A Taxonomy of IoT Client Architectures

Taivalsaari, Antero

2018-05-04

Taivalsaari , A & Mikkonen , T J 2018 , ' A Taxonomy of IoT Client Architectures ' , IEEE Software , vol. 35 , no. 3 , pp. 83 - 88 . https://doi.org/10.1109/MS.2018.2141019

http://hdl.handle.net/10138/321643
https://doi.org/10.1109/MS.2018.2141019

acceptedVersion

A Taxonomy of IoT Client Architectures

Antero Taivalsaari[1], Tommi Mikkonen[2]

[1]Nokia Technologies, Tampere, Finland
[2]University of Helsinki, Helsinki, Finland
antero.taivalsaari@nokia.com, tommi.mikkonen@helsinki.fi

Abstract. In this article we define a taxonomy of software architecture options for IoT devices derived from our industry projects, starting from the most limited sensing devices to high-end devices featuring full-fledged operating systems and developer frameworks. We learned that there is a plethora of architectural options for IoT devices, offering very different levels of software development capabilities. These capabilities can have a significant impact on the overall end-to-end architecture and topology of IoT systems.

Keywords: IoT, Programmable World, Software Architecture, Software Platforms

1 IoT Systems – the Big Picture

At the technical level, the Internet of Things is all about *turning physical objects and everyday things into digital data products and services* – bringing new value and meaning by making previously lifeless things more intelligent. Effectively this means adding computing capabilities and cloud connectivity to hitherto unconnected devices, as well as adding backend services and web and/or mobile applications for viewing and analyzing data and controlling those devices.

IoT systems are *end-to-end* (E2E) systems that consist of a number of architectural elements that tend to be pretty much identical in all IoT solutions – Devices, Gateways, Cloud platform and Applications [1]. *Devices* are the physical hardware elements that collect sensor data and may perform actuation. *Gateways* collect, preprocess and transfer sensor data from devices, and may deliver actuation requests from the cloud to devices. *Cloud* platform – usually offered as a Software-as-a-Service solution – has a number of important roles, including data acquisition, offline analytics, and device management and actuation. *Applications* range from simple web-based data visualization dashboards to highly domain-specific mobile apps.

There exists a wide spectrum of software architecture options for IoT devices, ranging from very simple, limited sensing devices to devices featuring full-fledged operating systems and developer APIs. In this article we define a simple taxonomy of these options based on number of industrial and academic IoT development projects (e.g., `https://health.nokia.com/es/en/steel-hr`, `https://wiki.mozilla.org/Connected_Devices/Projects`) carried out in the past four years.

2 IoT Devices – Software Architecture Options

There are various design drivers and tradeoffs in the context of IoT systems. Such factors include cost, update capabilities, dynamic programmability, security, energy efficiency, and communication latency. These factors largely determine the architectural options to follow [2]. At the high level, the software architecture choices for IoT client devices can be summarized as follows, ranging from simple to more complex architectures:

1. *No OS architecture*: for simplest sensing devices that do not need any operating system at all.
2. *RTOS architecture*: for slightly more capable IoT devices that benefit from a real-time operating system (e.g. FreeRTOS).
3. *Language runtime architecture*: for simple devices that require dynamic programming capabilities (e.g. JavaScript).
4. *Full OS architecture*: for devices that are capable enough to host a full operating system (typically some variant of Linux).
5. *App OS architecture*: for devices that are designed specifically to support third party application development (typically Android or Android Wear).
6. *Server OS architecture*: for devices that are capable enough to run a server-side operating system stack (typically Linux + Node.js).
7. *Container OS architecture*: for high-end IoT devices that are powerful to host a virtualized, container-based operating system stack such as Docker or CoreOS rkt.

The options have also been depicted and summarized in Figure 1.

2.1 No OS Architecture

The vast majority of today's IoT devices are really simple – smart light bulbs, thermostats, remotely controlled electricity plugs, air quality sensors, or ID tags or badges do not require complex software stacks.

