13 Middleware for the Internet of Things

Rahav Dor and Chenyang Lu
Washington University in St. Louis

The *Internet*, the network of networks, was not only a technological change. It affected human minds and changed how we operate as a collaborative society. In tandem with changing how we live, it affected how we think. We grew accustomed to sharing data and information that was once considered private, propriety, or of no interest to others. This level of connectivity and wealth of information enabled a long list of useful applications and services that made the Internet, our lives, what they are today. Another technological wave is coming and it may be larger in terms of connectivity and novel applications we will be able to build. The human endeavor is about to embark on the next disruptive technology – the *Internet of Things* (IoT).

We say that the IoT will be larger because, while the Internet connected people to information (predominately produced by people), the IoT will connect people and machines to data produced by everyday objects, and there are many more everyday objects than people. By and from inception the IoT will enable machines to talk to other machines and produce new data streams. Possibly the cardinal difference is that machines and people alike, will not only consume data but will be able to control and activate everyday objects. When the IoT achieves popularity it will surpass the Internet along the following key dimensions.

Scale The IoT will connect far more sources of data and physical everyday objects. This would produce more data streams, raw or refined.

Participants Autonomous machines and software processes, along with people, will natively participate in interactions.

Closed-loop Data will be comprehensible by people and machines and this will enable either or both to make decisions and command objects.

Diversity The objects are excessively heterogeneous.

The IoT will give machines and people alike access to data and myriad of everyday objects at a greater scale than the Internet does. In the IoT lingo these objects are named *Things*. Together with the networks and applications that connect Things, the Internet of Things (IoT) will be formed. Our toothbrush, the dog's water bowl, temperature sensors, motion sensors, our homes, roads, buildings, cities, and the world – will be producing data that someone, and something, can consume. Moreover, the IoT is bi-directional and will allow machines and people to issue commands to (actuate) connected Things. Applications (apps) would be composed and be informed by data emanating from Things, and these apps would actuate our world through Things. The impetuous for creating this emerging technology is to better our lives through the IoT.

13.1 Motivation for an IoT middleware

This chapter is focused on an enabling technology that is essential to realize the IoT. The world is full of Things, diverse in their computing and networking faculties. Many Things not conceived with such faculties and those will be bolted onto them, or remotely sensed to become a part of the IoT ecosystem. This presents substantial variety in what can be sensed or actuated, and in how communication can be accomplished. How could we possibly inter-connect such diverse number of Things, computer programs, in-network processes, and data of varied structure and amount? How can we enabled application developers to compose smart-apps using the world as a data source, without requiring expertise with the intricacies and oddities of each and every sensor, actuator, or data stream? The answers critically reside with the technology discussed in this chapter.

We view the IoT as a Cyber Physical System (CPS) on a global scale where everyday *physical* Things are enhanced with *cyber* resources (sensing, computing, networking, and actuating). In this CPS, in-network processes can perform computation on data streams as the data traverses the network, and emanate additional streams and services. This CPS extends to the cloud where historical data and additional software processes and services run.

In this chapter we provide an overview of one technology that will be important in enabling this vision – a software layer that will help app developers interface with such myriad of Things. Such software layer is generically referred to as *middleware*. It is used in many contexts and for many purposes. A middleware abstracts low-level details of what lies below it (here Things, computers, streams, services, and the network) and provides useful services to what lies above it (here apps). The useful services of our middleware can be for example finding and communicating with Things in a uniform way. This is of added-value because most IoT apps will want to connect to many Things, so considerable design and coding efforts will be duplicated unless something provides that functionality for all. We will introduce three different middleware, each with its unique approach, advantages, and disadvantages.

It will require more than a book chapter to host a complete representation of competing middleware that could power the IoT. In the struggle between providing meaningful information about a few and a high level survey, we choose the former. Our selected three allow us to discuss middleware systems with very different architecture, or even philosophies, and in doing so we provide the reader with what we hope is a broad perspective. This choice should not suggest that other technologies, such as the work of ([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]) are less likely to find their place in the IoT.

The middleware we choose are. The *AllJoyn* framework [13], which follows the peer-to-peer (P2P) network model and uses Remote Procedure Call (RPC) from one peer (app) to another to invoke services or change state. In a P2P network, participating peers share tasks and perform work for each other. RPC is an inter-process communication technique that allows one process to invoke services provided by another process, where the developer is not required to deal with the details of the remote invocation. AllJoyn services are stateful and interactions occur within sessions that apps join. This architecture is of value for apps that need some sort of continuity as time progresses or a state of the world due to past interactions. It also allows AllJoyn to manage network dynamics such as device disconnection from wireless interference or mobility. We will see though, that this architecture choice makes the framework challenging to run on resource constrained Things.

CoAP is a standard [14] by the Internet Engineering Task Force (IETF). It is developed around the idea that since Internet technologies and standards are so successful they would be as fruitful for enabling the IoT. Importantly, where CoAP is adapted, seamless Internet-IoT interoperation will be enabled. CoAP changes Internet technologies to fit the requirements of resource constraint Things by defining compact packet structure to lower computation demands and network bandwidth. It depends on the unreliable UDP transport for the same reason, and it touts the HTTP/REST architecture. Thus CoAP is stateless and fits well onto resource constraint Things because hosts do not need to maintain state in memory. This design though, is not the best fit for apps whose future behaviour depends on the past and they would like to delegate hysteresis management to the server.

The Building Operating System Services (BOSS) [15] provide operating system abstractions for Things in smart buildings. As an operating system we will see that BOSS provides services surpassing AllJoyn and CoAP. Depending on the app that one intend to build these services may be a blessing or introduce unneeded overhead.

13.2 The services we may want from a middleware

We can begin to appreciate why would we want a middleware by asking what added-value services we may want it to provide? We would like to identify services that have little to do with apps core functionality and are general enough that most IoT app would need. A well designed middleware will allow app developers to author only their app core logic and delegate all other operations to the middleware.

We start by noting that a good middleware will “wrap the IoT” by abstracting the diversity of Things within it and provide easy access to the resources inside. Since we view the IoT as a Cyber-Physical System the middleware should provide access to both physical Things such as sensors and actuators, and cyber Things such as services and data that the network and other apps makes available.

Consider as an example an app we wish to author, named *OurApp*, which need the temperature reading at a location. Suppose that the temperature can be provided by a local public sensor *tSensor*, or by an app *EnvApp* that already runs in the network. *EnvApp* is designed to use *tSensor* when it is functioning, accessible, and not busy with other activities. It can also ask a nearby private temperature sensor *tDorSensor* for a deference reading. Otherwise *EnvApp* will revert to querying the Internet for the temperature in that location. We should ask ourself a few questions:

- How would we, or worse *OurApp* without human-infused knowledge, know that there is a sensor, or an app providing a temperature reading service?
- How would *OurApp* know about the services provided by *tSensor* or *EnvApp*?
- What are the interfaces to access *tSensor* and *EnvApp*? What parameters should be passed? What data and in which structure will be returned?
- Can we submit a service request to either using a unified interface?
- How would *OurApp* know which is functioning correctly at the time of service request?
- What is the reliability and trustworthiness of those providing services? What is their accuracy and tolerance?

