MUST send back a response with the following headers:

“Type” : 200
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String - currently 0.8
“Time” : Registrar's current time
“From” : Address of the server

The Server MAY send back a Reason clause using the header key “RespClause”. The RespClause could be any String which the Server chooses but a common reason clause for 200 is “OK”.

The server remembers these advertised capabilities as long as the registration has not expired. The advertised capabilities automatically expire when the registration expires.

11 PUBLISH

11.1 Overview

The PUBLISH method in IONMP allows entities of the system to PUBLISH its data to the network. In most cases, the entities which use the PUBLISH method are Sensor Nodes.
Only REGISTERED devices of a network will be allowed to PUBLISH data.

11.2 When to PUBLISH data

The Protocol does not specify the interval at which entities publish their data. Each end-point can decide for itself how frequently it wants to PUBLISH information.

11.3 Constructing a PUBLISH Message

The entity which wishes to PUBLISH its capability with the gateway sends out a PUBLISH request along with its Device ID and the key that was computed during the Registration process. If the Registration was done without hash authentication (Section 9.3.1), then the Key is nothing else but the device key issued by the infrastructure. If the Registration was done using a two step authentication method (Section 9.3.2), then the key must be set to the hash value computed in the two step authentication process.
The entity which wishes to PUBLISH its data MUST send out a request message to the gateway with the following keys

“Type” : “PUBLISH”
“Ver” : “0.8”
“From” : “Device_Address”
“Nid” : “DEVICE_ID”
“Mid” : “A unique ID for each set of data being transmitted”
“Time” : node's_time
“Key” : “A Key depending on how the registration was done”
“Data” : JSON Array as described below

```
“Data” : Array of one or more JSON objects with the following
   syntax :
   {  “Name” : The name which this ION had advertised itself as
            (Eg. “NodeName1”, “NodeName2”, etc.,
      One or more return value key-pair
      “<name of return value>” : value  ,
      “<name of return value>” : value  ,
      (one or more such return values)

      For example, if the DHT sensor is trying to send its
      temperature and humidity, then it sends back:
         "Data" : [{ "Name" : "TemperatureSensor",
         "Temperature" : 29.4, "Humidity" : 60.5 }]
```

In most the cases, for each device, we would only have one NodeName and for the “Return” key, there will be only JSON Object. This could also be an array of JSON Objects with the same schema as above.
The protocol has provision for multiple NodeName objects to allow the case where we have one microcontroller connected to multiple sensors. An alternate way to collect data from all the sensors is for each one of these sensors to be treated as a seperate entity with its unique device ID / Key. Each of these will publish the data as though it were a device by itself. By taking this approach we would be adding to the network traffic significantly.
Instead, the protocol has provision for multiple Return values to faciliate sensors which read multiple values. A classic example of this is the DHT11 sensor. It can sense the ambient temperature and the ambient temperature.

After the gateway successfully receives a SUBSCRIBE message, it MUST return a 202 Accepted Message with the following headers

```
   “Type” : 202
   “Mid” : Same ID sent by the request
   “Nid” : Same Device ID sent by request
   “Ver” : Version String - currently 0.8
   “Time” : Registrar's current time
   “From” : Address of Gateway
```

The Server SHOULD send back a Reason clause using the header key “RespClause”. The RespClause could be any String which the Server chooses but a common reason clause for 202 is “Accepted”.

When a client receives a 202 Message, it MUST NOT send back the same data again. Like wise if the Server ever receives multiple PUBLISH messages with the same Mid, it MUST only consider the first request it receives. All other messages with same Mid should be ignored.

The Data used in the ADVERTISE and PUBLISH messages can be best understood with a simple example.

