English

Asema IoT Central

Notification API 1.0

ASEMA

Table of Contents

1. Introduction

Asema IoT Central is an event driven system which means that whenever something happens in the system, some event is generated. External parties can plug into some of these events in order to receive changes in the system, the measurements and the states of the objects.

Notifications are delivered through the notification API to the users of the API - the so called "subscribers". The notifier API offers three distinct technologies for notifications: HTTP Push, WebSockets and MQTT. The data available from all of these is equivalent. However, as they are different technologies, their use applies to different situations. What works well in one use case may be useless in some other. The purpose of supporting multiple methods is to have a toolkit to choose from in each application. Some application may also use multiple methods, Asema IoT Central does not limit this in any way.

Important
HTTP Push, WebSockets and MQTT are protocols based on somewhat different design philosophies, the most notable being that HTTP Push and WebSockets have been designed for point-to-point communication whereas MQTT is for point-to-multipoint. Due to the difference, their configuration and use differs so pay attention to the differences in details when setting these up. Notice especially how they behave when there are network problems (disconnects, offline units, etc) and when a reconnect / resubscribe is needed.

1.1. HTTP Push

HTTP Push is a **connectionless**, **point-to-point** notification technology. The upside of this design is that because it is connectionless, it can be very lightweight on the servers as no connections need to be kept open. Not having connections open makes the method also robust to lost connectivity and easy to manage in proxies and load balancers. The downside is that opening connections takes time which slows it down when a large amount of data needs to be delivered. However, in case the recipient also supports it, Asema IoT Central does support HTTP with keepalive so that the connection is not cut between messages. This greatly improves throughput, especially if encryption is user. The other potential downside is that both parties need to be able to receive a connection, i.e. they need to be servers. So to receive HTTP Push notifications, you need an HTTP server.

To receive HTTP Push notifications, you need to send a subscribe command to the Asema IoT Central JSON API. See below for details of the contents and format of the command. The response is the number of a stream that identifies the subscription. This number acts as the handle for all further operations with the subscription.

Because the HTTP Push protocol is connectionless, there needs to be a way to detect stale subscriptions of notifications. For this reason Asema IoT Central uses an acknowledgement procedure for HTTP Push messages. This means that whenever you receive an HTTP Push notification, you must ack it. If not, the subscription will eventually be closed (the system allows for a few missed acks before it thinks the connection is stale and removes it). A closed subscription must be resubscribed to before further notifications are sent. To help in recognizing a dead subscription, Asema IoT Central will regularly (with 30 second interval) send a short heartbeat (a notiication called "PING") over the channel. So if no ping is reveiced in 45 seconds, the subscription is dead. If that happens, resubscribe.

HTTP is in general well supported by all modern programming languages. So taking it into use should not require the installation of any additional packages or modules.

1.2. WebSockets

WebSockets is a **connected**, **point-to-point** notification technology. As there is an ongoing connection, the subscriber is responsible for initiating the connection i.e. is the client. To implement WebSockets, you don't need a server, just a standards compliant WebSocket client.

To receive WebSocket notifications, you need to send a subscribe message to the Asema IoT Central WebSocket API. See below for details. The response is the number of a stream that identifies the subscription. Similar to HTTP Push, this number acts as the handle for all further operations with the subscription. However, note that the handle is unique to WebSockets. If you already have a notification stream number for HTTP Push, it will not work for WebSockets and vice versa.

As each notification stream requires a connection, the socket is kept open for the duration of the subscription. Note that the server that runs Asema IoT Central always has a limit, though usually quite high, to the number of connections that can be kept open. Exceeding the limit will cause new connections to fail.

If a WebSocket connection closes, the subscription closes. To properly manage the subscription, you need to have some bookkeeping of open notifier connection states and recognize closed sockets. Usually WebSocket clients to use include a callback that is invoked when the socket closes. Once a socket closes, the subscription must be redone.

Clients and modules for WebSockets can be found for most programming languages. Implementation will depend on the module you choose. Please refer to the documentation of those modules on the practicalities of actual programming.

1.3. MQTT

MQTT is a **connected**, **point-to-multipoint** notification technology. Unlike the two other notification methods, MQTT uses a special server called a broker to queue and send messages. Everyone, including Asema IoT Central itself, is a client to that broker. Usually the broker is a completely separate service, often running in a completely separate server somewhere. That said, to simplify the configuration in many cases, Asema IoT Central does integrate its own MQTT server so that there is no need for a separate one. In this case Asema IoT Central willl act a client to itself internally.