An added-value middleware would enable us to discover that such services exist and provide a unified way of accessing their services. A better middleware will address concerns such as the last two questions as well.

Take as another example an app that identifies location and a person as they approach their home. The app may glean the required information by proximity perimeter arrival, followed by an home entrance event, and then locate the person in a particular room. Suppose that the core functionality of this app is to play the person's favorite music as they enter a set of rooms and transfer the audio as they roam around the house. Another app, built by different developer, needs similar location and person identification in order to turn the lights to a favorite settings. Yet a third app, built by a coffee aficionado app developer, brew coffee to a personalized temperature and have it ready at the appropriate time. If the coffee app senses that the coffee cup was not lifted off the coffee maker it would gently flicker the lights and talk through the audio system in the room the person is in. All of these apps, and likely many others, would need access to a (similar) set of services to enable each app to accomplish its (different) core functionality. Here are some of those services we already identified.

- Person identification service
- Proximity perimeter arrival service
- Entrance service
- Room ID service
- Audio actuation service
- Light actuation service
- Coffee machine actuation
- Coffee machine temperature reading

It would be beneficial if access to those services would be unified by a middleware that abstracts the individual sensors and actuators particulars. It would be an engineering marvel if each app could include only the logic pertaining to its core functionality.

This concludes the motivation part of the chapter. Next we review in more details the three middleware.

13.3 AllJoyn case study

The first middleware we review is the AllJoyn framework [13]. Being a framework it is a collection of libraries and classes providing a certain architecture, using which software can be built. The architecture in this case is a peer-to-peer (P2P) network that enable direct app to app communicate over a local network. Qualcomm released AllJoyn in 2011 to connect apps with nearby devices. In December 2013 the framework was contributed to the AllSeen Alliance, an open source project of the Linux Foundation. The core services that AllJoyn provide are.

1. A process named *Router* that manage interactions between apps and handles networking. Working with the underlying network layers, managing devices that leave and rejoin the network, serializing method calls and delivering them to the remote app that will run the method.
2. Manage an abstraction named *BUS*. Each host on the network that runs an AllJoyn Router can be connected to another host by an AllJoyn BUS-segment. The BUS is a concatenation of all BUS-segments and its main function is to maintain routes connecting multiple distributed hosts.
3. Discovery of nearby devices.
4. Manage Remote Procedure Calls (RPC).
5. Represent resource-limited devices by proxy entities, making them appear as equal members on the distributed BUS. These entities will appear on Routers, which always run on resource-rich hosts.

Architecturally the Router (item 1 above) and the BUS (2) provide the AllJoyn middleware with the mechanics of connecting hosts and routing traffic. Different middleware may choose other ways of achieving this machinery. RPC (4) is another architectural choice; it is how AllJoyn choose to invoke remote services. The proxy (5) is AllJoyn's approach to connecting Things to the middleware. Things connect via TCP to an AllJoyn Router, which acts as their proxy. The impact of this architectural choice is that while TCP guarantees packet delivery it requires more memory and compute resources than UDP.

When an app developer uses AllJoyn she or he are free from worrying about networking issues. AllJoyn will handle link failures and deal with the consequences that occur when apps running on mobile devices temporary disconnect, for example because it moved out of range. AllJoyn will manages such dynamics, enabling connections to persist as devices disappear and appear.

13.3.1 System requirements

AllJoyn libraries are available for C, C++, C#, Java, and Objective-C programming languages; running on Android, iOS, OS-X, Linux, Windows, and OpenWrt [16] operating systems.

The networks supported by AllJoyn are Wi-Fi, Power line, and Ethernet. Peers need to be on the same local network because the framework does not support communication between subnets. AllJoyn is still evolving and not all programming languages and networks combinations are supported on all operating systems.

AllJoyn core services are provided by a software component called the AllJoyn *Router*, which must run on each physical host on which apps run. The Java based Router daemon consumes over 3 MB of memory and the C daemon about a quarter of this. With this memory requirement the daemon must be deployed on at least μServer (e.g. [17, 18]) class of machines or a modern smartphone.

The framework's solution to connecting resource constraint Things is the AllJoyn *Thin Client* (AJTC). AJTC core service is a TCP connection from a Thing to an AllJoyn Router running on a resource-rich host. The Router acts as a proxy for the limited device, which appears as an equal AllJoyn host on the network. In AllJoyn the thin client is a BUS attachment connecting a Thing to the rest of the BUS. To have a small footprint and for portability AJTC is written purely in C and provide a limited set of AllJoyn APIs. These API are quite different (only the major abstractions are preserved) from the standard AllJoyn APIs, which is a limitation on the usability of the middleware because app developers need to learn two different APIs.

AllJoyn's original target was not the IoT but one that is focused on proximity based apps running on smartphones. Still today, AllJoyn standard Thing is the Arduino Due [19] with an Ethernet shield, making the system a wired-network IoT. We did not find rigorous empirical study of AJTC, however publication of the AllSeen alliance reports that the smallest AJTC implementation consumes about 25 KB of memory, a reasonable requirement that makes it fit on many modern Things.

13.3.2 Using AllJoyn, an overview

An application wishing to connect to AllJoyn follows the following steps:

1. The app instantiates a BusAttachment object from the framework and uses it to connect to an AllJoyn Router.
2. AllJoyn will create a uniqueName for the app, which the app will get by calling the *getUniqueName()* method.
3. The app associates uniqueName with a uniqueWellKnownName of its choice.
4. The app calls *requestName(String name, ...)* method to tell AllJoyn about its uniqueWellKnownName.

Apps or Things that connect to the AllJoyn middleware are known only by their uniqueWellKnownName and therefore it should be chosen to advertise their purpose. We will learn later that some middleware, for example BOSS, allow us to specify additional searchable tags that can be used to find services.

Following the setup phase the app has two paths it can follow. It can find other AllJoyn apps and join their session(s), or create a session into which other apps may join.

Applications communicate by invoking each other methods, which appear to be local. Behind the scenes AllJoyn uses RPC to invoke methods. Methods may get values (e.g.

read from sensors), activate a remote Thing (actuate), and use other app methods as supported by that app developer.

13.3.3 AllJoyn Router

The Router is the main module providing AllJoyn core services. Routers may run as a dedicated service or run on the same device alongside an app. Each Router forms its own BUS attachment segment on the host it is running on and communicates with other Routers to form the AllJoyn distributed BUS.

