An Architecture for Unified Access to the Internet of Things

Jack Jansen

CWI, Amsterdam

<jack.jansen@cwi.nl>

Steven Pemberton

CWI, Amsterdam

<steven.pemberton@cwi.nl>

Abstract

The Internet of Things is driven by many tiny low-powered processors that produce data in a variety of different formats, and produce the data in different ways, sometimes on demand (such as thermostats), sometimes by pushing (such as presence detectors). Traditionally, applications have to be a mash up of accesses to devices and formats. To use the data in a cohesive application, the data has to be collected and integrated; this allows very low demands to be put on the devices themselves.

The architecture described places a thin layer around a diverse collection of Internet of Things devices, hiding the data-format and data-access differences, unifying the actual data in a single XML repository, and updating the devices automatically as needed; this then allows a REST-style declarative interface to access and control the devices without having to worry about the variety of device-interfaces and formats.

Keywords: Internet of things, iot, REST, XML, Software Architecture

1. Internet of Things

Moore's Law is about three properties of integrated circuits: the number of components on them, the price, and the size. Hold two of these constant, and the other displays its full Moore's Law effect: after one cycle, you can get the same thing for half the price, the same thing for the same price at half the size, or the same thing at the same size with twice the number of components.

So while computers have been getting more powerful at one end, and people have been optimising price/size/power in the middle, computers have been getting smaller, and cheaper at the other end. While a dozen years ago, you could count the number of CPUs in your house on one hand, now they are being added to everything, very often with networking built in.

The result is that there are now millions of cheap, tiny devices, with low processing power, embedded in devices everywhere.

The internet of things is not necessarily a new concept: twenty years ago a typical petrol station would have had embedded devices in the pumps, the storage tanks, the tills, the vending machines, and these would all have been centrally accessible and controllable. What is new is the ubiquity, and the diversity.

A problem that accompanies this diversity is a lack of standardisation. For instance, there are a number of different access methods, such as on demand, where you go to the device to access the data, push, where the device sends the data out at some point, and you had better be listening if you want to access it, and storage in the cloud, where the device either sends the data to some central point, or the central point polls the device for its values. Similarly, there are various data formats used, including XML, JSON, and a number of other text formats.

This all means that creating applications that combine data from different devices involves programming and dealing with lots of fiddly detail.

2. Social Problems

Apart from the technical problems, there are also some social problems involved, particularly since companies producing the devices like to keep control, with the possibility of monetising the data they have access to.

doi:10.14337/XMLLondon17.Pemberton01

The problems include ownership: since the data may be stored on a device in the cloud, who owns the data? Do you as owner of the device even have unrestricted access to your own data? There is also privacy: whether or not the data is stored on the device itself, who can access and see the data, and what are the access mechanisms for ensuring the data is not publicly visible? And then you have the issue of control: who is allowed to do anything with the data or device?

3. An Architecture

This paper describes an architecture and system based on it, that addresses these issues and that permits:

- retainment of ownership of the data,
- an access control mechanism to keep control over who may see and modify the data and devices,
- hiding the data-format and data-access differences by placing a thin layer around the diverse collection of devices,
- integration of the data into homogeneous collections,
- keeping the data and devices updated automatically as needed, without intervention.

The resulting architecure allows a REST-style declarative interface to access and control the devices without having to worry about the variety of device-interfaces and formats.

4. Design

The central element of the design is an XML repository that stores the incoming data. XML gives the advantage of data homogeneity, and an advanced existing toolchain, and the separation of elements and attributes facilitates the separation of data from metadata.

The essence of how the system works is that this repository is kept up-to-date with the devices bi-directionally: if the device changes, the repository is updated to match, and if the data in the repository is changed, the device is updated. This has been referred to previously, in contradistinction to WYSIWYG (What You See Is What You Get) as TAXATA (Things Are eXactly As They Appear) [1].

To achieve this, there is a thin functional layer around the repository that communicates with the devices. Plug-ins for devices and formats are responsible for knowing how to access the data from the devices, obtaining data from them, converting as necessary to XML, and for sending data back to the devices in their native format should the data change in the repository.

In support of this there are events that can be listened for and reacted on within the repository. The fundamental event is *value changed*, that signals when a data value changes, and allows event listeners to react; however, other events include timer events, and events signalling changes in the structure of a sub-tree, such as insertions and deletions.

Using the DOM model of events [2], events are allowed to bubble up the tree, so that listeners can respond at the value itself, or higher up the tree for a group of values.

