M2M Protocol Interoperability Using the Smart Object API

Michael J Koster
October 4th, 2013

The previous post in this series discussed the concept of adding an event model to RESTful APIs. This is a follow-on to that discussion, describing how the object model and event model implemented in the Smart Object API are used to create a multi-protocol IoT hub, simultaneously exposing MQTT, CoAP, and HTTP/REST interfaces to an instance of a Smart Object.

A connected weather Station demonstrates multiple Smart Object instances, residing in different gateways, servers, and cloud services, connected together and updated in an event-driven network using the different protocols, also updating a live feed on the Xively service.

What is the Smart Object API?
To review, the Smart Object API is a web object encapsulation of the Observable Properties, Data Model, Event Model, and Software Components that make up the virtual representation of a connected object or data source.

Figure 1 shows how a Smart Object encapsulates a set of resources and exposes them as M2M Protocol *Endpoints*. The endpoints of different M2M protocols can expose a single Smart Object, providing a multi-protocol bridge function with embedded data models, protocol translation, and event forwarding.

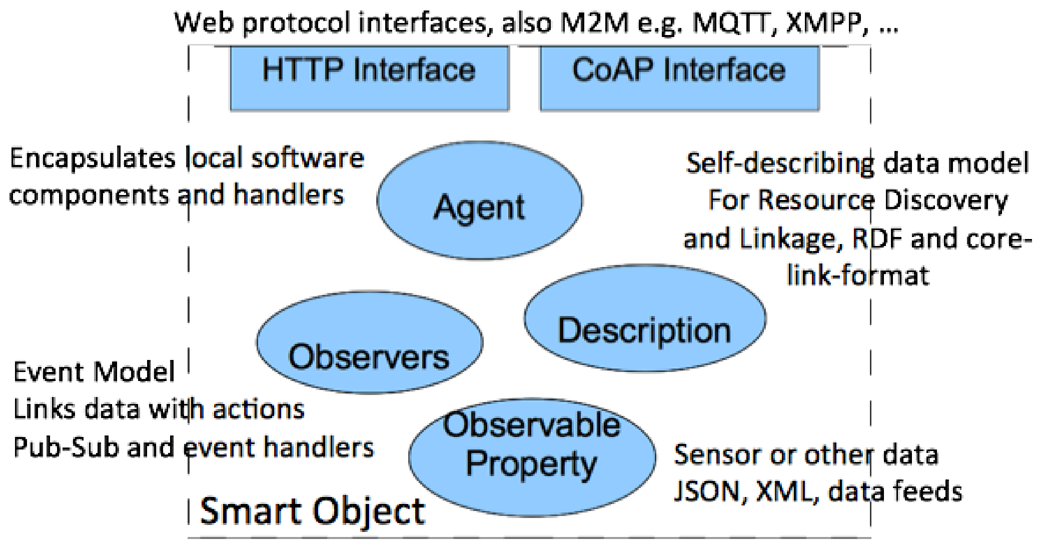

Web protocol interfaces, also M2M e.g. MQTT, XMPP, ...
HTTP Interface
CoAP Interface
Encapsulates local software components and handlers
Agent
Self-describing data model For Resource Discovery and Linkage, RDF and core-link-format
Description
Observers
Event Model Links data with actions Pub-Sub and event handlers
Observable Property
Sensor or other data JSON, XML, data feeds
Smart Object

Figure 1 - Smart Object Resource Encapsulation

The **Observable Property** supplies the most recently updated representation of a data source, which can be practically any supported content type, and optionally can provide time and location along with sample history and other metadata. A Smart Object may contain zero or more Observable Properties.

Observers map data to actions, resulting in event driven communication or computation based on updates of Observable Property data or metadata. Observer classes include *Subscribers*, *Publishers*, and *Event Handlers*. These map changing data to common event driven software patterns.