- Apps connect to the local BUS attachment on the host they are running.
- Things connect to a Router by using AJTC.

A Router serving AJTC can be running on a dedicated host or on a Wi-Fi Access Point (AP). It serves as a proxy for the Things connected to it, presenting them as equal peers in the middleware.

AllJoyn only handles local communication. It does not support bridging between subnets and therefore only apps within the same Wi-Fi network range or subnet can communicate.

Hosts must run an AllJoyn Router to connect to the middleware and the Routers are connected to each other as a basic networking service provided by AllJoyn – the formation of a distributed BUS. The BUS and the apps attached to it on each host form a distributed system with the AllJoyn framework providing the appearance (from the apps point of view) that they have access to each other's methods as if they were all local. This simplifies app development, leaving the low-level details to the middleware. The Routers are responsible for maintaining the distributed BUS, the discovery process discussed below, and low level services such as serializing and transporting method invocations and the returned data across the BUS.

13.3.4 Connecting to AllJoyn

To participate in AllJoyn communication an App instantiate a persistent singleton object of type BusAttachment from a class provided by the framework. This object is the app connection to the BUS.

As part of the middleware the app determine its wellKnownUniqueName. As matter of practice AllJoyn recommend the use of the Reverse Domain Name Notation for this name, followed by some unique ID. The application developer is responsible for making sure that this name is unique among all apps that could be present on the distributed BUS, an imposition on developers that a middleware should avoid.

Once this process is complete the app is connected to the AllJoyn middleware and can continue to the next step, which is advertising itself or discovering other applications.

13.3.5 App Advertising and Discovery

Apps with useful services should tell other apps that they are available. Available for example, to process data from sensors, or to participate in a multi-host game. To advertise an app calls the *advertiseName(String name, ...)* method, which will register it with the AllJoyn Router to which the app is attached via its own BusAttachment.

AllJoyn will prompt any other app that will go into discovery mode about the availability of this app, as well as other apps that previously advertised themselves with any AllJoyn Router on any host. It is the responsibility of the Routers to communicate such that apps do not need to worry about on which host other apps run. It is of great advantage to app developers that all services appear as a local method call.

To go into discovery mode an app creates a BusListener object from the framework. The framework libraries and classes we discussed so far provided complete implementations of the architecture AllJoyn designers had in mind. The BusListener class though requires some specialization to customize its methods such that they are more relevant to the app needs or core functionality. Once this object is instantiated the app registers it with its BusAttachment. With this process complete, the app can query for any wellKnownUniqueName string or a prefix of it by calling the *findAdvertisedName(String namePrefix)* method. This is a non-blocking method so the app is free to perform other operations while it waits for the discovery to complete. If AllJoyn Routers find an app matching the specified query string a call-back will be invoked on the BusListener object, triggering the *foundAdvertisedName(String name, ...)* method.

Apps can remove their name from the advertised list AllJoyn Routers maintain by calling *cancelAdvertiseName(String name, ...)*. The Routers will immediately notify other apps by triggering *lostAdvertisedName(String name, ...)* method of the other apps BusListener objects.

13.3.6 Starting to communicate

After the connection to the BUS (13.3.4) and discovery (13.3.5) processes are complete apps need to take one more step to make their services available or use other apps service. In AllJoyn apps can talk to each other only after they join the same session. Any app can create a session and other apps can join it. Apps can create and join more than one session, each coalescing apps with a shared purpose or benefiting from logically related services. An app starting a session can set the following restrictions on it.

- Which physical layer is supported (e.g. Wi-Fi).
- Proximity of apps that can join, apps running on any host on the network or only apps running on localhost host.
- Number of apps in the session, a pair of peers or any number of them.

The app defines these restrictions by instantiating a SessionOpts object, which it passes to the BusAttachment along with a designated port parameter and a SessionPortListener object; Both classes implementation are provided by the framework.

The abstraction AllJoyn calls *port* is a value that other apps need to know in order to join the session. AllJoyn assumes that other apps wishing to connect with a particular app will know a priori what is the port number. There is no “port discovery” process provided by AllJoyn. This is yet another drawback of the AllJoyn middleware, in particular for apps that are made by different vendors.

The *session* abstraction in the AllJoyn architecture now contain all of the necessary parts for communication and control.

- Port, the address of a specific session.
- SessionOpts object, holds the session configuration.
- Various methods that will be triggered when important events occur.

This list embodies one of the major advantages of AllJoyn, which is rooted in the philosophy behind this middleware. AllJoyn does not see its role simply as providing services when asked for, it actively helps apps by managing the dynamic that could transpire due to other apps actions, or network conditions. For example, before an app will be allowed to join a session, the app owning the session will be triggered, allowing it to decide whether to accept or reject the request to join. This is above and beyond passing the restrictions discussed earlier. AllJoyn will also aid the app trying to join by sending it a status object with the value OK and invoking its *sessionJoined()* method if joining will be allowed. Hence both the passive session owner and session joiner will be triggered when needed. This saves significant amount of code that app developers do not need to author, or ever see in their code, allowing them to focus on their app core functionality.

There are additional services that become available as soon a session is formed. For example, the *sessionLost()* method will be triggered if the session is broken for some reason, and any app in the session will have its *sessionMemberAdded()* and *sessionMemberRemoved()* methods trigger as apps join or leave the session.

13.3.7 Dynamic network conditions

As mobile devices move about their wireless network strength changes or the device can roam beyond network coverage. When this occurs the middleware will trigger the *lostAdvertisedName(String name, ...)* method on all other applications BusListener objects that are part of the session that was lost. AllJoyn guarantees that this trigger will occur within 120 seconds at the latest. The delay is purposeful; within this period AllJoyn does not expunge the state of the lost app. If it returns within this period it will not need to go through all of the connection steps to rejoin the sessions it was a part of.

13.3.8 Apps specific functionality

Networked apps need to communicate with each other. In AllJoyn the first communication step is for apps that have services to offer to start session(s), which other apps can join. This was discussed in section 13.3.6. After a session is formed communication occurs through remote method invocations (RPC). But how do apps know about the specific services of other apps?

To allow apps to specify and publicize the services they provide AllJoyn uses an abstraction it names BusObject. For the middleware this is only a known object name it can reference, with no methods or services defined in it. It is the app developer who will implement methods (services) in their own instance of BusObject, to fit the app's needs and the services it wants to make available to callers. To make the names and signature of these methods known the developer prepares a file named BusInterface (in XML) describing them.

A BusInterface file must accompany each BusObject. Apps desiring to communicate with app *A* instantiate an object called ProxyBusObject, which knows how to read the BusInterface description file and expose the services described in it as "local" methods available to the app. De-facto the ProxyBusObject is a local representation of a remote BusObject.

