www.universal-devices.com

Object Oriented Approach to IoT Interoperability

IoT Semantic Interoperability Workshop 2016

Contents

1. Background

With the ever ubiquity, awareness, and public adoption of automation and Smart Building devices and solutions, and with the advent of robust and prevalent off-the-shelf communicating devices, IoT interoperability has become of utmost importance. From information model consumer perspective, a device driver must be developed for each “type” of a device since, currently, all information models are tied to specific instances of “type” such as lights, thermostats, sensors, etc. and with no abstraction/meta model layer. In this respect, with each variation of the same type, an updated driver is required to address the new features. In short, the current state of the art is ill equipped to handle dynamic and real-time/runtime binding, event handling, and interoperation between devices.

When discussing runtime binding, the main question is: do devices really need to have á priori knowledge of the types/classes of other devices with which they attempt to interoperate with? Or does it suffice for a device only to discover a set of permissible actions and properties supported by the other device?

This distinction is quite important since, if a device needs to have á priori knowledge of the types/classes of other devices, then, we reach an intractable situation where not only we would need to define all classes in the world but also we would have to provide extensibility and composition features to support any variations thereof. In and of itself, this paradigm will make it quite impractical, if not impossible, to have devices bind at runtime.

And, this is precisely the situation where we find ourselves in: a myriad of consortiums, alliances, and efforts that try to define the whole world, each differently, and even new vernaculars that are counterintuitive and ambiguous with no resemblance to reality. For instance, even though Color might be represented as a Class in implementation, from a purely semantics perspective, it’s an “adjective” since it is describing the characteristics of some other “thing”.

Object Oriented Analysis and Design has been around for decades and it has been successfully used in a variety systems and specifically Plug & Play device driver development. As in real life, Classes define generic types and Objects are specific instantiation of classes. Classes have behaviors and properties and may inherit from other classes (polymorphism). There is a clear correlation between objects and their properties to natural languages. At a high level:

Objects: Objects/Subjects in a sentence or phrase and corresponding to nouns

Properties: Nouns (they could be other objects) or adjectives

Behaviors: Verbs; actions that an object can do or allows to be done to it. In Service Oriented Architecture, these are called Services

Adverbs: Parameters for Behaviors

As such, behaviors are the permissible verbs (actions) that can be done to objects such as “query”, “turn on”, “turn off”, “send”, etc. properties, on the other hand, are the permissible adjectives or other objects that define the intrinsic characteristics of objects such as “name”, “color”, “level”, “cool set point”, “temperature”, “status”, etc.

In this respect, then, once we define the manner in which consumers of an object can query behaviors and properties of other objects, then it’s quite possible for them to “infer” what can be done to those objects and what can be expected from them at runtime (very similar to reflection). This said, however, for this methodology to work – and just like in any programming environment – we need a well-defined list of globally understood base types such as int, boolean, float, double, enum, etc. in addition to Units of Measure that describe values for those properties. Thankfully, NIST (along with other SDOs) have UOMs pretty well defined and it’s been quite static in nature.

2. Node Meta Model (NMM)

In order for devices to bind to each other at runtime, there needs to exist a well understood meta model that describes the behaviors and properties of each to the other. We shall call this Node Meta Model (NMM) simply to make a distinction between representation of nodes vs. data (which does not have any behaviors) and vs. objects (may make things confusing).

At the highest level, this model should be able to represent thematic relationships (i.e. who does what to whom) between nodes and, at the minimum, must comprise:

1. Services or Commands:
    a. What can you do to me? (e.g. change my set point, query, turn on/off, etc.)
    b. What can I send to you? (e.g. I was turned on, off, etc.)
    c. Can I listen to property change events from another?
2. Properties: what are my intrinsic characteristics? (e.g. cool set point; UOM=temp, on level; UOM=luminance, etc.)
    a. Can I publish property change events to others? (e.g. set point changed)
3. Compositions:
    a. Do I have any other nodes within me? (e.g. a multi sensor)

Optionally:

1. Node Type/Class: the class of the node itself is completely optional. i.e. it’s NOT necessary to know that a node is a Thermostat. If the type is given, then the consumer may use that information
2. Editors (default editors to represent properties, enumerations, etc.), Icons, and National Language Support
3. A repository for all the above: this may be important to support UOMs, editors, NLS, Icons at runtime but only works for Internet connected devices unless the repository can be included in computer operating systems or routers on the LAN

3. Example

This a very simple example that uses XML to define NMM for a thermostat. This definition is transport independent and can also be represented in JSON.

```
<nodeDef id="Thermostat" nls="143">
  <editors> … </editors>
  <properties>
    <st id="ST" editor="I_TEMP_DEG" />
    <st id="CLISPH" editor="I_CLISPH_DEG" />
    <st id="CLISPC" editor="I_CLISPC_DEG" />
    <st id="CLIMD" editor="I_TSTAT_MODE" />
    <st id="CLIHCS" editor="I_TSTAT_HCS" />
  </properties>
  <cmds>
    <sends>
      <cmd id="DON" />
      <cmd id="DOF" />
    </sends>
    <accepts>
      <cmd id="CLISPH">
        <p id="" editor="I_CLISPH_DEG" init="CLISPH" />
      </cmd>
      <cmd id="CLISPC">
        <p id="" editor="I_CLISPC_DEG" init="CLISPC" />
      </cmd>
      <cmd id="CLIMD">
        <p id="" editor="I_TSTAT_MODE" init="CLIMD" />
      </cmd>
      <cmd id="QUERY" />
    </accepts>
  </cmds>
</nodeDef>
```

<nodeDef>	id	Name of this node definition (e.g. “DimmerSwitch”)
	nls	NLS key string used to override names
<editors>		Editors scoped to this node
<st>	id	A predefined property e.g. “CLISPH” or a generic one
	editor	The id of the editor to use
<sends>		The commands this node can send out.
<accepts>		The commands this node accepts.
<cmd>	id	Name of a command.
<p>	id	Name of a command parameter. A command may have one unnamed parameter; all others must be named.
	editor	The id of the editor to use for this parameter
	init	(Optional) id of the status value this parameter should be initialized and/or synchronized with. For example, CLISPH is both a status and a command.