To receive MQTT notifications, you subscribe to a topic **at the broker**. The subscription contains a topic. Any notification published by someone to the same broker will then be sent back to you. There is no stream number, just the topic. To unsubscribe, you unsubscribe from the topic.

Because of the multipoint nature, MQTT subscriptions do not die with connection errors. The broker will simply detect that your socket associated with the topic no longer exists and will discard the message (unless it is set to queue them and wait for reconnect). Once reconnected, the notifications will again be sent to the subscriber(s).

Of the three protocols, MQTT is perhaps the least widely supported in programming languages. However, especially open source clients do exist for the majority of programming languages. Take a look at especially the open source Paho libraries of the Eclipse project.

2. Using HTTP Push

2.1. Subscribing

Subscribing to HTTP push takes place by sending one JSON-RPC message to the JSON API of Asema IoT Central. The message has the method `subscribe` and as a sole parameter, in the form of a list, the URL to which the notification is to be pushed. When a notification occurs, this URL will be called by Asema IoT Central.

```
{
 "id": 1,
 "method": "subscribe",
 "params": ["http://127.0.0.1:5555/notify"]
}
```

The return value of the call is a result that contains the `stream_id` of the subscription.

```
{
 "id": 1,
 "method": "subscribe",
 "result": {
  "stream_id": 10
 }
}
```

To add objects to this stream, use `add_to_subscription`. The parameters are again a list. The first value of the list is the stream id (just received above), the next values are the GIDs of the objects to subscribe to.

```
{
 "id": 2,
 "method": "add_to_subscription",
 "params": [10, "abcdefgh123456"]
}
```

2.2. Parsing incoming data

Incoming data for HTTP Push will contain a structure familiar from JSON RPC. It has an `id` (always zero), a `method` and `params` that define the data. The method will tell you what the notification is about. For a full list of methods, see the last chapter of this document.

The `params` will depend slightly on the type of notification method received but in general the content is a GID that tells which object this notification refers to and values the are notified. Below is an example of the most common notification: property change.

```
{
    "id": 0,
    "method": "property_changed",
    "params": {
        "gid": "abcdefgh123456",
        "property": "temperature",
        "stream_id": 10,
        "value": "23.4"
    }
}
```

The format is JSON so you can use your favorite JSON tool for parsing the structure and then take action according to your program logic.

2.3. HTTP Push example in Python

The following code is an example in Python on how to receive notifications over HTTP Push. It uses the Tornado HTTP server for Python to receive the notifications.