13.3.9 AllJoyn summary

AllJoyn provide services that allow app developers to focus on their core app functionality and abstract away the complexity of enabling communication between apps. Apps advertise their presence and the services they want to provide using the middleware. AllJoyn allow apps to discover each other as long as they are on the same local network. App A can call apps {B, C, ...} methods and any app in {B, C, ...} app can call A's methods. All apps {A, B, C, ...} see each other methods as if they were local to them whether they run on the same or remote hosts. This locality abstraction is of great value to developers. With AllJoyn developers are saved from a lot of network communication details and code that they do not need to author. They instantiate a few objects, call methods in the correct sequence, and author two files: implementation (code) of their app services (authored in the BusObject class) and the specification in XML of these services (authored in the BusInterface file). However, the bootstrapping process a developer need to go through to connect to the middleware, the number of objects and sequencing of calls, is intricate relative to the other middleware we will discuss.

Each connected app on the AllJoyn middleware holds a proxy representation of all other apps it is connected to. Each proxy is an object holding the representation of the other app. This is a rather resource intensive architectural choice but as many design alternatives has its advantages, for example apps can be dynamically alerted to important events occurring in the middleware.

AllJoyn manages the network dynamics. Apps can drop out of network range and rejoin the session with no work imposed on the app developer. This indeed is a real situation, in particular when a host moves and the network is wireless. It is further pronounced when one considers low power wireless networks that are sensitive to noise, interference from nearby networks, and even the presence of people.

AllJoyn's solution for Things, the thin client (AJTC), yields a relatively large runnable that does not fit on many Things. The AJTC lack of support for wireless technologies such as Bluetooth or Zigbee is an additional limitation in its current implementation.

AllJoyn is limited to support local apps because it does not support inter-network communication. With this limitation it cannot support more than a local IoT, limited to a single network such as Wi-Fi. As we write this chapter the AllSeen alliance discusses a Gateway Agent that will remedy this.

13.4 CoAP case study

In this section we examine a middleware that was forged for the IoT and designed by an Internet standard organization. The core idea behind the standard is adapting Internet technologies as the basis for an IoT middleware. The Constrained Application Protocol (CoAP) [14] by the Internet Engineering Task Force (IETF). The IETF is responsible for many of the Internet standards and its guidelines play important role in our net-based society. CoAP first draft was crafted at the end of 2010 as data transfer protocol for constrained networks and nodes (e.g. 8-bit micro-controllers). The current standard is from 2014 and follows the original principles: using Internet technologies to build an IoT as open and scalable as the Internet.

As in other sections of this chapter, our objective is to examine design alternatives of CoAP abstractions and not to explain every aspect of the standard.

13.4.1 Internet technologies

There are many technologies that define and enable the Internet. When this book is written, IP is used at the network layer, TCP or UDP at the transport layer, and HTTP at the application layer. HTTP is used as a request-response protocol (e.g. requesting services by using the GET or POST verbs and receiving back a response). Many services operate in the application layer and can use each other services or services from the lower layers. Many Internet apps communicate using REST [20] architecture, use URI [21] to identify resources such as servers or their services, and exchange XML or JSON objects as containers for describing requests or data. The structure of XML and JSON objects may be specified by schemas [22, 23]. WADL [24] or a similar language can be used to specify (describe) the services' APIs. To be concise in the rest of the chapter we will refer to this collection of technologies as "Internet Technologies Stack" or ITS for short.

13.4.2 Compatibility with Internet technologies

CoAP architects knew that a simple deployment of ITS on Things would not work. Things do not have sufficient computing power, memory, or network bandwidth to run them. There are also latent reasons that ITS would not be applicable. For example, constrained nodes cannot handle the relatively large packets of the Internet. CoAP philosophy is: look for ways to borrow ITS principles, support a good set of services that make sense for the IoT, and make everything require less. If the implementations are kept interoperable then CoAP may enable integration between the Internet and IoT that is as seamless as one could hope for.

CoAP relies on existing protocols tailored for resource constraint devices and its primary contribution is at the application layer: The physical and data link layers are the responsibility of the network chips and firmware implemented on Things. At the network layer it relies on Low Power Wireless Personal Area Network (6LoWPAN [25]) instead of IPv6 or IPv4 that are used on the Internet. 6LoWPAN is a specialized flavor of IP version 6 protocol that minimizes packet size and was designed to work on wireless sensor networks. At the transport layer CoAP uses UDP exclusively, but allow apps to request a guaranteed delivery of some messages ad-hoc. The realization of reliable transport becomes the app responsibility and not of the network as is the case with ITS. The session and presentation layers are rarely used in modern implementations of the OSI model.

13.4.3 The application layer

In ITS the Hypertext Transfer Protocol (HTTP) runs in the application layer. It became the de-facto protocol used by Internet apps for requesting and receiving services. Many apps also adhere to the Representational State Transfer (REST) architecture. Such success and proliferation should be considered when it comes to designing the IoT software stack. Indeed, CoAP is designed to realize REST on constraint nodes, dubbed the Constrained RESTful Environments (CoRE). Its design takes into consideration limitations of the packet size, higher probability of packet loss, Things' duty cycle, and their computing, memory, network throughput, and power constraints.

The most important objective for CoAP is to provide a Request-Response operations. In CoAP requests are denoted inside CoAP messages by including a *method code*. Responses carry the returned content and a *response code*.

Upon arrival at the server the *method* within the request is applied to the *resource* on the server. All that an app needs to know to request a service is the resource URI and send it a CoAP message. In AllJoyn we had to go through an intricate object creation sequence and any two communicating apps were almost entangled (remember that each held objects representing the other app) to be able to request services from each other. This is one of the great benefits of adapting the REST architecture. CoAP apps are decoupled and very little setup is needed to consume services.

CoAP specify stateless (REST) architecture. A client request a service, and the server responds. The server does not hold state for the request, each request-response stands on its own.

This is different from AllJoyn and brings about a problem that we did not face before. CoAP is asynchronous responses can arrive in any order, so how can apps match response to request when multiple requests are made? For example, when the following requests are sent: "send current heart rate", "send average temperature", "send current blood oxygenation". CoAP architects addressed this challenge with an abstraction they named a *token*. The client generates the token at the time it sends the request message, and the server will return the token with the response, which the client can match.

As in ITS, resources are identified by their URI and apps ask for a service by sending a CoAP message to a URI. The message contains a verb naming the method that the server should apply on the resource. CoAP defines the following methods with semantics very similar to HTTP:

GET Returns a representation (e.g. the current sensor reading) of the resource.
PUT Ask the server to create or update the resource with the given representation (e.g. turn ON).
POST Similarly to PUT, ask the resource to handle the request. As a result the representation of the resource may change.
DELETE Ask the server to delete a resource.

The response message will contain the representation of the resource after the method was applied to it. It will also include a return code: Success, Client error (malformed request syntax or some other reason due to the fault of the consumer), or Server error.