Finally there are constraints and relationships, that specify how values relate to each other, and ensure that values that depend on others are automatically updated (in the process possibly changing the state of the related devices).

5. Some (necessarily simple) Examples

For instance there is a single bit: `lights`. When the lights are on, that bit is 1, and when the lights are off it is 0.

However, it works both ways: to turn the lights off, you just set the bit to 0; if anything changes the value to 1 they go on again.

There is another single bit: Is Jack home? (Which is not two-way ;-))

There are two ways to influence a value in the repository. One is equality "=", which ensures that the equality is *always* true.

For instance, if we said

```
lights = jack-home
```

this would mean that whenever Jack is home the lights are on, and whenever he isn't home, the lights are off. However, this would be upsetting if he wanted to sleep.

Consequently, we use the other method of influencing a value: "←". This only changes the value when the value of the expression changes. So, if we say:

```
lights ← jack-home
```

this would ensure the lights are on when he arrives home (they may have already been on), and ensures they are off when he leaves (they may already have been off).

Since this only happens when changes happen, it allows an override. For instance a switch on the wall also

has a bit in the database, and this can be bound to the value of the lights:

```
lights ← switch
```

(Note how switches are no longer hard-wired to their function.)

You probably don't want the lights to come on when it is already light, but you can have a sensor that detects whether it is dark or not, with an affiliated bit in the store:

```
lights ← jack-home and dark
```

This switches the lights on if Jack comes home in the dark; it switches the lights on if Jack is already home and it gets dark; and it also ensures that the lights are off when Jack leaves (whether they were on or not already, and whether or not it is dark). Note that this also ensures that the lights go off when it gets light.

Note that you could combine these statements into a single one using equality:

```
lights = (jack-home and dark) or switch
```

However, the separate statements allow a certain degree of modularity, since, for instance, if you decide to reassign the switch to another purpose, the other statements continue to work.

6. Is Jack Home?

How do we know if Jack is home?

Well he carries a mobile phone, that connects to the wifi, maybe a bluetooth watch that a sensor can pick up, and he has a laptop that also connects. These all get recorded in the repository (along with other details such as IP address assigned). So we could say

```
jack-home = jack-phone and jack-watch
            and jack-laptop
```

However, sometimes he switches his laptop off. How about:

```
jack-home = jack-phone or jack-watch or jack-laptop
```

Well, he might accidently or deliberately leave his phone or watch at home. Then we use a heuristic:

```
jack-home =
     count(jack-phone, jack-watch, jack-laptop) > 1
```

This is not absolutely failsafe, but likely to be satisfactory.

(For the purpose of exposition, we have treated `jack-home` as if it were a single standalone value, but in reality it will be part of a structured value, such as `person[name="jack"]/present`).

7. Living together

Jack doesn't live alone though. So there is a bit that records if anyone we know is home:

```
anyone-home = jack-home or jill-home or jim-home
```

and we would use that in preference to just `jack-home` in the above examples.

We can let the central heating automatically activate depending on whether someone is home or not:

```
heating = anyone-home
```

Of course, the required temperature of the heating is also a value in the database, as well as the actual temperature, so unlike the lights example, we don't need an extra override, since that is already taken care of.

8. Lock

One of the devices we have built is a door lock that is openable with any RFID device (a phone, a bank card, a dongle, etc) that has been registered with the lock.

Opening the lock is easy. If you swipe the RFID by the reader, the identification gets stored in the repository at `lock/request`, so the lock may be opened if that identification is in the list of allowed values:

```
lock/unlocked ← lock/request in lock/allowed
```

However, this only opens the lock. There are two options for relocking. One is if the lock is intended to be opened, and left open until it is locked again. Then you swipe a second time to lock it, and replace the above statement with:

```
lock/unlocked ← lock/unlocked xor
                (lock/request in lock/allowed)
```

Then a swipe just toggles the locked/unlocked state.

The other option is if the lock is opened with a swipe and then locks itself shortly after. For this we use timer events:

```
lock/unlocked ← lock/request in lock/allowed

changed(lock/unlock):
   dispatch(init, lock, 2sec)

init(lock):
   lock/request ← ""
   lock/unlock ← 0
```

Here we see a listener for the `value-changed` event on the lock. This dispatches an *init* event to the lock after 2 seconds. The listener for the init event relocks the lock.