In such simple IoT devices, there is no need for an operating system or an application platform at all. All the software is written specifically for the device, and software development is typically carried in-house. Hence, there is no need for third-party developer support. Support for firmware updates may be limited or non-existent.

Given the fixed nature of software in these types of low-end devices, the amount of RAM and Flash memory can be kept minimal. In many cases, only a few kilobytes or tens of kilobytes of RAM will suffice.

For battery-operated low-end devices, optimization of network communication plays a major role. Communication protocols such as MQTT, LWM2M and CoAP are important, while more capable devices tend to utilize HTTP-based communication and more verbose data formats such as JSON or XML.

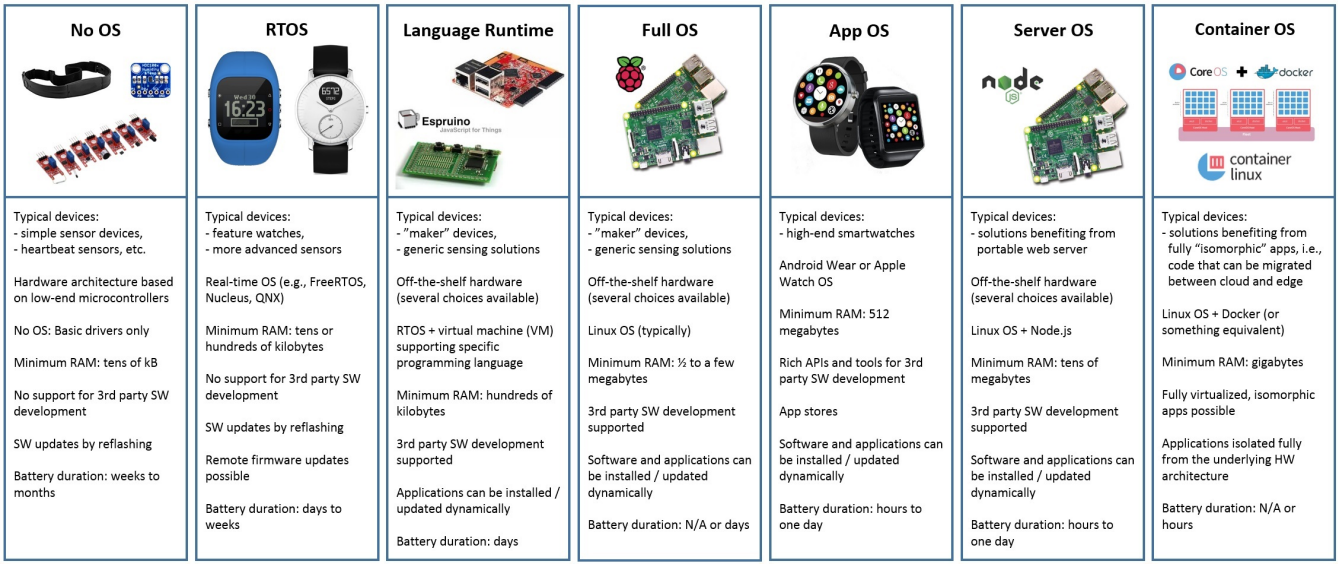

Fig. 1. Basic Software Architecture Choices for IoT Devices.

2.2 RTOS Architecture

For slightly more capable devices supporting a richer set of sensors, a *real-time operating system (RTOS)* may be beneficial. Popular open source and commercial real-time operating systems provide convenient developer toolkits and a basic set of APIs supporting second-party software development. They also provide built-in support for important product features such as secure firmware updates.

Software development for RTOS-based IoT devices is usually carried out in-house, since such devices do not typically provide any third-party developer APIs or the ability to reprogram the device dynamically (apart from performing a full firmware update). Typical development languages for RTOS-based devices are C or C++, although even assembly code might be used in some areas.

The memory requirements of RTOS-based architectures are comparable to No OS architectures, often necessitating as little as a few tens of kilobytes of RAM and a few hundred kilobytes of Flash memory. Devices in this category are often battery-operated, thus placing a lot of requirements on optimizing network connectivity and energy consumption more broadly.