We see that CoAP specification makes it is very simple to consume services. But with so little coupling between apps and services, how do apps know what services are available? We will answer this important question in the next two sections.

13.4.4 URIs

Borrowing from HTTP, CoAP identify, or locate, resources by URI. The naming scheme is *coap* or *coaps* corresponding to *http* and *https* respectively (the latter ones being the secured version). The resource identified by the URI is operated on by a method provided with the CoAP message.

13.4.5 Discovery

To allow us to use proper Internet terms we need to introduce a few new abstractions before we embark on discussing discovery and answering the question we posed above.

Hosts are servers, clients, or Things. Basically, any machine or device on the network that can run code is a host.

The abstraction named */.well-known/* is used by ITS to retrieve the *resources* of a host. Its typical use is to obtain meta information such as the services available on a host. Correspondingly, in CoAP */.well-known/core* is the default entry point for requesting the list of services rendered by a Thing.

In Internet jargon "Discovery" is called *"Web Discovery"* and the description of relations between resources is named *Web Linking* (link is another name for URI). Using *web discovery* we find out about a host URI and then use *web linking* to find out about its *services*.

Web Application Description Language (WADL) or a similar language can be used to describe the API of an application. These languages describe the services in formats that can be easily parsed by machines. For example, the set of resources, relationships between resources (how they web link from each other), the methods that can be applied to each resource, and so on.

With these terms in mind we are ready to discuss how discovery works in CoAP.

With energy consideration as a prime directive, CoAP architects decided to make web discovery optional. A host chooses to support resource discovery. Since a host may be handling multiple resources such as heart rate reading, blood oxygenation reading, medicine actuation – each identified by a unique URI (a URI equals a service) – then the host can selectively decide which (if at all) are discoverable.

Discovering services is similar to climbing a tree in our back yard. If we can reach the lowest tree branch, we can start climbing; We then look for the next reachable branch. If we can find out which hosts are connected on the network, we can ask each host what resources it hosts. Since we don't even know which hosts are available we send the following CoAP message

```
/.well-known/core
```

to a *multicast-address* of the network itself. This multicast address can cover anything from the local to global segments of the network. The responses will be a list of hosts, which will allow us to then query the hosts for their services.

But CoAP architects still had a problem to solve. The CoAP standard defines that the port on which CoAP hosts listen is configurable. AllJoyn had a similar challenge and there it was decided that an app port had to be known a priori. If we do not know anything about the hosts, how do we know which port they are listening to? CoAP deals with this design challenge by requiring that hosts that choose to support discovery join one or more of an *all-CoAP-nodes* multicast address and that they listen for discovery requests on the default CoAP port number, as well as on IPv4 and IPv6 multicast addresses.

This solution continues to provide decoupling between apps and services and renders apps with discovery for hosts on the network, followed by discovery of their services.

The next design question we would like to address is which entity should do the discovery filtering, Hosts or apps? The number of hosts on the network and the number of services on each host can be large, producing a lot of traffic if hosts do not filter their discovery responses. On the other hand, in the IoT many hosts are Things, which should be very diligent with their energy. With both benefits and drawbacks CoAP architects left it up to the hosts to decide whether to support filtering or not. The design choice made by CoAP architects is apt. If a host can afford it, it can choose to support filtering. When it cannot, this design choice did not rob IoT systems from filtering as it can be performed by more powerful hosts or the apps.

An app can simplify the discovery when a host name is known a priori by using DNS [26, 27] to obtain the host IP address. For example, there may be a generally accepted agreement that all home medical system hosts are named *medihome*. Once an app has the IP address of the local medihome host it can send a direct (unicast) discovery message to that address to query it's available resources.

Upon a discovery query a host will return the Resource type, Interface description, and Media type (when applicable) of it's available services. As an example we may get the following response from the host

```
</sensors/medical/heart-rate>;if="sensor",
</sensors/medical/blood-oxygenation>;if="sensor"
```

This is a terse response, which is actually terser than what we have shown here because it will be returned as a long string and not broken to lines as we have depicted here for readability. In the response the Comma symbol separates the resources, and Semicolon the parameters of each resource. This response indicates one Interface named *sensor*, two resources: *heart-rate* and *blood-oxygenation* are available from this interface, and the URI of each is given to us. Having this information now allows the app to interact directly with the services.

Lets look at an example where the host is asked to filter its response. Here is a CoAP message asking the host to return information only about the *heart-rate* sensor

```
GET /.well-known/core?rt=heart-rate
```

The response will be

```
</sensors/medical/heart-rate>;rt="heart-rate";if="sensor"
```

This cryptic way of specifying queries is common on the Internet, where often URIs include the request parameters. As long as the overhead is negligible we hypothesize that passing JSON or XML objects may facilitate effective communication that is easier to work with.

13.4.6 CoAP summary

There are a few libraries that implement the CoAP standard in Java or C. A library named **libcoap** [28] is available for both Contiki [29] and TinyOS [30]. It is an implementation of version 3 of the standard (we are now in version 18). libcoap consumes about 12 KB on Contiki and a limited version (among other omissions only GET and PUT methods were implemented) consumes approximately 7 KB on TinyOS. A typical request (for data or actuation such as "turn LED ON") sends about 115 bytes. A typical response (an answer to "get temperature" request) returns about 220 bytes. However when using a service that returns three modalities (temperature, humidity, and voltage) about 230 Bytes were transmitted [31]. Only 10 additional bytes for two additional modalities. In this example the content portion takes about 15 bytes and the overhead such as headers 215 bytes, suggesting that CoAP and the other protocols it relies on in the lower levels are still chatty. This is worthy of further research.

CoAP offers good separation between services and apps as opposed to the coupled approach we saw in AllJoyn. As always this comes with a price tag. Any reasonable implementation of AllJoyn that chooses meaningful method names provides context for developers and maintainers of an app. It will be harder to wrap context around the terse GETs and PUTs of CoAP. How to enjoy both worlds is an area looking for innovation.

Furthermore, we treated the stateless nature of CoAP as an advantage thus far; but stateful architecture have advantages as well. Any object of AllJoyn is a container holding some state of the world it represents. Programming paradigms that use objects enjoy the benefit of state without any extra work on behalf of the apps. There are many settings in which state is advantageous. Take for example a multiplayer game. It is very natural for an app to instantiate an object for each of the players, endowing the app with state about it's own player and of the others. Trying to do that with a RESTful architecture would be hard and would result in more complicated app code.

We did not like the cryptic nature of URI based queries, but this is nothing new, we don't like it on the Internet as well. More than technical, this is a cultural problem. Designers think in certain ways when their total real-estate is a URI string. The result may be limited filtering for example. With an objective of connecting the world, expressive discovery and filtering is important and the research community should look for ways beyond "communicating on the URI line".