The **Description** contains the Data Model for the Smart Object, including the structural and taxonomic models for resources and resource configuration, and high level information models for application level discovery and linkage.

The **Agent** is a container for the software that runs the Smart Object, consisting of locally executed event handlers and background processes (daemons). Event handlers are invoked by the local Observers configured to monitor local Observable Properties.

Figure 2 shows the structure of a Smart Object and the layered encapsulation of Observable Properties, which themselves have Observers and Descriptions as associated properties.

An Observable Property may have zero or more Observers, in any combination of Publishers, Subscribers, and Event Handlers.

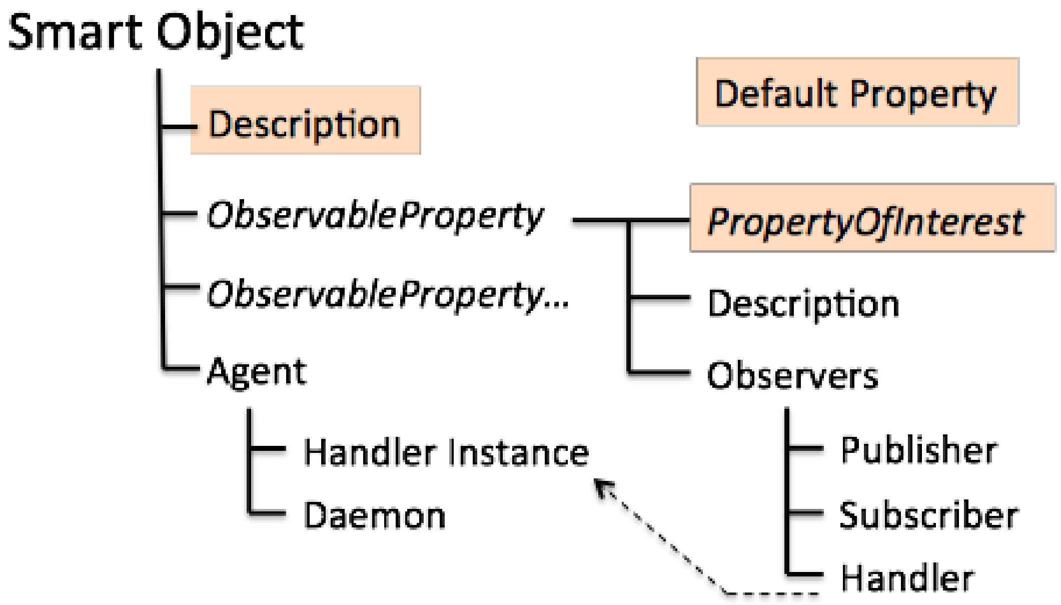


Figure 2 - Smart Object Structure

For more information about the SmartObject, with demo and code examples, see the online presentation at Slideshare:

IoT Toolkit and the Smart Object API - Architecture for Interoperability

M2M protocols mapped onto the Smart Object resource and event models

The Publish/Subscribe model used in MQTT and many other M2M systems is very easily mapped to resource observers. Figure 3 shows a diagram of an MQTT Observer associated with a Smart Object Observable Property such that it can publish updates to the MQTT broker that result from PUT operations and subscribe to the MQTT broker updates and apply them to the Observable Property .