Let's say a DHT11 wants to Advertise its capabilities and then continously publish the temperature and humidity values to the gateway. For brevity sake, consider that the registration has been completed using a simple single step Registration as explained in Section 9.3.1

Payload of the Advertisement request would be :

```
   {  “Type” : “ADVERTISE” , “Ver” : “0.8”,
      “From” : “Device_Address”, “Nid” : “DEVICE_ID”,
      “Mid” : “The same Mid of Register”, “Time” : node's_time,
      “Key” : “Device_key”,
      “Data” : { “Name”: “DHTSensor1”, “NodeType” : “Sensor”,
         “Location” : “Kitchen”,
         “Capabilities” :[“Read Temperature”, “Read Humidity”]
         “Return” : {“Temperature” : “Float”,
            “Humidity” : “Float” }
             }
         }
    }
```

Subsequently when the sensor wants to send its reading to the Gateway, it would send out the PUBLISH request by sending its payload as below:

```
   {  “Type” : “PUBLISH” , “Ver” : “0.8”, “From” : “Device_Address”,
      “Nid” : “DEVICE_ID”, “Mid” : “A_Unique_ID”,
      “Time” : node's_time, “Key” : “Device_key”,
      “Data” : [{
         “Name” : “DHTSensor” ,
         “Values : {“Temperature” : 28.5, “Humidity” : 60.5 }
      }]
   }
```

12 CONTROL-ing Output devices

12.1 Overview

The CONTROL method in IONMP is used to control IONs which are output devices. Typically these requests originate from the gateway. In most cases, the entities which receive the CONTROL method are Output Nodes.

Only REGISTERED devices of a network will be allowed to receive CONTROL messages.

12.2 CONTROL messages to sensors

Although the CONTROL message is meant for output devices, it can also be used to send messages to Sensor nodes. Although rare, this is usualy done when it is possible to send certain parameters to the sensor. For eg., if the sensor can accept a message using which the units of measurement can be configured.

12.3 Constructing a CONTROL Message

When an entity (typically the infrastructure) wishes to control another entity, the sender sends out a CONTROL request along with its Device ID and the key that was computed during the Registration process. The CONTROL method can only be sent to those devices which had advertised its ControlMethod as Deamon. The value of the “KEY” header depends on whether the registration was done using a single step process or a two step hash process. The entity which wishes to CONTROL an ION MUST send out a request message to the ION with the following keys

```
“Type” : “CONTROL”
“Ver” : “0.8”
“From” : “Device_Address”
“Nid” : “DEVICE_ID”
“Mid” : “A unique ID for the control message ”
“Time” : node's_time
“Key” : “A Key depending on how the registration was done”
“Data” : JSON Object as described below
```

```
“Data” : One or more JSON arrays. Each JSON object has the
   following key: value syntax:
   “Name” : Name advertised previously (eg. NodeName1, NodeName2,
   ... etc.) as described in Section 10.3.
   “Action” : A string representing the control command being sent.
   “Parameters” : Either null, a String or a JSON Name-value object
      The JSON object for parameters will be as under:
      “Param1Label” : param1Value, “Param2Label” , param2Value etc.,
      Where “param1”: The name of the 'named' parameter
      param1Value : The actual value for parameter param1 and so on.
```

The label and values correspond to the value part of the “Parameters” key which the node had sent when it Advertised its capabilities as described in Section 10.3.

Note that the Location is not required to be sent to the device. Location is something that other entities need to know about. However as far as device itself is concerned, the name is enough.

In most cases, for each device, we would only have one NodeName. The protocol has provision for multiple NodeName objects to allow the case where we have one microcontroller connected to multiple output devices. An alternate way to CONTROL all devices is by treating each device as a seperate entity with its own unique device ID/Key. We could then send individual CONTROL message to each of these. By taking this approach we would be adding to the network traffic.

When an ION receives the CONTROL message and understands it, it MUST send a 202 Accepted message. The Header keys for this 202 are :

“Type” : 202
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String - currently 0.8
“Time” : ION's current time
“From” : Address of ION
“Return” : JSON object denoting the return value

The “Return” parameter is only valid if the Actuator returns a value when the CONTROL method is called. In most of the cases, the actuator will not return anything and in such cases this header key is not required.

The ION SHOULD send back a Reason clause using the header key “RespClause”. The RespClause could be any String which the Server chooses but a common reason clause for 202 is “Accepted”.