We have learned about a middleware that was conceived by borrowing from Internet technologies and adapted for the IoT. For an Internet browser to be able to connected all the way to a Thing, while using a uniform software stack (that is, without adding too many intermediaries, bridges, gateways, or other translators) would be a great advantage. This exemplify why we think that CoAP is a good piece of innovation. Not because it invented a new wheel. Exactly because it did not.

13.5 BOSS case study

Building Operating System Services (BOSS) [15] is a research endeavor from the University of California Berkeley. It was not designed as a general purpose IoT middleware but as a distributed CPS for controlling buildings. We review BOSS in this chapter because we believe that some of its core modules and services can be repurposed as a general IoT middleware, some of them novel and absolutely useful.

Buildings today are controlled by numerous, isolated, and closed systems. For example, the HVAC system cannot readily access data from the alarm motion sensors infer occupancy. BOSS aims to integrate these isolated systems and manage the building under one OS. Because BOSS exposes all sensing and actuation points of the building, apps not envisioned before can be written. BOSS includes mechanisms to allow for such app composition, possibly by a third party, while still keeping the building operation safe. The building operation is considered safe at two levels, under the overall supervision of BOSS and at the default safety control loops of the original equipment manufacturers.

BOSS is a distributed OS with modules running on machines that make the most sense from the perspective of safe building control. The main modules are.

- Component abstraction layer.
- Naming and semantic modeling.
- Semantic data querying.
- Real-time-series data processing and management.
- Transaction management.
- Authorization.
- Running apps.

BOSS is implemented mostly in Python, with performance sensitive parts in C. The system follows service-oriented design. Most services communicate using REST architecture and exchange JSON objects. This is an improvement over the terse messaging of CoAP.

13.5.1 Components abstraction layer

It difficult to acquire data from sensors or send command to actuators because of device (Thing) diversity. The objective is uniform communication, but Things are built by so many manufacturers, that use diverse protocols and offer or need data in various encodings and formats. To make matters worse, Things have little computation power so it is hard to run modern software on them and so its hard to abstract the lack of uniformity. To overcome these challenges BOSS uses the Simple Measurement and Actuation Profile (sMAP) [32] to present and access all of Things in a uniform way. sMAP is a specification for accessing data feeds emanating from diverse sources. On the output side of sMAP all Things live in a shared namespace, from which all Things respond to REST calls.

13.5.2 sMAP

The core abstraction in sMAP is the *timeseries* tuple *(time, value)* which is named a *point*. Points are made unique by assigning a Universally Unique Identifier (UUID) to each, and are made expressive when apps add metadata to them. Metadata are key-value tuples that describe each point, tagging it with any information that apps deem important. Standard metadata includes what is being measured, units, calibration constants, sampling rate, and how much buffering is desired. sMap provides the services to read and set points, subscribe to changes, receive periodic notifications, and define or retrieve key-value metadata describing the point. sMAP also supports the aggregation of a few sensors' data into a single new point.

sMAP deals with the heterogeneity of Things by using *drivers* to connect to Things. Drivers are Things-specific code fitted to the intricacies of sensors, actuators, and other devices to make them accessible to sMAP. By definition drivers provide base services such as reliable delivery, failover, buffering of sensory data, metadata management, and secures actuation (using SSL). Buffering allows sMAP to save data local to a Thing when its connectivity to the rest of the network is temporary cut.

Things often measure more than one modality of the world. A pulse-oximeter for example has two modalities. It measures both the heart rate and blood oxygenation of a person. In sMAP each device (Things) is named a *point* and each measured modality is named a *channel*. Points publish into one or more sMAP channels. All channels are exposed at URIs and are accessible over HTTP with either TCP or UDP. One of the advantages of using HTTP is that devices on low bandwidth links may use proxies to cache their sMAP data. TCP can only be used to read points that are already in sMAP namespace on a μServer grade machine, not directly from Things. This is the same TCP-UDP design trade off that CoAP had to cope with.

We discussed earlier in the chapter whether a middleware can provide the services we expect while being efficient enough to run on resource constraint devices. The designers of sMAP decided to implement the IP stack, as well as use JSON objects as data containers. This design decision has implications because both the IP stack and JSON introduce an overhead that makes them hard or impossible to implement on Things, at least on today's micro-controllers. To address this challenge sMAP designers used specialized versions of some layers of the IP stack: Code on Things uses UDP instead of TCP, Embedded Binary HTTP (EBHTTP) [33] instead of HTTP, and Encoded JSON instead of JSON. The result, an implementation of sMAP for 802.15.4 network takes just over 19 KB of code and consumes 10% of the bandwidth [32]. These figures are for sMAP only, not BOSS, but they are important in and of themselves because in our opinion sMAP can play a key part in enabling the IoT with or without BOSS. sMAP was successfully deployed over regular computers, wireless gateways, and 802.15.4 motes. Drivers exist for HTTP, OPC-DA, 6LoWPAN, RS-485, and BACnet/IP [32]. The recommended approach is to run sMAP on the devices it monitors or controls and when this is not possible, on the next closest gateway or proxy.

13.5.3 Semantic modeling and query

After sMAP does its work, all Things are present in a common namespace. BOSS then takes the next innovative step and simplifies discovery well beyond what we have seen so far. BOSS takes a unique approach to finding components. Because buildings are complicated and require domain knowledge, BOSS designers integrated a proximate query language as a core service [34]. Apps specify the components they are interested in in terms of their relationship to other components and search for devices based on their functional or spatial relationships. BOSS uses an approximate query language and the < and > operators, with $A > B$ meaning that A supplies or feeds into B. The end result of this technology is that apps might not lose their usefulness and context when they are deployed in different buildings. For example, queries such as the following (taken from [32]) can be issued.

```
lights in room 410
lights on the fourth floor
cannot dim the lights below 50%
access is only provided at night
```

The query language uses spatial, electrical, HVAC, and lighting views of the building, as well as a GIS database as metadata to inform these queries.

Services can also be found by using their canonical DNS names when the flexibility of the query language is not needed. Either way, such a query language clearly cannot be handled by Things. BOSS solution is to run modules on the any server that has a strong enough processor. To achieve that, BOSS was designed as a distributed OS: sMAP is run on Things or as close as possible to them. Things are then exposed as points and channels in a shared namespace. The rest of BOSS modules, databases, query language, and so on can run on any machine, which can be either inside the organization or even in the cloud.

13.5.4 Time-series data processing

For apps to control buildings in any meaningful way they need access to the state of the building. It is certainly beneficial to have access to the current state, but many algorithms would benefit from historical data as well. The complication emerges when we realize that real time streams and historical data are in different places. The former emanating from Things and latter in a database. BOSS offers a time series module that provides uniform access to both.

The Time Series Service (TSS) contains three components. A database called *readingdb* that stores historical data, a query language, and data cleansing operators. Points are time stamped and stored in the database where each stream is treated as a time series. TSS is nicely integrated with the points' metadata which can be used for querying.