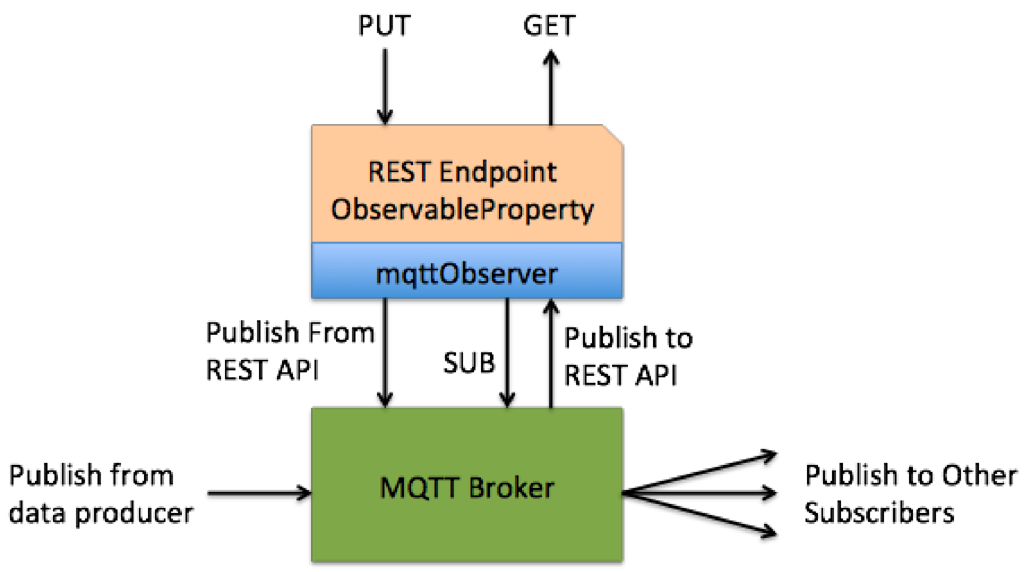


Figure 3 - MQTT Observer

The underlying binding that makes this work is using the REST resource path as the MQTT Topic. This way, the topic can be used to create and update REST endpoints, and REST updates can be mirrored to similar paths in other Smart Object instances. This also facilitates association of Description metadata with MQTT topics at endpoints.

Figure 4 shows the Observer-Publisher publishing the update resulting from an HTTP PUT ot CoAP POST to update the resource. In general, a Create operation could also result in the publishing of a topic, either on the create itself or on the first update.

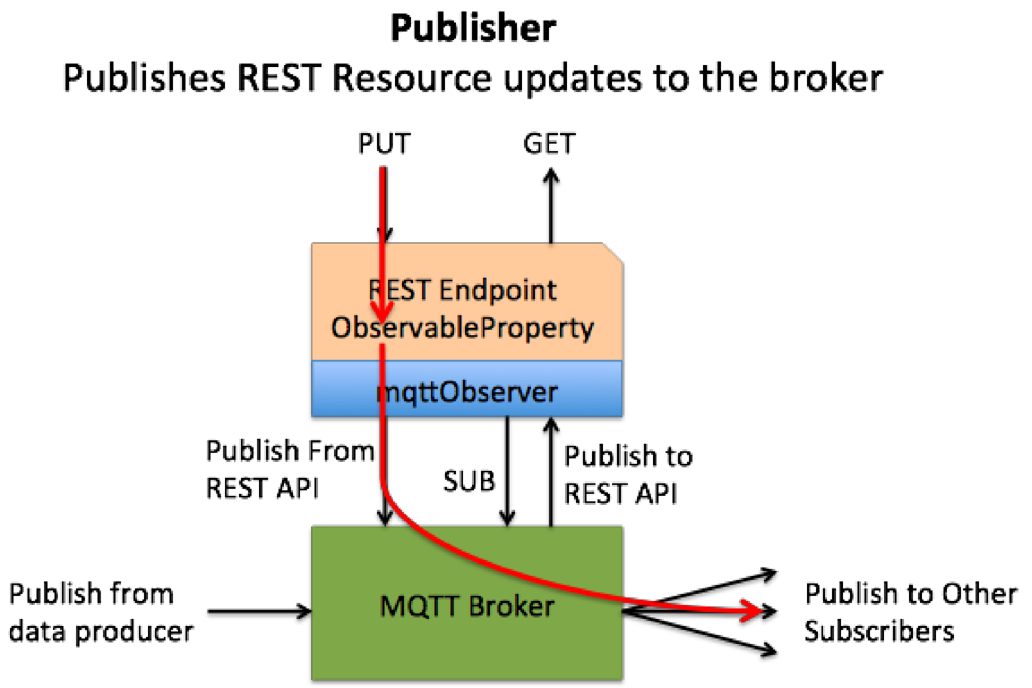


Figure 4 - Publication of REST updates

Figure 5 shows how the Observer-Subscriber creates a subscription in the broker, which updates the REST endpoint when a message is published from the broker on the topic corresponding to the resource path.