The value for the “Return” key is a JSON object which works in the same way as was described in Section 11.3 The only difference is that for PUBLISH message, the sensor node voluntarily sends out the request and the return value is part of the request message; However in case of a CONTROL request, the receiving entity processes the CONTROL message and sends back the resulting return values in the 202 Accepted message

When a gateway receives a 202 Message, it MUST NOT send back CONTROL messages for the same commands again. Like wise if the ION receives multiple CONTROL messages with the same “Mid”, it MUST ignore duplicate messages.

Case where the entity support the recieved parameters:
When an ION receives the CONTROL message, but does not support the attributes sent in the Data parameter, then the device must send back a 405 Not Implemented response.

“Type” : 405
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String - currently 0.8
“Time” : ION's current time
“From” : Device_Address
“RespClause” : (Optional) Textual representation of the failure. Typical clause is “Not Implemented”

The Data used in the ADVERTISE and CONTROL messages can be best understood with simple examples.
Let's say a controller attached to a Relay wants to Advertise its capabilities.

Subsequently a gateway wants to control this Relay.
For brevity sake, consider that the registration has
been completed using a simple single step Registration as explained
in Section 9.3.1

Payload of the Advertisement request would be :

```
{  “Type” : “ADVERTISE” , “Ver” : “0.8”,
   “From” : “Device_Address”,“Nid” : “DEVICE_ID”,
   “Mid” : “The same Mid of Register”,
   “Time” : node's_time, “Key” : “Device_key”,
   “Data” : [{ “Name” : “Light”,  “NodeType” : “Output”,
      “Location” : “Room”, “Action” : “Switch”
      “Capabilities” :[“Control Light”],
      “Parameters” : {“State”, “Boolean”}
    }]
 }
```

Case 1: When the gateway sends the correct CONTROL message to the
endpoint:

When the gateway wants to switch ON the light
connected to the ION, it would send out the CONTROL request by
sending its payload as below:

```
{  “Type” : “CONTROL” , “Ver” : “0.8”, “From” : “server address”,
   “Nid” : “DEVICE_ID”, “Mid” : “A_Unique_ID”,
   “Time” : node's_time, “Key” : “Device_key”,
   “Data” : { “Name” : “RoomLight”, “Action” : “Switch”,
      “Parameters” : {“State” : true}}
}
```

Since all values in the DATA header are as per the advertisement,
the device understands that this message is meant for it.
It thus sends back a message to the gateway with the 202 Response.
Payload for this is :

```
{ “Type” : 202 , “Mid” : “Same ID sent by the request”,
   “Nid” : “Same Device ID sent by request”, “Ver” : “0.8”,
   “Time” : ION's current time, “From” : “Device_Address”,
   “RespClause” : “Accepted”
}
```

Case 2: When the gateway sends a CONTROL message with wrong
paramters to the endpoint:
The gateway wants to switch ON the light connected to the ION,
However it sends an incorrect CONTROL Message :
sending its payload as below:

```
{  “Type” : “CONTROL” , “Ver” : “0.8”, “From” : “Device_Address”,
   “Nid” : “DEVICE_ID”, “Mid” : “A_Unique_ID”,
   “Time” : node's_time, “Key” : “Device_key”,
   “Data” : { “Name” : “RoomLight”, “Action” : “Switch”,
      “Parameters” : {“State” : “TRUE” } }
}
```

Observe that in this case, the gateway sends the value of “TRUE” as a String instead of sending out a boolean value. In this case, the Actuator does not recognize this command and hence sends back a 405 Not Implemented response.

```
 {   “Type” : 405, “Mid” : “Same ID sent by the request”,
     “Nid” : :Same Device ID sent by request”, “Ver” : “0.8”,
     “Time” : ION's current time, “From” : “Device_Address”,
     “RespClause” : “Not Implemented”
  }
```

13 SUBSCRIBE

13.1 Overview

The SUBSCRIBE method in IONMP allows entity to subscribe to the specific events. This method is intended for communication between the server components. However any entity MAY use the specification if it is suitable for it.

13.2 Construction of a SUBSCRIBE method:

When any entity in the system is interested in keeping track of an event, it MAY SUBSCRIBE to the events generated or proxied by another entity. The entity that wishes to keep track of events SUBSCRIBEs with the other entity that is aware of these events. Whenever these events are generated, the latter NOTIFIES the subscriber.