13.5.5 Transaction management

A transaction is a sequence of operations that takes a system (for BOSS it's the building) from a starting state to an end state, where both states are defined as correct. Correct can carry different semantics such as safe, coherent, or valid. During the transaction the system is allowed to be in an incorrect state. The role of a transaction management system is to guarantee that a transaction is either carried through to its completion, from a starting correct state c_1 to an ending correct state c_n, or the system will be rolled back to its starting state c_1.

This is a nice concept, but not so easy to guarantee when the system is distributed. If a transaction actuates Things that sit on different network domains, network failures can prevent the transaction management module from yielding its promised correctness. To solve this design challenge BOSS defines a fault domain, outside of which transactions are no longer guaranteed. The lowest fault domain is a sensor or actuator. sMAP is responsible for that level, and it, or the device internal fallback control logic, are responsible for the device's basic (safe) operation if the device is cut from the rest of the network. The best way to achieve this is by running sMAP as a co-located service on the device itself. But as noted earlier this is often not possible for either computing requirements or procedural issues. The question remains then, how can the fault domain be expanded? BOSS addresses this challenge with the following abstractions.

Lease Associated with an actuator. Defines the time that an operation is valid. If the lease expires and the transaction had not yet been completed the action will be rolled back.

Revert sequence Specifies how to undo each action that is part of the transaction.

Error policy Defines what to do in case of a partial failure.

Values for these abstractions are specified by apps when they requests transactions. BOSS will run the revert sequence if the transaction had not run to its completeness and the lease expired. If the error policy specifies so, actions may be reverted on partial failures as well.

13.5.6 BOSS summary

BOSS packs many features beyond a basic IoT middleware. It provide rich metadata for each point and makes these tags, as well as semantic data available for querying. Apps can discover Things based on the relationships between Things and their physical environment (“lights on the fourth floor”). This makes apps transferable between locations. BOSS discovery is much more flexible and robust than we have seen in AllJoyn or CoAP.

sMAP decouples Things from apps logic by conforming to the request-response paradigm. This is the same approach that CoAP takes. An app running on BOSS sends asynchronous request, which BOSS will respond to. A BOSS app only needs to know the name of a service it wants to use and to learn these names they can be queried without any prior knowledge. This is a major difference from AllJoyn in which we had to follow intricate steps for one app to know about another, or know about sensors and actuators. Custom classes had to be authored and described in XML. Proxy objects representing remote apps had to be created to allow remote method invocation. While sMAP drivers and sMAP specification are still a custom coding exercise for each device, they are being developed with separation of concerns. sMAP objects are not a part of any app. This eases software development and is a welcomed design choice.

Discovery in BOSS brings about an interesting advantage by means of its semantic query but it introduces a great burden of configuration. There will be no semantic in BOSS without defining the entire spatial, electrical, HVAC, and lighting views. It would be an interesting research to try and infer the semantics with less configuration.

Transaction management in the context of actions performed on a sequence of actuators was a new concept we learned about in this section. It adds safety and piece of mind for CPS.

For data transfer we much prefer the use of JSON (in BOSS) or XML versus remote method invocation (in AllJoyn). Although JSON may be too descriptive for embedded systems we believe that being expressive triumph the bandwidth challenge. Moreover, BOSS use of Encoded JSON demonstrates the bandwidth challenge may be addressed.

The request-response paradigm has an additional advantage. Data producers and consumers are decoupled. They can run on different machines and no app need to maintain hooks into another app to receive data or operate an actuator. At the same time, for large body of apps this is a disadvantage. A multi-player game server, for example, would benefit from maintaining the state of all participants. AllJoyn will excel at such stateful situations and games servers are not the only type that would benefit from maintaining state.

BOSS bases its architecture on existing, current, standards used on the general Internet – while specializing them. BOSS designers has shown that despite using standards such as HTTP, and JSON (in their compact forms) the end result is a solution that requires reasonable amount of memory (just over 19 KB in one case). Deployments of BOSS exists for on a number of embedded systems. For full disclosure though, many of them are not running on Things, but on the closest computer connected to the Things.

Therefore, it is still an open question whether the truly resource constraint devices can afford sMAP.

13.6 A CPS middleware for the IoT – summary and observations

We started by thinking on what services a middleware for the IoT will need to render and continued with an analysis of three existing middleware. The middleware we visited present different approaches.

AllJoyn is an object-oriented middleware. We believe that this is the preferred approach for apps that need to maintain state or apps that can benefit from hysteresis (imagine a service that can become better over time if it remembers what happened before). Many types of services are of this type. Right now AllJoyn operates within the confinement of a single Wi-Fi network, but if it were to go beyond, it would need to deal with the issue of scale. Services that hold state require more memory as the number of clients they serve increases. At an IoT scale this eventually becomes impractical. AllJoyn specializes in managing the network dynamics, a feature that we have not seen provided by the other two middleware.

CoAP adapts the technologies of the Internet to the IoT. This offers great advantages such as the promise for seamless Internet-IoT interoperability, scalability, and services that can run inside the network (for example proxies that can perform computations on passing data). Proxies also improve the life time of Things by responding on behalf of a Thing, providing data that is still fresh in the proxy memory.

BOSS (like CoAP) follows the request-response paradigm and uses Internet technologies. It adds a database, a module to describe relationships between Things, semantic querying beyond basic discovery, data series service that treats real-time and historical data uniformly, and transaction management.

Discovery is limited in AllJoyn. No matter how creative and descriptive a service name can be made, it is not going to be descriptive enough. CoAP improves by adding parameters to the query string (such as the resource type) but neither are a match for BOSS metadata, use of JSON, and the ability of the apps themselves to add key-value pairs to the metadata.

AllJoyn is the only middleware that manages network dynamics, allowing devices to temporarily roam out of the network range without losing their session configuration. Both CoAP and BOSS may be lacking in this area. This is the result of being stateless – they do not have the session abstraction. But they present a different architectural philosophy. For them there is really no need to manage leaving and joining. A BOSS channel is either there or not, temporarily. For intermittent network conditions sMAP offers buffering, storing sensory data until the point is connected again. But still, for many types of apps AllJoyn will be a much better choice here. For example, apps that would enjoy being automatically notified about other apps being present (or not) on the network.

As we have been discussing different protocols or middleware that offer different advantages. Their design inherently embodies different tradeoffs. We believe that a successful IoT architecture will eventually run a set of protocols, there isn't and there shouldn't be one "winner" protocol or middleware. As this chapter tried to demonstrate a middleware that excel in some of its features will lack at others. A set of viable middleware should therefore be used and interoperate to optimize the overall yield of the IoT infrastructure.