2.3 Language Runtime Architecture

In addition to RTOS-based software stacks, there are IoT development boards that provide support for a *specific built-in language runtime or virtual machine (VM)*. For instance, popular *Espruino* (`https://www.espruino.com/`) or *Tessel 2* (`https://tessel.io/`) IoT development boards provide built-in support

for JavaScript applications, while Pycom's *WiPy* boards (`https://pycom.io/development-boards`) enable Python development.

Compared to No OS or RTOS solutions, language runtime based IoT devices are significantly more capable in the sense that they can support third-party application development and dynamic changes, i.e., updating the device software (or parts thereof) dynamically without having to reflash the entire firmware.

At the conceptual and technical levels, language runtime based IoT devices are very similar to early mobile application development platforms such as the *Java™2 Micro Edition (J2ME)* platform, where a dynamic language runtime serves as the portable execution layer that enables third-party application development and the creation of developer-friendly application interfaces. Such capabilities leverage the interactive nature of the dynamic languages, allowing flexible interpretation and execution of code on the fly, without compromising the security of the underlying execution environment and device. Basically, applications run in a *sandbox* that provides only a limited access to the underlying platform features.

At the implementation level, language runtime based IoT devices typically have an RTOS underneath. In that sense, these devices can be seen as the next evolutionary step up from devices built on the RTOS architecture.

The technical capabilities and memory requirements of devices based on language runtime architecture vary considerably based on the supported language(s). The size and complexity of the virtual machines also varies considerably, and thus the minimum amount of Flash or ROM memory can also range from a few tens of kilobytes to several megabytes. However, storage memory is so inexpensive nowadays that those memory prices have only a marginal impact on the total cost of a device.

2.4 Full OS Architecture

The next level up from the language runtime architecture are IoT devices that are powerful enough to run a full (typically Linux-based) operating system. The Raspberry Pi 3 device is a great example of such a device.

The presence of a full operating system brings a lot of benefits, such as built-in support for secure file transfers, user accounts, device management capabilities, security updates, very mature development toolchains, and numerous other features. The generic nature of devices supporting Full OS architecture also makes it possible to effortlessly run various types of third-party applications and services, including the aforementioned language runtimes for different programming languages.

Compared to low-end No OS or RTOS architectures, the memory and CPU requirements of Full OS stacks are significantly higher. For instance, the desire to run a Linux-based operating system in a device bumps the RAM requirements from a few tens or hundreds of kilobytes (for an RTOS-based solution) to half a megabyte at the minimum. The significantly higher energy consumption requirements make it difficult to use such devices in use cases that require battery

operation – except in tablet- or laptop-sized solutions with a battery capacity of at least a few thousand milliampere hours (mAh).

2.5 App OS Architecture

At the current high end of the IoT device spectrum, there are wearable device platforms such as *Android Wear* (`https://www.android.com/wear/`) or *Apple watchOS* (`https://www.apple.com/watchos/`) that are in many ways comparable to mobile phone application platforms from 3-5 years ago. These wearable device platforms provide very rich platform capabilities and third-party developer APIs – however, they also bump up the minimum hardware requirements considerably. For instance, the minimum amount of RAM required by Android Wear and Apple watchOS is half a gigabyte (512 MB) – over 10,000 times more than the few tens of kilobytes of RAM required for simple IoT sensor devices (!).

The processing power requirements of App OS devices are also dramatically higher than in simplest microcontroller-based IoT devices. Typically, an ARM Cortex-A class processor is mandated (for instance, an ARM A7 processor running at 1.2 GHz is stated as the minimum requirement for Android Wear currently), limiting maximum battery duration to a few days, or only to a few hours in highly intensive use.

2.6 Server OS Architecture

Much to nearly everybody's surprise, JavaScript surpassed the other programming languages in popularity in 2016[1]. While JavaScript was originally designed in the mid-1990s as a simple scripting language for the web browser, in recent years its use has rapidly spread into various other areas. The current success of JavaScript can be attributed especially to the *Node.js* ecosystem (`https://nodejs.org/`) that has popularized the use of the JavaScript language also in server-side development, thus turning JavaScript into *lingua franca* for web development from client to cloud.