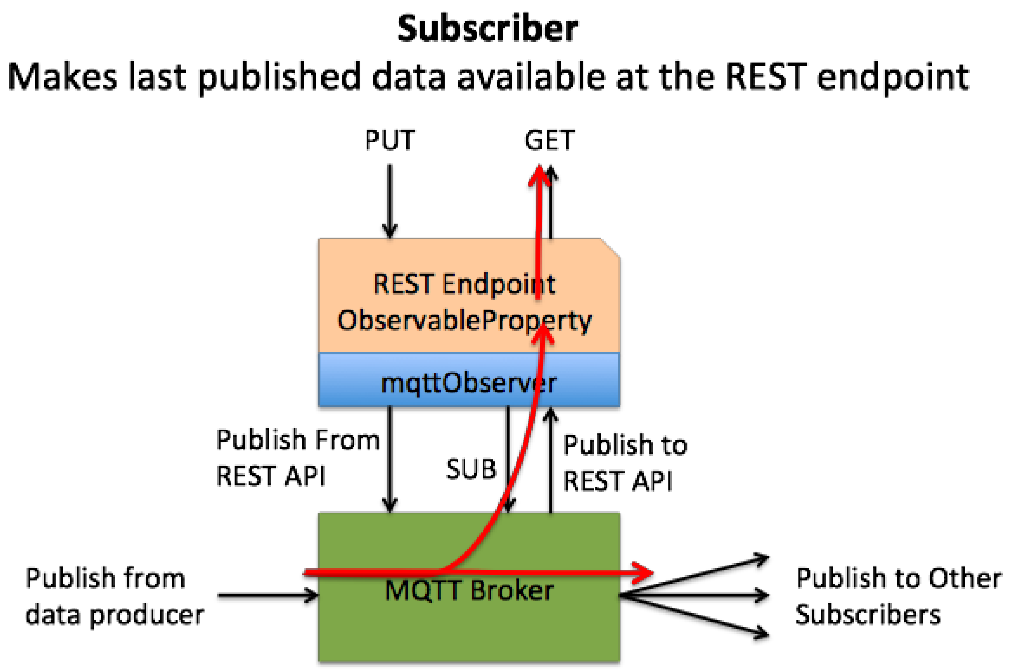


Figure 5 - MQTT Subscriber-Observer

Figure 6 shows how the Publisher and Subscriber can be used together to create an MQTT REST bridge. Topics published by the broker update the REST endpoint, and REST updates publish the topic to other subscribers.