The entity which wishes to SUBSCRIBE MUST send out a request message to the NOTIFYing entity with the following keys

```
“Type” : “SUBSCRIBE”
“Ver” : “0.8”
“From” : “Server_Address”
“Nid” : “The Subscriber's ID”
“Mid” : “A unique ID for the subscribe message ”
“Time” : node's_time
“Key” : “Subscriber's Key”
```

Note : Just like endpoints, each component of the server will be assigned a unique ID. For example, the server may have one sub-component to handle Registrations, while it has one sub-component to handle Subscriptions, and so on. Each sub-component will have its own unique ID. Further, it is possible to have each sub-component running on a different machine. In such a case, even the address will be different for each subcomponent.

In addition to the regular headers, there are two additional headers in the SUBSCRIBE message. They are :

Expires : This is the amount of time in seconds for which the subscription is valid. The subscriber will only receive notifications till this time. After the end of this period it is the job of the subscriber to refresh its subscription

Events : This is a JSON Object.
The format of the JSON Object is as follows:

```
“Events” : {
   “Name” : “<Name of device>”,
   “Location” : “<device location>”,
   “Parameter” : “<desired parameter>”,
   “Condition” : “<” | ”>” | “<=” | “>=” | “==” | “!=”
   “CondValue” : <condition value>
```

Note : In Lieu of the Location and Name Parameter, an an implementation may also choose to send only the Nid Key. Nid will be unique across all endpoints of the installation. Likewise, the combination of Name and location will be unique across the entire system.

<Name of the device> is the name which the device advertises

<device location> is the device’s location.
Note that the combination of the Device name and location will be unique for the entire installation.

<desired parameter> is the parameter which this entity wishes to subscribe to - eg., temperature, distance, etc.,

condition : The comparison being done. A match of this check will trigger the notification. Can be one out of <,>, >=, <=, ==, !=. These are self explanatory.

<condition value> : The value with which the parameter is being checked. For example, if specified as

```
“Parameter” : “Temperature”, “Condition”: “>”,
“CondValue” : 29
```

The subscriber will get a notification whenever the temperature is higher than 29.

When the subscription is successful, the SUBSCRIBE server MUST send a 202 Accepted message. The Header keys for this 202 are:

```
“Type” : 202
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String - currently 0.8
“Time” : Server's current time
“From” : Address of the server
```

14 NOTIFY

14.1 Overview

The NOTIFY method in IONMP allows all subscribed entities to receive notifications when events are triggered. This method is intended for communication between the server components. However any entity MAY use these specification if it is suitable for it.

14.2 Construct of a NOTIFY method:

The NOTIFY message is sent to Subscribers whenever certain events occur. When the Notifier wants to send out a NOTIFY message, it MUST send out a request to the subscriber with the following keys

“Type” : “NOTIFY”
“Mid” : Unique Message Id
“Nid” : Unique ID of the Notify Server
“Ver” : Version String – currently 0.8
“Time” : Time when the data was published. If data was published by an ION, this time should be sent to the published ION time
“From” : Address of the Notify server
“Data” : JSON object denoting the return value as described below

The value for the “Data” key is a JSON object which has details about the publishing entity. The format is :

“Parameter” : “ParamValue” ,
“SubscribeId” : “MID of the original SUBSCRIPTION message which triggered this NOTIFY message”

In lieu of the “Nid” parameter nested within the “Data” key, a combination of the Device Name and Location can be used.

When the NOTIFICATION message is received by the subscriber, it MUST send a 202 Accepted message. The Header keys for this 202 are:

“Type” : 202
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String – currently 0.8
“Time” : Subscriber's current time
“From” : Address of the subscriber

15 QUERY

15.1 Overview

The QUERY method in IONMP is used to discover the identity, address and capabilities of one or more entities. This message is always sent TO an infrastructure entity. In most of the cases, it is also sent BY and infrastructure entity.

