IoT – A Software Developer‘s Perspective

Designing scalable architectures for reuse in Embedded Software Engineering

by Ian Macafee – Principle Consultant - EVOCEAN

The term IoT has become widely used in recent years. Google dictionary defines it as:

Internet of things - the interconnection via the Internet of computing devices embedded in everyday objects, enabling them to send and receive data.

As a software developer I produce code that arguably receives and sends data. So, what, if anything, do I need to change to satisfy the IoT?
Perhaps you don't see a need to change. The logic of your code is the valuable part. How it receives and sends the data could be regarded as someone else's problem. At a concept level, this is true. You shouldn't have to change the logic of your functions but... perhaps there are ways of making your implementation easier to ***thingify***. Perhaps there are techniques and tools that you are yet to discover?

So, what do we mean by ***thingify***?

To understand the IoT context, as developers, we need to understand its mechanics. Given this understanding we can move forward knowing **why**, we should, and, **how to**, adjust. The purpose of this paper is to convey my observations of the past few years whilst working in this rapidly growing distributed software development world. I have witnessed how some companies find their way whilst others lag. I have noticed how many of the lagging companies have one thing in common. They have lots of useful IP buried in existing systems but are struggling to know how best to expose it to enable innovation. By breaking up systems into the component parts, i.e. things, they can be wired together in new ways. An organisation then has the possibility to discover new capabilities and new products without having to implement the details. Details take time and expertise. Wiring things is easy if I have things I can wire!

I have helped to influence thinking in several organisations and witnessed the injection of energy a new perspective can provide. I wish to pass on these experiences to help others catch up and reap the benefits both outside and within their organisations.

Mostly I am focused on helping to make software development fun, simple and effective.

About the author

Ian Macafee has more than 25 years of experience in software development, support, consulting and UML training. Since 2016 he is working as a Principal Consultant for EVOCEAN. His specific interests include behavioural modelling with Rhapsody and distributed architectures.

Thingification

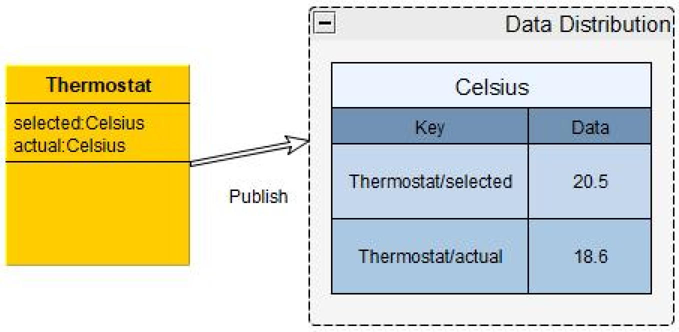
I think of IoT as a set of runnable functions that communicate through a **distributed database**. So, I regard a thing as a function built as a single executable and accessible through a database. From the point of view of its output this seems straight forward enough, the function puts data into the database and anything with access to that database can see the data.

I use the term database to refer to a set of tables that hold data independent of the function that creates that data. The data may or may not persist. If a function wishes the data to persist it would specify the data as **retained**. The length of retention may be dependent on factors such as the nature of the data and lifetime of the function.

Writing to a distributed database is commonly referred to as ***publishing***.

Publishing

Take, for example, a thermostat. It can be regarded as a function. It may have two attributes: the temperature that a user has selected for activation purposes; and the actual temperature it is sensing.



To organise the data in the database we have tables. The tables contain rows of data of the same type. In the example, our type is Celsius. A published temperature will be inserted into our table. But there may be multiple things inserting temperatures into our table. We need a way of organising our database such that we understand what each row of a table represents. A key.

Rather than using the term key, each individual row in our database is commonly referred to as a ***topic***.

You should think of a topic in a similar way to a folder structure. Topics can have hierarchy. If there is more than one thermostat in our system, then we will want to uniquely identify each one. We may have, for example, a thermostat in both the lounge and the kitchen.

Kitchen/Thermostat/selected
Kitchen/Thermostat/actual
Lounge/Thermostat/selected
Lounge /Thermostat/actual

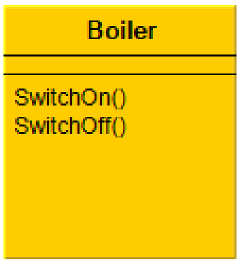
In this case our Celsius database table would have 4 rows, keyed as seen here on the side:

A consistent topic naming scheme is as important as how we would decompose our systems, organise our repositories and file systems.

Subscribing

Publishing data into a database is a simple idea to understand. However, the strategy for reacting to changes in data is an area that needs more thought.

For us, as developers, to avoid coupling our functions unnecessarily we simply provide methods, aka operations|triggers|receptions, in our functions that may be called by any user. Some carry parameters which we might consider as data, others simply trigger functionality to run.



Here the SwitchOn/Off methods would not naturally be regarded as data. Nevertheless, data ultimately triggers all methods. In some cases, time is the data, in our example a change in selected or actual could be used to SwitchOn our boiler.

```
If (actual < selected) then Boiler.SwitchOn()
```

The job of reacting to the change in thermostat attributes and evaluating the condition is the responsibility of neither the boiler nor the thermostat itself. In fact, it should not influence the design of either. It is simply a wiring job to connect the output of one thing to the input of another.