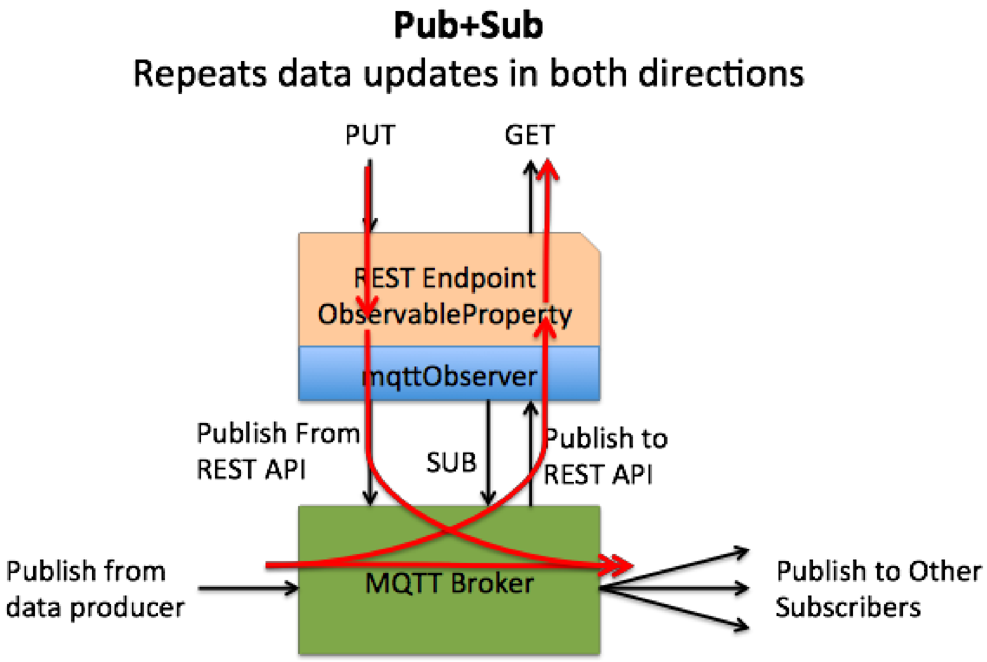


Figure 6 - MQTT REST Bridge

Figure 7 illustrates that multiple Smart Object instances can connect to a broker and participate in read-write sharing of REST endpoints across MQTT (or other protocol) connections.

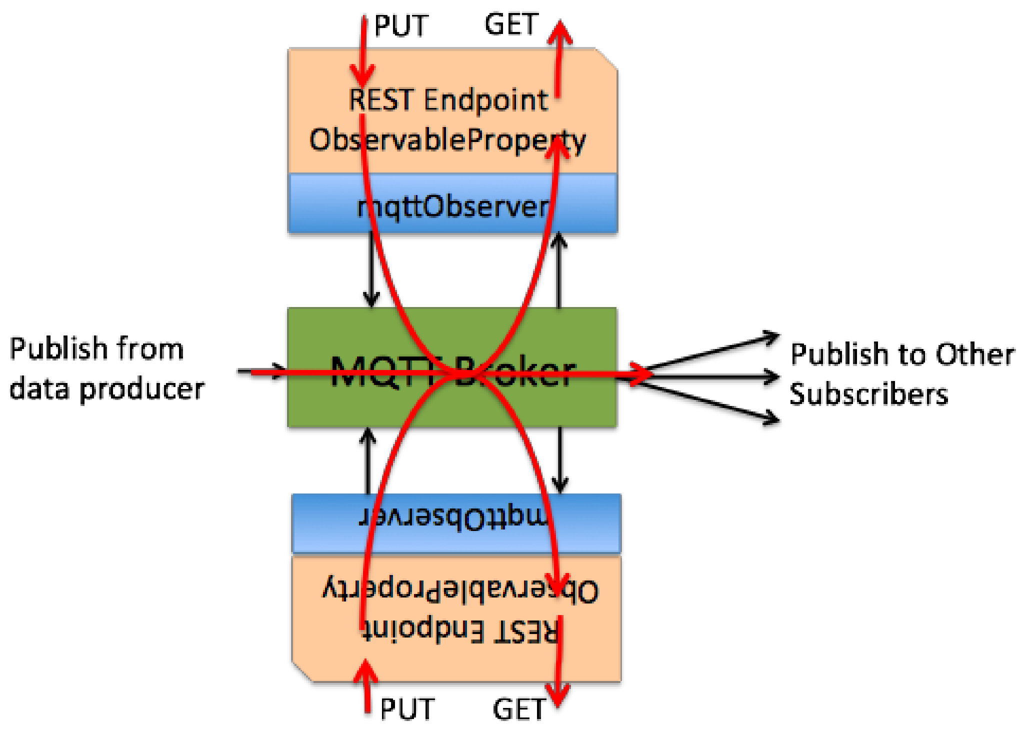


Figure 7 - Multiple REST endpoints with shared updates

CoAP Bridge using the object model and a semantic proxy

The CoAP protocol also includes a REST resource model with events and a rudimentary data model. The core-link-format (RFC6990) provides a link-format encoding and namespace for a simple semantic graph. The IPSO Application Framework builds on the basic vocabulary of core-link-format to describe home and building automation concepts.