15.2 Construct of a QUERY method:

The QUERY message is sent to the location server to discover about the capabilities of one or more end points. When an entity wants to discover the capability of an end point, then it MUST send out a request to the infrastructure with the following keys

“Type” : “QUERY” ,
“Ver” : “0.8”
“From” : “Sender_Address”,
“Name” : “The entity’s name as advertised”,
“Location” : “The location of the entity”,
“Target” : “DEVICE_ID” - The Device ID of the entity for which we want to find out the capabilities
“Mid” : “A_UNIQUE_VALUE”,
“Time” : node's_time

Note : In Lieu of the Location and Name Parameter, an an implementation may also choose to send only the Key. Nid will be unique across all endpoints of the installation. Likewise, the combination of Name and location will be unique across the entire system.

In Addition, the requestor MAY send out a “Key” header in which it sends out the 'key' or 'hashed key' value based on whether the initial registration was made using 9.3.1 or using section 9.3.2 This header is required when the server imposes security restrictions on requesting entities.

15.3 Response to a QUERY Method

The infrastructure responds to a QUERY method in one of the following ways.

14.3.1 When there is no active registraton for the specified TargetId

If there is no active registration for any entity with the device id of “DEVICE_ID” , then the server sends back a 404 Not Found.

```
{  “Type” : “404”, “Ver” : “0.8”, “From” : “Address of server”,
   “Mid” : MID Sent in original request, “Time” : server's_time }
```

15.3.2

When there is an active registration for the specified TargetId

In this case the server sends back a 200 OK.

```
{ “Type” : “200”,  “Ver” : “0.8”,  “From” : “Address of server”,
  “Mid” : MID Sent in original request, “Time” : server's_time,
  “Data” : An Array of JSON Object as specified below}
```

In addition the server SHOULD send back a Reason clause using the header key “RespClause”. The RespClause could be any String which the Server chooses but a common reason clause for 200 is “OK”.

The server MUST also send a “Data” header to represent the capabilities of the 'queried' end point. The value for the Data is an array of JSON Objects. If a single device is being queried, the array has only one element. If TargetId was specified as * the array could have multiple JSON Objects. JSON object's schema is:

```
   “Nid” : Node ID of the device being queried
   “Contact” : Address of the device being queried
   “Expires” : Amount of time left in the expiration
      + All other items from the advertisement.
```

Note that the Data array can be of zero or more length, although the typical length of the array would be 1.

Example of “Data” header :

```
   “Data” : [{ “Nid” : “Odev1”, “Contact” : “10.1.1.201”,
   “Expires”: 1253, “Name” : “Light”, “NodeType” : “Output”,
   “Action” : “Switch”,
   “Parameters” :[“State” : “True”],
   “Return” : {“State” : “Boolean”}
   }]
```

15.3.3 When a QUERY is made with TargetId of “*”

If the “TargetId” of the QUERY message has a value of “*”, then the infrastructure responds with the capabilities of ALL entities whose registration is active.

In this case the response from the server is exactly similar to the response in 14.3.2 except for the “Data” header. The “Data” header in this case is an array of JSON Objects. Each item of the array has the same schema as in 14.3.2

Server’s that implement the wild card(*) QUERY method should be extremely cautious. It allows the querying entity to get information about all entities of the system. A recommended technique is for servers to allow this method ONLY from it’s own host or from a host which is co-hosting some features – say for load balancing or high availability.

16 POLL for CONTROL messages

16.1 Overview

The POLLCONTROL method is sent by output devices to query whether there are any CONTROL messages pending for that output device. This method is used by those end points which either don’t want to implement a deamon to listen for messages or in case where the end point is behind a NAT and cannot be reached by the infrastructure. The latter is the more common reason for using the POLLCONTROL message

Only REGISTERED devices of a network can send POLLCONTROL messages. The message is only sent from an endpoint to the infrastructure

16.2 Frequency of POLLCONTROL messages

The IONMP protocol does not specify how frequently the end point needs to POLL the infrastructure. It is left to each implementation about how often it has to send these messages. However implementors must keep in mind that the POLLCONTROL message is used to actually control output devices. If the endpoint is too slow in POLLING for these events, users might experience a lot of sluggishness. On the other hand, if the endpoints poll for messages too often, then it may hog too much network bandwidth.