The popularity of Node.js has created interest in IoT devices that are capable of hosting a web server. For instance, the earlier mentioned Tessel 2 board is capable enough to run the Node.js stack, and even serve as a standalone web server. Similarly, Raspberry Pi devices are also commonly used for running the Node.js stack and other web servers.

By default, Node.js assumes the availability of at least 1.5 GB of RAM. However, Node.js can be configured to operate with considerably smaller amounts of memory, starting from a few tens of megabytes. In addition to (or instead of) Node.js, there are several other web server offerings that are more tailored to embedded environments.

[1] `https://thenewstack.io/javascript-popularity-surpasses-java-php-stack-overflow-developer-survey/`

2.7 Container OS Architecture

Container-based software architectures have recently become very popular especially in cloud backend development [3]. A *container* is a standalone, portable, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries and settings. Popular implementations include Docker and CoreOS rkt.

Containers isolate applications from one another and the underlying operating system infrastructure, while providing an added layer of protection for the application. This guarantees that the software will always run the same way regardless of its physical execution environment.

At the technical level, containers are effectively a lighter-weight operating system virtualization mechanism. In contrast with operating system VMs such as VirtualBox or VMware Workstation, containers do not virtualize a complete guest operating system but share the underlying operating system with other containers.

Given the independence of the physical execution environment that containers can provide, containers are a very attractive concept also for IoT development, especially in light of the current technical diversity of IoT devices. Thus, although container technologies add considerable overhead compared to traditional binary software, their use has already started also in the context of IoT devices. For instance, the use of Docker on Raspberry Pi devices is already possible (see, e.g., `https://www.raspberrypi.org/blog/docker-comes-to-raspberry-pi/`).

In general, from a purely technical viewpoint container-based architectures are definitely a viable option for IoT devices if adequate memory and other resources are available [3]. At the minimum, the host environment must typically have several gigabytes of RAM available, thus making this approach unsuitable for the vast majority of today's IoT devices. Although container-based IoT devices may seem excessive today, we see them as an important step towards fully isomorphic IoT system architectures that we will discuss below.

3 Trends and Observations

To summarize, our work at Nokia and Mozilla has revealed that there is a range of software architecture options and stacks for IoT devices, depending on the expected usage, power budget, and the need to support dynamic programming and/or third-party development. Table 1 provides a condensed summary of the software architecture options for IoT devices. In general, the more capable the underlying execution environment is, the more feasible it is to run various types of software architectures, platforms and applications on it.

Software architecture choice is impacted heavily by energy consumption requirements. In practice, one of the most significant differentiating feature driving or even dictating the selection of the software architecture in the majority of IoT devices is the *battery*. A battery-operated IoT device typically has strict minimum operating time requirements. Furthermore, the form factor

Table 1. High-Level Comparison of Software Architecture Options

Feature	**No OS / RTOS**	**Language VM**	**Full OS**	**App OS**	**Server OS**	**Container OS**
Minimum required RAM	Tens of kilobytes	Hundreds of kilobytes	A few megabytes	Hundreds of megabytes	Tens of megabytes	Gigabytes
Typical communication protocols	Constrained (MQTT, LWM2M, CoAP)	Constrained (MQTT, LWM2M, CoAP)	Standard Internet protocols (HTTP, HTTPS)	Standard Internet protocols (HTTP, HTTPS)	Standard Internet protocols (HTTP, HTTPS)	Standard Internet protocols (HTTP, HTTPS)
Typical development language	C or assembly	Java, JavaScript, Python	C or C++	Java, Objective-C, Swift	JavaScript	Various
Libraries	None or System-specific	Language-specific generic libraries	OS libraries, generic UI libraries	Platform libraries	Node.js NPM modules	Various
Dynamic SW updates	Firmware updates only	Yes	Yes	Yes	Yes	Yes
Third-party apps supported	No	Yes	Yes	Yes	Yes	Yes
Isomorphic apps possible	No	Yes	Usually not	Yes	Yes	Yes

– the physical size and shape of a piece of computer hardware – characteristics of the device play a significant role in determining the right tradeoffs, hence impacting also the type of software architecture that the device can support.