The CoAP Event model is based on an Observe extension to the GET operation that subscribes the client to changes in the Observed resource. these changes are best-effort communicated back on the return socket connection to the client in a similar fashion as websockets but using UDP.

The Object model of CoAP is a subset of the resource model of the Smart Object API, allowing the Smart Object Observable Properties to be exposed as CoAP endpoints.

Figure 8 shows how the CoAP interface exposes RDF triples that are stored in the Description graph as link-format relations through a *Semantic Proxy*.

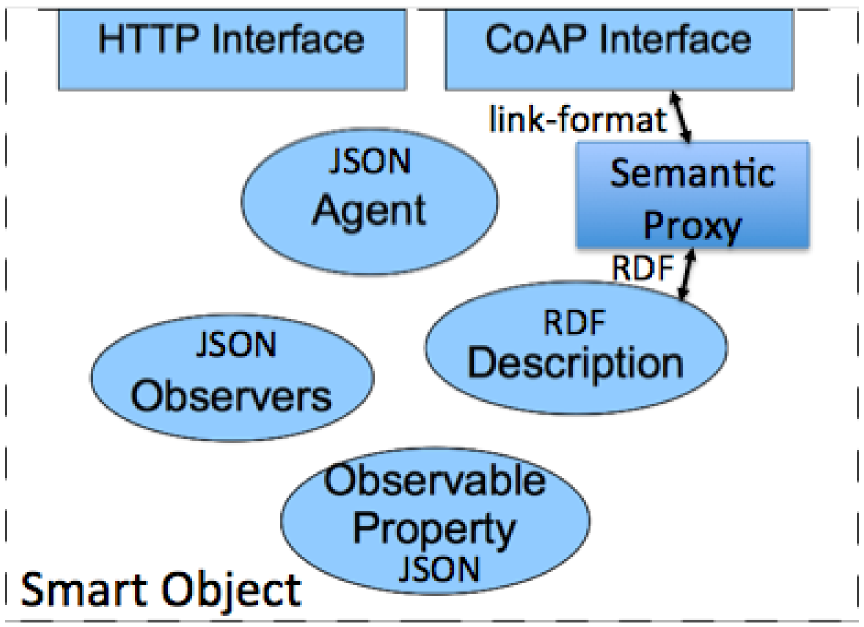


Figure 8 - Smart Object Resources exposed through the CoAP Interface and Semantic Proxy

The function of the semantic proxy is to provide a binding of the predicates in one namespace to and from the predicates in the other namespace, allowing a common RDF representation of concepts in both namespaces.

M2M Protocol Interoperability Demonstration based on a Weather Sensor

These concepts are demonstrated in concert in **figure 9**, which depicts our live on-line demonstration system involving multiple Smart Object instances in gateways, servers, and cloud instances communicating with each other through various M2M protocols.