16.3 Construct of a POLLCONTROL Message

When a POLLCONTROL message is sent to the infrastructure, the sender sends out a POLLCONTROL request along along with its Device ID and the key that was computed during the Registration process. If the Registration was done without hash authentication (Section 9.3.1), then the Key is nothing else but the device key issued by the infrastructure. If the Registration was done using a two step authentication method (Section 9.3.2), then the key must be set to the hash value computed in the two step authentication process.

The entity which wishes to POLL for CONTROL MUST send out a request message to the infrastructure with the following keys

```
“Type” : “POLLCONTROL”
“Ver” : “0.8”
“From” : “Device_Address”
“Nid” : “DEVICE_ID”
“Mid” : “A unique ID for the control message ”
“Time” : node's_time
“Key” : “A Key depending on how the registration was done”
```

16.4 Response to a POLLCONTROL Method

The infrastructure responds to a POLLCONTROL method in one of the following ways.

1. In case there is a CONTROL message pending for that entity.

   This typically happens either when an output device has to be controlled due to an event trigger
   or
   If an external entity, say a user, wants to control the output device using a mobile app

   In this case, the infrastructure responds with a CONTROL message which is in the same format as a regular CONTROL which is explained in section 12 - CONTROL-ing Output devices. The endpoint responds to this message in the same way as it would have done, had it received the control message from this infrastructure.

2. In case there is no CONTROL message pending for that entity.

   If there are no CONTROL messages for the entity, the infrastructure responds with a 404 Not Found message

```
{  “Type” : “404”, “Ver” : “0.8”, “From” : “Address of server”,
   “Mid” : MID Sent in original request,
   “Nid” : NID sent in original request, “Time” : server's_time }
```

17 CTLRESPONSE messages

17.1 Overview

When an output node sends out a POLLCONTROL method and receives a CONTROL message in response, the Request-Response dialog is completed. The output node cannot communicate the status of the control operation back to the server as part of the dialog. This is due to the fact that IONMP chooses a Request-Response model. For every Request, there can be ONE and ONLY ONE Response. The CTLRESPONSE method is used by the output node to communicate the status of the previous CONTROL message.

A CTLRESPONSE message MUST be sent only after an output node receives a CONTROL message in response to its previous POLLCONTROL message. It SHALL NOT be sent in any other circumstances

17.2 Construct of a CTLRESPONSE Message

A CTLRESPONSE IONMP message is a message whose Type header is set to “CTLRESPONSE”. The message also includes the common header fields like Ver, From, Nid, Time and Key. These are constructed in the same way as any other IONMP message.

There are two significant differences in a CTLRESPONSE message when compared to other messages:

The MID field of this message MUST be set to the same MID as the previous CONTROL message for which this CTLRESPONSE is being sent.
The message has an additional Header called “Status”. The value for this field indicates the status of the previous CONTROL message. The Status can either be 200, 202 or 405.
200 means that the previous control message was received and executed
202 means that the previous control message was received and forwarded to the actuating component. But we don’t know if the actuation completed or not
405 means that the previous control message was an invalid message and the actuator cannot handle it.

A CTLRESPONSE message looks as follows:

```
   “Type” : “CTLRESPONSE”
   “Ver” : “0.8”
   “From” : “Device_Address”
   “Nid” : “DEVICE_ID”
   “Mid” : “A unique ID for the control message ”
   “Time” : node's_time
   “Key” : “A Key depending on how the registration was done”
   “Status” : “200|202|405”
```

17.3 Response to a CTLRESPONSE Method

The infrastructure responds to a well formed CTLRESPONSE method by sending a 202 Accepted message. Well formed means that

a) the message has a proper JSON syntax
b) Message’s MID matches the MID of a previous CONTROL message
c) The value of the Key header equals the authenticated key for this device. This value depends on whether the initial registration was done using a single step authentication or a two step authentication.