Python urllib2 is used to send HTTP messages for subscriptions. When a notification arrives, Tornado calls `post` method of `MainHandler` which then uses simplejson `loads` to parse the data from JSON into a native Python object.

```
#!/usr/bin/python

import sys
import urllib, urllib2
import argparse
import datetime
import tornado.ioloop
import tornado.web
from simplejson import loads, dumps


CENTRAL_IP = "127.0.0.1"
CENTRAL_PORT = 8080
LOCAL_IP = "127.0.0.1"

streamId = None
requestId = 0

# Helper for making HTTP POST calls using Python urllib2
def make_json_post_request(url, command, params):
 global requestId

 requestId += 1
 payload = { 'id': requestId, 'method': command, 'params': params }
 headers = { 'X-Requested-With' : 'XMLHttpRequest', "content-type" : 'application/json' }
 result = None
 try:
  req = urllib2.Request(url, dumps(payload), headers)
  filehandle = urllib2.urlopen(req, timeout = 5)
  if filehandle is not None:
   data = filehandle.read()
   result = loads(data)
 except:
  print "Failed in contacting", url
 finally:
  return result


class MainHandler(tornado.web.RequestHandler):
 def post(self):
  payload = self.request.body
  data = loads(payload)['params']
  if data.has_key("property"):
   print "Incoming property notification: GID", data['gid'], "property:", data['property'], "=", data["value"]

  if data.has_key("state"):
   print "Incoming state notification: GID", data['gid'], "state:", data["state"]

  # The subscriber must send an "OK" back, otherwise the gateway will assume the service
  # is offline and will cut the subscription
  response = "OK"
  self.set_status(200)
  self.write(response)
  self.finish()
  return

 # suppress the log output for connections
 def log_message(self, format, *args):
  return

# Subscribe to a stream
def subscribe(gids, port):
 global streamId
 gw_url = "http://" + CENTRAL_IP + ":" + str(CENTRAL_PORT) + "/json"
 my_url = "http://" + LOCAL_IP + ":" + str(port) + "/notify"

 streamData = make_json_post_request(gw_url, "subscribe", [my_url])
 if streamData.has_key("error") and streamData["error"] is not None:
  print "Subscription error:", streamData["error"]["message"]
  return

 streamId = streamData['result']['stream_id']
 for g in gids:
  result = make_json_post_request(gw_url, "add_subscription_to_stream", [streamId, g])
  if result.has_key("error") and result["error"] is not None:
   print "Subscription error:", result["error"]["message"]
   return

# Unsubscribing the stream
def unsubscribe(streamId):
 gw_url = "http://" + CENTRAL_IP + ":" + str(CENTRAL_PORT) + "/json"
 make_json_post_request(gw_url, "unsubscribe", [streamId])


def main():
 gw_url = "http://" + CENTRAL_IP + ":" + str(CENTRAL_PORT) + "/json"

 url = gw_url
 print "\n============================================="
 print "API demo for HTTP push notifications from", url
 print "=============================================="

 parser = argparse.ArgumentParser(description='Subscribe to data from a gateway and listen to it over HTTP.')
 parser.add_argument('--port', dest='port', type=int, default=1111,
     help='Local port to be use by this script')
 parser.add_argument('--gids', dest='gids', default=None, nargs="+",
     help='The GID(s) of the object to subscribe to')

 args = parser.parse_args()
 if args.gids is not None:
  subscribe(args.gids, args.port)
 else:
  parser.print_help()
  sys.exit()

 application = tornado.web.Application([
  (r"/notify", MainHandler)
  ])

 print 'Listening to data at port ' , args.port
 application.listen(args.port)

 # Wait forever for incoming http requests (or until ctrl+c)
 try:
  tornado.ioloop.IOLoop.current().start()
 except KeyboardInterrupt:
  # Be polite, unsubscribe as we go offline
  unsubscribe(streamId)

if __name__ == '__main__':
 main()
```

3. Using WebSockets

3.1. Subscribing

To subscribe to notifications over WebSockets, a payload with method `subscribe` should be sent to the WebSocket server of Asema IoT Central. No params are needed as the socket that is opened will be used as the channel for notifications.

```
{
 "id": 2,
 "method": "subscribe",
 "params": []
}
```

As in HTTP Push, the return value will again contain the stream id of the subscription. To add objects to this stream, send a payload containing the `add_to_subscription` method and the stream id and objects to add to the stream like so (number 10 here is the stream id in this particular case, replace as necessary):

```
{
 "id": 2,
 "method": "add_to_subscription",
 "params": [10, "abcdefgh123456"]
}
```

3.2. Parsing incoming data

Incoming data for WebSockets looks exactly the same as for HTTP Push. So if you can parse one, you can parse the other. Below again an example for property change. For further explanation, please refer to the chapter on HTTP Push.

```
{
    "id": 0,
    "method": "property_changed",
    "params": {
        "gid": "abcdefgh123456",
        "property": "temperature",
        "stream_id": 10,
        "value": "23.4"
    }
}
```