9. Mesters's Law

One principle that we have applied in the project is Mesters's Law, named after its instigator:

"A Smart anything must do at minimum the things that the non-smart version does"

So, for instance, a thermostat that doesn't allow you to change the desired temperature at the thermostat itself, but requires you first to find the thermostat remote control does not fulfil Mester's Law.

To this end, the individual devices must have as few dependencies on the general infrastructure as possible. Clearly, there is nothing much you can do if there is a powercut and you have no backup power supply, but you don't want to depend on the wifi to be running, or the domain name server to be up, in order to be able to get in to your house.

What this means is that our system runs on the local devices as well, so that there are several copies of the system distributed, and communicating with each other: there is no dependency on a central version of the system being up and running.

10. Privacy

A basic principle is that none of the data is visible outside the system, unless explicitly revealed to someone.

This allows Jack, should he wish, to expose that he is home, without exposing details such as his phone identity.

For instance, he can reveal to the janitor of the building whether he is home, or reveal whether anyone is home to other inhabitants of the building without revealing any other details, such as his phone MAC address. This means the janitor can't also determine if Jack is at the bar down the road.

Since the architecture is primarily state-based, with events a side-effect, in effect it is the reverse of IFTTT-style systems that are quite common nowadays for IoT solutions [3].

The advantage of the state-based paradigm, together with the hierarchical containment that XML gives us, is that it supplies the scaffolding for the necessary security and privacy mechanisms. Doing fine-grained access control in an event-based system like IFTTT would be more difficult, because there would basically have to be access rules for every event/trigger, but with this system it can be done on the basis of subtrees.

As mentioned earlier, a design decision was to store data in XML elements and use attributes for storing meta-data. Part of that meta data is information about access.

While this part of the system is not yet implemented, we are investigating two possible access mechanisms: one, based on ACLs (access control lists) [4] mirrors the system that is used in hierarchical filestores, such as Unix, that are based on user-indentities. However, the one that has our current preference is a system based on Capabilities [5], where to access a part of the structure you have to be in possession of a token.

11. Communication

Since the system is state-based, the ideal communication method is REST.

REST (REpresentational State Transfer) is the architectural basis of the web. As Wikipedia points out:

"REST's coordinated set of constraints, applied to the design of components in a distributed hypermedia system, can lead to a higher-performing and more maintainable software architecture."

In other projects we actually have proof of this claim: we have seen it save around an order of magnitude in time and costs.

Therefore communication with the system, and between instances of the system is HTTP/HTTPS.

12. User Interface

Since all actions are now controlled by changing data, all we need is a straightforward way to access and change data.

Luckily we have XForms [6], which has more or less the same structure as the system: a collection of (XML) data, events, and constraints.

On top of that, XForms has a method of binding user interface controls to the data for displaying and updating the data.

This has been treated in some details in an earlier paper "XML Interfaces to the Internet of Things" [7].

13. Conclusion

We have a system that insulates us from the details of the different devices, how to drive them and the format of the data. It offers a powerful security mechanism, and straightforward access protocol.

It gives us a very simple yet powerful mechanism for reading and controlling devices.

The system is at an early stage of development at present: we currently have a system running at two locations.

Bibliography

[1] *The ergonomics of computer interfaces*. designing a system for human use. Lambert Meertens and Steven Pemberton. 1992. Centrum Wiskunde and Informatica (CWI).
http://www.kestrel.edu/home/people/meertens/publications/papers/Ergonomics_of_computer_interfaces.pdf

[2] *Document Object Model (DOM) Level 2 Events Specification*. Tom Pixley. World Wide Web Consortium (W3C). 13 November 2000.
http://www.w3.org/TR/DOM-Level-2-Event

[3] *IFTTT (If This Then That)*.
https://en.wikipedia.org/wiki/IFTTT
Wikipedia. Accessed: 13 May 2017.

[4] *Access Control List*. Wikipedia. Accessed :13 May 2017.
https://en.wikipedia.org/wiki/Access_control_list

[5] *Capability-based Security*. Wikipedia. Accessed: 13 May 2017.
https://en.wikipedia.org/wiki/Capability-based_security

[6] *XForms 2.0*. Erik Bruchez, Steven Pemberton, and Nick Van den Bleeken. World Wide Web Consortium (W3C).
https://www.w3.org/community/xformsusers/wiki/XForms_2.0

[7] *XML Interfaces to the Internet of Things*. Steven Pemberton. XML London 2015. June 2015. 163-168.
doi:10.14337/XMLLondon15.Pemberton01