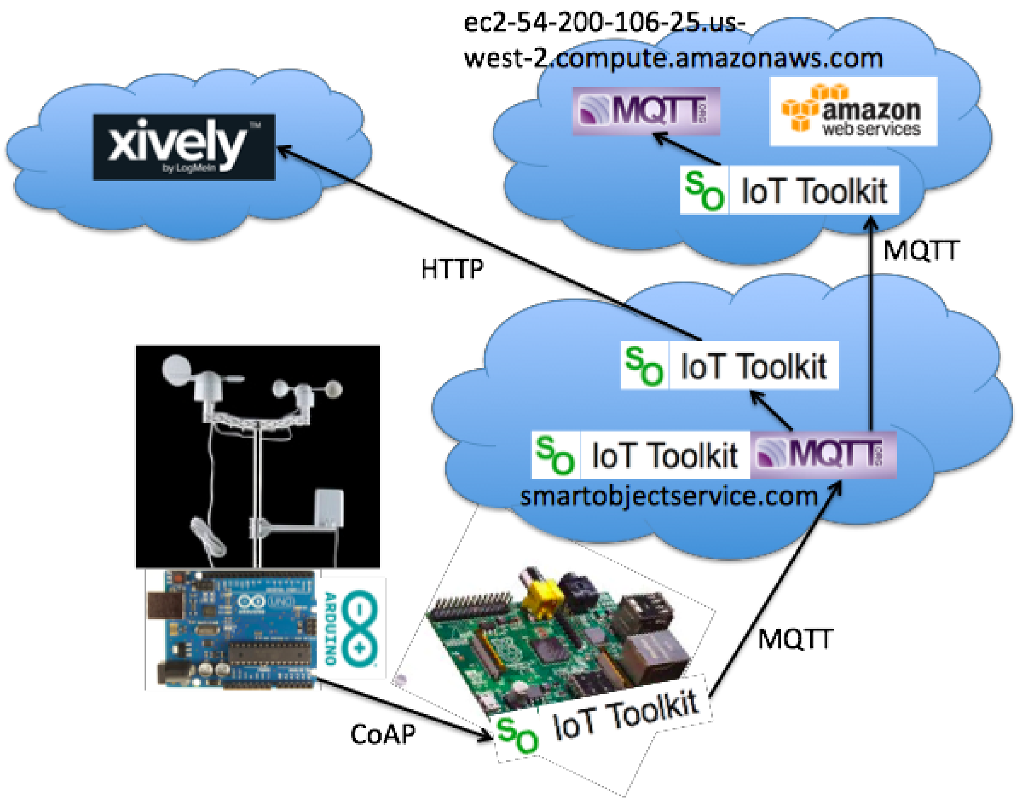


FIgure 9 - Connected Weather Sensor
Demonstration of Multi-Protocol Interoperabilty

The Arduino reads the hardware sensors and updates Observable Properties in the gateway (single board computer e.g. Raspberry Pi) using CoAP POST or HTTP PUT.

AN MQTT Observer in the gateway relays updates to a SmartObject instance in the PaaS at http://smartobjectservice.com, which operates an MQTT broker and HTTP + CoAP REST bridge.