3.3. WebSockets example in Python

The following example uses the Python websocket module for subscribing to data notifications. The notable difference to the HTTP Push is that Python websocket is callback based. Whenever some message is received, the module invokes the `on_message` method. Action on the stream must be placed in this callback. In the example a call id is used to recognize the response to the initial subscription so that stream additions can be separated from the rest of the messages to the stream.

```
#!/usr/bin/python

import sys
import websocket
import argparse
import datetime
from simplejson import loads, dumps
```

```
CENTRAL_IP = "127.0.0.1"

class WebSocketListener(object):
 def __init__(self, port, gids):
  self.ws = websocket.WebSocketApp("ws://127.0.0.1:%s/"%port,
       on_message = self.on_message,
       on_error = self.on_error,
       on_close = self.on_close)
  self.ws.on_open = self.on_open
  self.gids = gids
  self.streamId = None
  self.callId = 0
  self.subscriptionCallId = -1

 def serve_forever(self):
  self.ws.run_forever()

 def on_message(self, ws, message):
  json = loads(message)

  if json['id'] == self.subscriptionCallId:
   if json['errors'] is None:
     self.streamId = json['stream_id']
    for g in self.gids:
     self.callId += 1
     payload = { 'id': self.callId, 'method': "add_subscription_to_stream", 'params': [self.streamId, g] }
     self.sendWebSocketMessage(dumps(payload))
  else:
   if json['method'] == "controller_state_changed":
    print "Incoming state notification: GID", json['params']['gid'], "state:", json['params']["state"]

   elif json['method'] == "property_changed":
    print "Incoming property notification: GID", json['params']['gid'], "property:", json['params']['property'], "=",
 json['params']["value"]

 def on_error(self, ws, error):
  print error

 def on_close(self, ws):
  print "### closed ###"

 def on_open(self, ws):
  self.subscribe()

 def subscribe(self):
  self.callId += 1
  self.subscriptionCallId = self.callId
  payload = { 'id': self.callId, 'method': "subscribe", 'params': [] }
  self.sendWebSocketMessage(dumps(payload))

 def unsubscribe():
  self.callId += 1
  payload = { 'id': self.callId, 'method': "unsubscribe", 'params': [self.streamId] }
  self.sendWebSocketMessage(dumps(payload))

 def sendWebSocketMessage(self, msg):
  self.ws.send(msg)


def main():
 print "\n============================================="
 print "API demo for WebSocket notifications from", CENTRAL_IP
 print "==============================================="

 parser = argparse.ArgumentParser(description='Subscribe to data from a gateway and listen to it over HTTP.')
 parser.add_argument('--port', dest='port', type=int, default=8081,
     help='WebSocket port of the server')
 parser.add_argument('--gids', dest='gids', default=None, nargs="+",
     help='The GID(s) of the object to subscribe to')

 args = parser.parse_args()
 if args.gids is not None and  args.port is not None:
  listener = WebSocketListener(args.port, args.gids)
 else:
  parser.print_help()
  sys.exit()

 print 'Connecting to websocket port ' , args.port

 #Wait forever for incoming data (or until ctrl+c)
 try:
  listener.serve_forever()
 except KeyboardInterrupt:
  # Be polite, unsubscribe as we go offline
  listener.unsubscribe()

if __name__ == '__main__':
 main()
```

4. Using MQTT

4.1. Subscribing

Subscribing to MQTT notifications takes place by subscribing to a topic at a broker. There is no payload to send to Asema IoT Central, the subscription always takes place at the broker. If you use Asema IoT Central itself as the broker, remember to call the correct port i.e. the port you have configured as the MQTT server port. How to contact the broker and inform it about a topic depends on the MQTT client software you use. The example code below shows one method with Python and Paho.

As the subscription is not sent to Asema IoT Central, it must be made aware of the requirement to send notifications. This also applies to the case when Asema IoT Central is the broker. This takes place at the admin UI of Asema IoT Central. At the object configuration menu there is a configuration window for each object. Notifications are activated here by entering the topic per each object and then setting notifications to "ON" for that object.

4.2. Parsing incoming data

Incoming data for MQTT is again identical to the other methods, WebSockets and HTTP Push. The only notable difference is that `params` will not contain a stream id as MQTT does not use one. Below again an example for property change. For further explanation, please refer to the chapter on HTTP Push.

```
{
    "method": "property_changed",
    "params": {
        "gid": "2df0f78fd3690150c49e532857335ef5d8dc11dc",
        "property": "temperature",
        "value": "23.4"
    }
}
```

4.3. MQTT example in Python

The program listing below shows how to make MQTT subscriptions with Python and Paho. Make sure you are contacting the correct broker and have the correct topic. It is easy to make a mistake in one of these.