The 202 response has the following keys:

“Type” : 202
“Mid” : Same ID sent by the request
“Nid” : Same Device ID sent by request
“Ver” : Version String – currently 0.8
“Time” : Server's current time
“From” : Address of the server

The Response to a CTLRESPONSE request can also be a failure response The possible failure rsponses are :

1. If the JSON message is not well formed – As in all such responses, the response will be a 400 Bad Request

2. If the previous CONTROL message did not have proper authentication details, then a 401 Authentication Failed response is sent back

18 Summary of REQUESTS and RESPONSES

18.1 REQUESTS

REGISTER : Registration with Infrastructure

ADVERTISE : Advertising capabilities

PUBLISH : Publish devices readings to infrastructure

CONTROL : To send control messages to a node

SUBSCRIBE : Subscribing to events of interest

NOTIFY : Notify subscribers when an event of interest occurs

QUERY : Discover the capabilities of entities

POLLCONTROL : Check with the infrastructure if there are any control messages for the entity

CTLRESPONSE : After an entity receives a CONTROL message in response to a POLLCONTROL message, the entity may either process this message succesfully successfully or fail to do so. This status is communicated using the CTLRESPONSE Method.

18.2 RESPONSES

200 OK : On Successful Registration and Advertisement

202 Accepted : Publish, Control, Subscribe and Notify are successful

400 Bad Request : This response is sent for any Request if the JSON payload is malformed

401 Authentication failed : When Authentication fails for any message

403 Forbidden : When entity is not in infrastructure's database

404 Not Found : Either when there are no matching entries for a QUERY request or when there are no control messages for a POLLCONTROL message

405 Not Implemented : Response to a CONTROL message which has invalid data or parameters that that the endpoint doesn’t understand

19 Changes from Previous Version

NA - This is the first release version

20 References

1. SIP Protocol Specification - RFC3261
   https://tools.ietf.org/html/rfc3261#page-9

2. HTTP Protocol - RFC 7231
   https://tools.ietf.org/html/rfc7231

3. JSON Specification - RFC 7159
   https://tools.ietf.org/html/rfc7159

4. Request-Response Model
   https://en.wikipedia.org/wiki/Request%E2%80%93response

5. NAT Traversal Overview
   https://notes.shichao.io/tcpv1/ch7/

6. NAT Traversal
   https://en.wikipedia.org/wiki/NAT_traversal

7. Protocol for NAT Traversal
   https://tools.ietf.org/html/rfc4787

8. RFC Keywords
   https://www.ietf.org/rfc/rfc2119.txt

9. MD5 Hashing
   https://en.wikipedia.org/wiki/MD5

21. Acknowledgements

Tejaswi Chunduri - Embedded Developer and Alumni of BITS Pilani. For her invaluable review of the protocol and support in implementing parts of the protocol in the ThingsPing server implementation

Project mentors at BITS Pilani - For supporting our capstone project and providing valuable guidance. This protocol was partially implemented in our capstone project at BITS.

Associates at Qantom Software - Current and ex staff of Qantom

Ex Associates at Pingtel Corp., - Ex colleagues of Pingtel, who have been a never dying source of inspiration. The guidance provided by a pioneer of the SIP protocol has helped shaped many parts of this protocol.

22. Full Copyright Statement

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, provided that

a) The above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to Qantom Software Pvt. Ltd or other organizations, involved in this exercise.

b) Any modifications made to the protocol specification must be submitted back to Qantom software so that it might be incorporated by Qantom back to the protocol, if appropriate. The reason why Qantom has made this protocol as an open protocol is to ensure that IOT devices are more interoperable and to follow a standard open protocol. We expect a reciprocal from all entities who use this protocol and wish to modify / adapt this protocol to suit their needs.

The limited permissions granted above are perpetual and will not be revoked by Qantom Software or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and Qantom Software Private Limited DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

23. License

The protocol has been written to facilitate free and open interoperability and is thus release under a minimal restriction free license. This protocol may be freely used in accordance with the copyright section.

Put in simple terms, the terms of using this protocol standards are:

> You may use this protocol for any reason - commercial or non-commercial. You are allowed to modify the protocol to suit the needs of your implementation. In case you choose to modify the protocol, you MUST submit the changes back to us so that we can incorporate these changes back to the protocol. After all, we wrote this protocol to ensure that IOT devices interoperate from any vendor can interoperate with devices from other vendors!