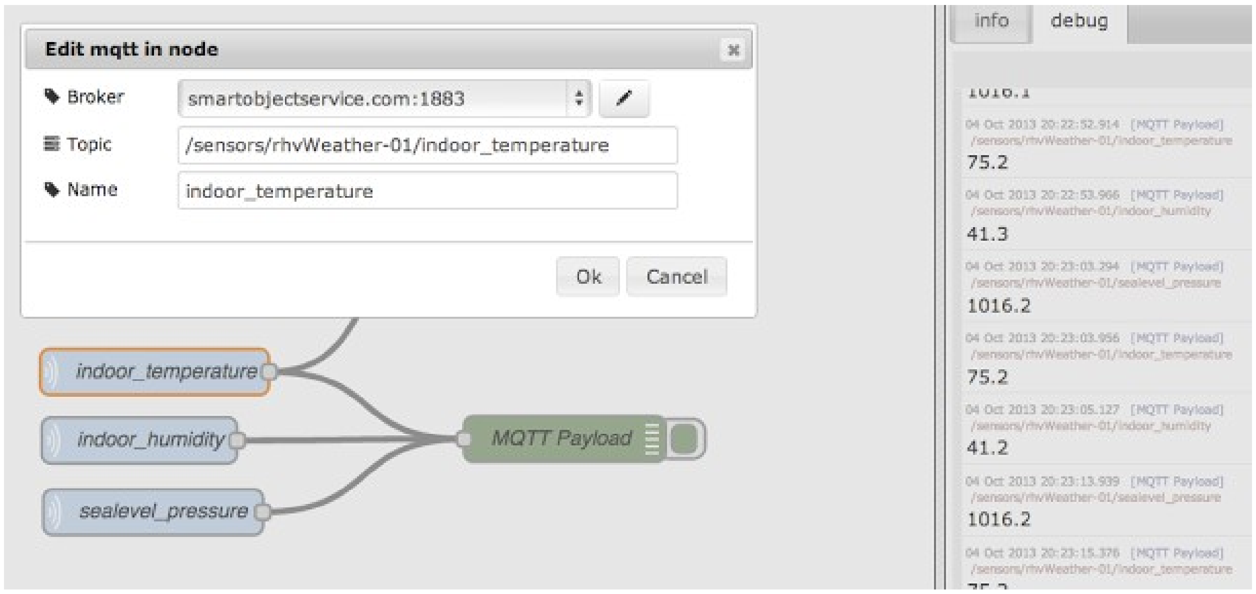
A second Smart Object instance at the PaaS subscribes to the MQTT broker and updates itself. These updates are filtered and aggregated to reduce load on the 3rd part API and sent to Xively to update the data feed for storage, viewing, and sharing.

Another Smart Object instance in a user's personal cloud (here an Amazon EC2 t1.micro instance) also subscribes to the MQTT broker and provides a local Smart Object and MQTT broker to supply weather data to weather forecasting services and other data users for a small monthly compensation. This instance provides CoAP, HTTP, and MQTT access to live weather data from the weather sensor.

Node-RED integration

Node-RED provides an event-graph programming model that is a natural fit for the Smart Object Event Model.

Node-RED instances are easily connected over the network interface by subscribing to MQTT topics and publishing back to the REST broker using existing I/O nodes.



Broker: smartobjectservice.com:1883
Topic: /sensors/rhvWeather-01/indoor_temperature
Name: indoor_temperature

```
04 Oct 2013 20:22:52.914  [MQTT Payload]
 /sensors/rhvWeather-01/indoor_temperature
75.2
04 Oct 2013 20:22:53.966  [MQTT Payload]
 /sensors/rhvWeather-01/indoor_humidity
41.3
04 Oct 2013 20:23:03.294  [MQTT Payload]
 /sensors/rhvWeather-01/sealevel_pressure
1016.2
04 Oct 2013 20:23:03.956  [MQTT Payload]
 /sensors/rhvWeather-01/indoor_temperature
75.2
04 Oct 2013 20:23:05.127  [MQTT Payload]
 /sensors/rhvWeather-01/indoor_humidity
41.2
04 Oct 2013 20:23:13.939  [MQTT Payload]
 /sensors/rhvWeather-01/sealevel_pressure
1016.2
04 Oct 2013 20:23:15.376  [MQTT Payload]
 /sensors/rhvWeather-01/indoor_temperature
```

For local integration, there will be nodes that connect to Smart Object event handlers directly and publish back to Observable Properties through a Node-RED Observer.

Roadmap for IoT Toolkit

IoT Toolkit is the reference implementation of the Smart Object API and related tools. The API is nominally complete and relatively stable. The roadmap going forward is to add useful features and create a developer release that can be installed as a Python library and hosted on Pypi.

Some examples of roadmap items are:

– Object model, Data model: create namespaces and ontologies

– Programming model UI: Node Red integration

– Graph-based resource access control

– Harden code, exception handling, API Tests

– Scalable server, multi-tenancy, TLS everywhere

– XMPP endpoint and proxy for XEP-0060, XEP-0323, XEP-0325

– Web UI for debug and demo, applications including Navigator, Dashboard, Analytics, Graphs and Charts