The code follows the structure of basic Paho MQTT client operation. For details and more features, please refer to the documentation of the client you use.

```
#!/usr/bin/python

import sys
from simplejson import loads, dumps
import argparse
import datetime
import paho.mqtt.client as mqtt

BROKER_IP = "127.0.0.1"
BROKER_PORT = 1883

class MqttListener(object):
 def __init__(self, topics):
  self.topics = topics
  self.client = mqtt.Client()
  self.client.on_connect = self.on_connect
  self.client.on_disconnect = self.on_disconnect
  self.client.on_message = self.on_notification
  self.client.connect(BROKER_IP, BROKER_PORT, 60)

 # Wait for notifications
 def run(self):
  self.client.loop_forever()
 # The callback for when the client receives a CONNACK response from the server.
 def on_connect(self, client, userdata, flags, rc):
  for t in self.topics:
   print "subscribe to", t
   client.subscribe(t)

 # The callback on socket disconnects with broker.
 def on_disconnect(self, client, userdata, rc):
  if rc != 0:
   print("Unexpected disconnection.")

 # The callback for when a PUBLISH message is received from the server.
 def on_notification(self, client, userdata, msg):
  # parse the incoming message using simple json
  json = loads(msg.payload)
  if json['method'] == "controller_state_changed":
   print "Incoming state notification: GID", json['params']['gid'], "state:", json['params']["state"]

  elif json['method'] == "property_changed":
   print "Incoming property notification: GID", json['params']['gid'], "property:", json['params']['property'], "=",
 json['params']["value"]

def main():
 print "\n=============================================="
 print "API demo for MQTT notifications from", BROKER_IP
 print "=============================================="

 parser = argparse.ArgumentParser(description='Subscribe to data from a broker and listen to it over MQTT.')
 parser.add_argument('--topic', dest='topic', default=None, nargs="+",
     help='The topic to listen to')

 args = parser.parse_args()
 if args.topic is not None:
  listener = MqttListener(args.topic)
  listener.run()
 else:
  parser.print_help()
  sys.exit()

if __name__ == '__main__':
 main()
```

5. Supported notifications and their content

5.1. Objects added

Sent when a new object is added to the system. Params contains the GIDs of the added objects.

```
{
    "id": 0,
    "method": "objects_added",
    "params": {
        "objects": ["abcdefgh123456", "efgh123456abcdf"],
        "stream_id": 10,
    }
}
```

5.2. Objects updated

Sent when the settings (not properties) of an object have changed in the system. Params contains the GIDs of the changed objects.

```
{
    "id": 0,
    "method": "objects_updated",
    "params": {
        "objects": ["abcdefgh123456", "efgh123456abcdf"],
        "stream_id": 10,
    }
}
```

5.3. Objects deleted

Sent when the objecst have been deleted from the system. Params contains the GIDs of the deleted objects.

```
{
    "id": 0,
    "method": "objects_deleted",
    "params": {
        "objects": ["abcdefgh123456"],
        "stream_id": 10,
    }
}
```

5.4. Property changed

Sent when a property of an objecst has changed. Params contains the GID of the object, the name of the property and the value of the property.

```
{
    "id": 0,
    "method": "property_changed",
    "params": {
        "gid": "2df0f78fd3690150c49e532857335ef5d8dc11dc",
        "property": "temperature",
        "value": "23.4"
    }
}
```

5.5. Property collection changed

Sent when several properties of an objecst has changed and a mass notification was requested. Params contains properties, which is a dictionary of several property value pairs.

```
{
    "id": 0,
    "method": "property_collection_changed",
    "params": {
        "gid": "2df0f78fd3690150c49e532857335ef5d8dc11dc",
        "properties": {
            "temperature": "23.4",
            "speed": "60",
        ]
    }
}
```

5.6. Actuator state changed

Sent when the state of an actuator changes. Params contains the value of the state which can be one of the following: `off, on, denied_off, denied_on, wait_on, wait_off, blocked, paused`

```
{
    "id": 0,
    "method": "actuator_state_changed",
    "params": {
        "gid": "2df0f78fd3690150c49e532857335ef5d8dc11dc",
        "state": "off"
    }
}
```

5.7. Heartbeat

Sent when a heartbeat signal has been received from a controller

```
{
    "id": 0,
    "method": "heartbeat",
    "params": {
        "stream_id": 10,
        "gid": "2df0f78fd3690150c49e532857335ef5d8dc11dc"
    }
}
```

5.8. Error

Sent when a notifiable error occurs in the system. Params contains the error code and error message.

```
{
    "id": 0,
    "method": "error",
    "params": {
        "stream_id": 10,
        "gid": "2df0f78fd3690150c49e532857335ef5d8dc11dc",
        "error_code": "200",
        "message": "Example error"
    }
}
```

5.9. Plugin invokation complete

Sent when some added plugin has finished running. Params will contain the call_id originally sent to the plugin, the name of the plugin and a variant containing any data that the plugin may produce once it has finished.

```
{
    "id": 0,
    "method": "plugin_invokation_complete",
    "params": {
        "stream_id": 10,
        "call_id": 21,
        "plugin_name": "myfancyplugin",
        "data": {}
    }
}
```

Asema Electronics Ltd
Copyright © 2011-2019