The availability of inexpensive off-the-shelf hardware is driving the industry towards "overly capable" IoT devices. Interestingly, the recent emergence of inexpensive IoT chips, development boards and "maker" kits is driving the industry towards IoT devices and solutions that have "too much" processing power and memory for the actual needs. Given the availability of low-cost off-the-shelf hardware, it may often be simpler and a lot more affordable to buy stock hardware instead of building custom HW solutions. Furthermore, the extra capacity can be beneficial, e.g., for improved security features.

IoT devices are bringing back the need for embedded software development skills and education. An interesting observation is related to software engineering education. Software development for IoT devices is very similar

to "classic" embedded systems development, and is thus bringing back the need for embedded, small memory software development skills [4]. This is in contrast with recent software industry survey reports that emphasize the importance of higher-level programming skills (see, e.g., [5]).

Isomorphic IoT systems will emerge. Earlier in this paper, we noted that software containers and virtualization technologies are becoming available also in IoT devices. We believe that within the next 5-10 years, this will lead the industry to *isomorphic IoT system architectures* in which the devices, gateways and the cloud will have the ability to run *exactly the same software components and services*, allowing flexible migration of code between any element in the overall system. In an isomorphic system architecture, there does not have to be any technical differences between software that runs in the backend or in the edge of the network. Rather, when necessary, software can freely "roam" between the cloud and the edge in a seamless, liquid fashion.

Along the way towards isomorphic systems, edge computing will play an increasingly important role. Given the rapidly increasing computing and storage capacities of IoT devices, it is clear that in the future computation and intelligence will be increasingly balanced between the cloud and the edge (IoT devices and gateways). This can be very beneficial, since the ability to preprocess data in IoT devices (and gateways) allows for lower latencies and can also significantly reduce unnecessary data traffic between the devices and the cloud. Together with the emergence of mesh networking and low-power wide area networking (LPWAN) technologies, edge computing can be expected to significantly alter the topologies and the overall software architecture of IoT systems.

Interoperability is still a major issue. Today, the majority of IoT systems rely on the expectation that devices will only work with their 'own' cloud backend. Similarly, the most common way to use a device is via a specific application that is associated with one particular vendor's devices only. Even though there has been significant convergence in the past few years, we are still several years away from universal Programmable World standards as envisioned by Wasik [6] and discussed in our previous IEEE Software paper [1].

4 Conclusions

This article has presented a taxonomy of software architecture options for IoT devices, starting from the most limited sensing devices to high-end devices featuring full-fledged operating systems and developer frameworks. Although the vast majority of IoT devices today have very simple software stacks, we foresee the overall software stack complexity increasing rapidly because of hardware evolution and the general desire to support edge computing, software containers and isomorphic system architectures.

References

1. Taivalsaari, A., Mikkonen, T.: A Roadmap to the Programmable World: Software Challenges in the IoT Era. IEEE Software **34**(1) (2017) 72–80
2. Spinellis, D.: Software-Engineering the Internet of Things. IEEE Software **34**(1) (2017) 4–6
3. Celesti, A., Mulfari, D., Fazio, M., Villari, M., Puliafito, A.: Exploring Container Virtualization in IoT Clouds. In: 2016 IEEE International Conference on Smart Computing (SMARTCOMP), IEEE (2016) 1–6
4. Weir, C., Noble, J.: Small Memory Software: Patterns for Systems with Limited Memory. Addison-Wesley (Software Pattern Series) (2000)
5. VisionMobile: Developer Economics State of the Developer Nation, Q1 2016. `http://www.visionmobile.com/reports/developer-economics-state-developer-nation-q3-2016` [Online; accessed 18-November-2017].
6. Wasik, B.: In the Programmable World, All Our Objects Will Act as One. Wired (2013)