So, I am finally getting around to what our things should subscribe to.

Put simply, a thing need only subscribe to the inputs defined by the context of that thing.

To do this, it must subscribe to a topic that identifies it as the owner.

In the case of our boiler, located in the garage, for example, those topics would be:

Inputs/Garage/Boiler/switchOn
Inputs/Garage/Boiler/switchOff

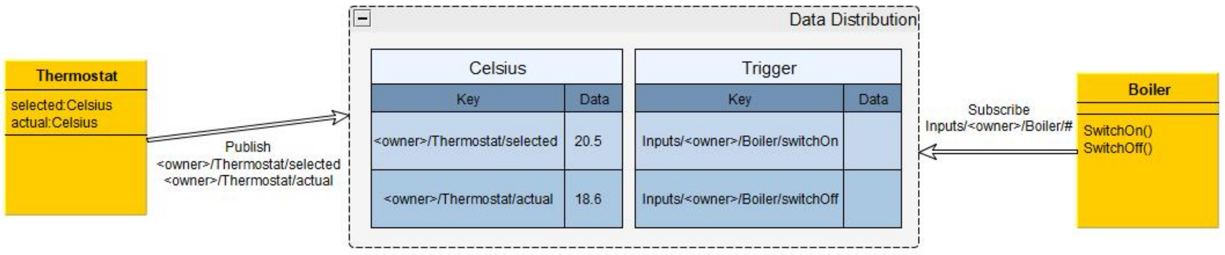
Perhaps there is also a thermostat in the garage. The garage system could subscribe to all its inputs via a single wildcard subscription to:

Inputs/Garage/#

N.B. Similarly it is possible to subscribe to all thermostat data using a level wildcard:

+/Thermostat/#

So, now we have two standalone software executables, boiler and thermostat. Both can be implemented to subscribe to their inputs and publish their outputs without knowledge of the systems in which they are deployed.



Celsius

Key	Data
<owner>/Thermostat/selected	20.5
<owner>/Thermostat/actual	18.6

Trigger

Key	Data
Inputs/<owner>/Boiler/switchOn	
Inputs/<owner>/Boiler/switchOff	

To build our, somewhat artificial, system we need to:

1. Instantiate 1 boiler and 3 thermostats and uniquely identify them
2. Wire them together with some additional logic to create the desired system behaviour
    - Our boiler will need to SwitchOn if any thermostat is showing (actual < selected) as true and this can be built into our wiring logic

guiding you beyond

www.rhapsody.expert

Rhapsody + TXF (Thing eXecution Framework)

For many years, in fact more than 15, I have been using Rhapsody to develop software. I find it especially useful because it provides the ability to simplify complex behaviour by utilisation of state charts. When communication between objects is required the OXF (Object eXecution Framework) is perfectly suited to running an application on a real-time operating system. The actual operating system is abstracted through a layer known as the OSAL, Operating System Abstraction Layer. This makes it possible to port an application across different operating systems with minimum, if any, rework.

In recent years I have required to develop larger and more complex systems. These systems have had some aspect of distribution across processors. On several occasions I have had to use more than one programming language to produce my overall system in the most efficient way.

All of this has meant I have experimented with different approaches to inter-process communication. Shared memory, TCP, pipes, UDP, serial comms to name but a few. Then I moved on to middlewares. DDS (Data Distribution Service) was my first experience. It worked for the specific problem being solved at the time but had a steep learning curve and put restrictions on the way I wanted to think about distributed applications. DDS's means of communication is based on the publish subscribe paradigm I have suggested in this paper. It is a candidate for a suitable IoT implementation and is used by many. However, I also discovered MQTT. It is my current middleware of choice. MQTT puts fewer constraints on the application that uses it.

Regardless, I am aware that, just as settling on a specific operating system is often a mistake for progression of a business, settling on a specific middleware is likely to result in a problem down the line when something better comes along. However, I have settled on the publish subscribe paradigm. I think it fits into real life. I like things I can explain in terms that make sense to everyone. If I speak (publish some data) you can choose to listen (subscribe) and react in whatever way suits you.

I have therefore created a layer of software I call the Thing eXecution Framework (TXF). Which is based on the publish subscribe paradigm and has a Middleware Abstraction Layer built into it.

By simply utilising a library and a set of stereotypes a competent Rhapsody user can easily thingify his application using the TXF. He can then build his executable to include the chosen middleware adapter.

MQTT

http://mqtt.org/

As mentioned above MQTT would be my recommendation to anyone thinking about how to move towards a standard protocol for the IoT.

A simple google of *MQTT* opens an expanse of tools, implementations, and knowledge.

Node-RED

https://nodered.org/

Node-RED is a free tool for wiring together the internet of things, licensed under the Apache Licence. Node-RED was originally developed by IBM's Emerging Technology Services team and is now a part of the JS Foundation. It is a tool of simplicity that provides endless opportunities.