For example, given their contrasting nature CoAP and AllJoyn are suitable for different use cases. CoAP for large-scale, loosely coupled applications, whereas AllJoyn for local-area in which a number of apps need to be aware of each other. CoAP would be better in accessing resource constraint Things in terms of energy and computation demands. On the other hand, in the setting of a single IoT location such as a home or a city park, where different apps may need to interact or at least know about each other presence, AllJoyn provides better services. BOSS provides an appealing platform for buildings, but work remains to generalize it to other types of IoT applications.

The middleware reviewed in this book chapter as well as others that weren't remain heavyweight for too many Things. Work remains in developing lightweight protocols and services to seamlessly incorporate small Things into IoT networks.

IoT networks present researchers with an opportunity to look again at security and privacy, an area that is not handled well on the Internet. The discussion about "who owns the data" should involved many disciplines as philosophy, law, and business. Until that occurs, computer scientists, through middleware and tools, can articulate solutions that turn the current practices on their heads. For example by having the protocols and middleware support configuration by owners of Things that allow them to limit who (and which device) can use the data.

Finally, we believe that the network itself should be capable of doing more complex computation as the data traverses nodes on the path. How much computation is done and on which node will be determined at run time according to energy availability, the capabilities of the nodes, security and privacy, and concurrent operations. We do not believe that doing everything in the cloud (as the trend strongly favors today) is realistic or architecturally beneficial.

Problems

13.1 What must be true of an Wi-Fi access point for it to run AllJoyn?

13.2 Why did AllJoyn designers recommends the use of Reverse Domain Name Notation (reverse-DNS) as a naming convention?

13.3 Why does an app connection to a Router need to be persistent and a singleton?

13.4 What are the limitations of AllJoyn in term of scalability to IoT applications? What in the design of AllJoyn is the source for this limitation?

13.5 What are the optimizations made in AllJoyn and CoAP in order to accommodate resource constraints of IoT devices?

13.6 Before apps discover what services are available in an IoT location they know nothing about what is available on the network. Critically missing is knowledge about which network port servers are listening in order to answer a discovery query. Without the knowledge of the port number, how it is possible for CoAP apps to discover what is available on the network?

13.7 Why does BOSS style query language makes apps transferable between IoT locations?
13.8 What are the differences in the discovery schemes employed by CoAP, AllJoyn and BOSS?
13.9 You are asked to build a smartphone app that receives data from Things and other apps around a house. Which middleware do you think will be the best to use for this use case and why?

References

[1] J. M. Schlesselman, G. Pardo-Castellote, and B. Farabaugh, “OMG Data-Distribution Service (DDS): architectural update,” in *Military Communications Conference (MILCOM 2004)*, vol. 2. IEEE, 2004, pp. 961–967.

[2] D. Locke, “MQ Telemetry Transport (MQTT) v3. 1 protocol specification,” IBM developerWorks Technical Library, Tech. Rep., 2010.

[3] S. Vinoski, “Advanced Message Queuing Protocol (AMQP),” *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, Nov.-Dec. 2006.

[4] L. L. V. Mäntysaari, “Extensible Messaging and Presence Protocol (XMPP),” *Seminar on instant messaging and presence architectures in the Internet*, 2005.

[5] Microsoft, “Service Bus,” http://msdn.microsoft.com/en-us/library/ee732537.aspx, Feb. 2015.

[6] OpenIoT consortium *et al.*, “Open source solution for the Internet of Things into the cloud,” https://github.com/OpenIotOrg/openiot/wiki/Documentation, 2012.

[7] D. Guinard and V. Trifa, “Towards the Web of Things: Web mashups for embedded devices,” in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009)*, 2009, p. 15.

[8] C.-L. Fok, G.-C. Roman, and C. Lu, “Servilla: a flexible service provisioning middleware for heterogeneous sensor networks,” *Science of Computer Programming*, vol. 77, no. 6, pp. 663–684, 2012.

[9] ——, “Adaptive service provisioning for enhanced energy efficiency and flexibility in wireless sensor networks,” *Science of Computer Programming*, vol. 78, no. 2, pp. 195–217, Feb. 2013.

[10] G. Hackmann, C.-L. Fok, G.-C. Roman, and C. Lu, “Agimone: Middleware support for seamless integration of sensor and ip networks,” in *Distributed Computing in Sensor Systems*. Springer, 2006, pp. 101–118.

[11] C.-L. Fok, G.-C. Roman, and C. Lu, “Agilla: A mobile agent middleware for self-adaptive wireless sensor networks,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 3, p. 16, July 2009.

[12] C. Dixon, R. Mahajan, S. Agarwal, A. B. Brush, B. Lee, S. Saroiu, and P. Bahl, “An operating system for the home,” in *Networked Systems Design and Implementation (NSDI 2012)*, vol. 12. USENIX, 2012, pp. 337–352.

[13] AllSeen alliance, “Open source IoT for the Internet of everything,” https://allseenalliance.org/, April 2015.

[14] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP),” *IETF*, 2014.

[15] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. E. Culler, "BOSS: Building Operating System Services," in *Networked Systems Design and Implementation (NSDI 2013)*, vol. 13. USENIX, 2013, pp. 443–458.

[16] OpenWRT, "OpenWRT wireless freedom," https://openwrt.org/, April 2015.

[17] Raspberry Pi foundation, "Raspberry Pi," https://www.raspberrypi.org/, April 2015.

[18] BeagleBoard foundation, "BeagleBoard," http://beagleboard.org/, April 2015.

[19] Arduino, "Arduino Due," http://www.arduino.cc/en/Main/arduinoBoardDue, April 2015.

[20] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[21] World Wide Web Consortium *et al.*, "Naming and addressing: URIs, URLs, ..." http://www.w3.org/Addressing/, 2006.

[22] D. Lee and W. W. Chu, "Comparative analysis of six XML schema languages," *Sigmod Record*, vol. 29, no. 3, pp. 76–87, 2000.

[23] K. Zyp, F. Galiegue *et al.*, *JSON Schema: core definitions and terminology*, IETF, Jan. 2013.

[24] M. J. Hadley, "Web Application Description Language (WADL)," Sun Microsystems, Inc., Tech. Rep., 2006.

[25] J. Hui and D. Culler, "Extending IP to Low-Power, Wireless Personal Area Networks," *Internet Computing, IEEE*, vol. 12, no. 4, pp. 37–45, July 2008.

[26] P. V. Mockapetris, "Domain names - concepts and facilities," *ISI*, 1987.

[27] ——, "Domain names - implementation and specification," *ISI*, 1983.

[28] O. Bergmann, "libcoap: C-implementation of CoAP," http://libcoap.sourceforge.net/, Jan. 2013.

[29] Contiki, "Contiki: The open source OS for the Internet of Things," http://www.contiki-os.org/, April 2015.

[30] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[31] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg, "Implementation of CoAP and its application in transport logistics," *Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, 2011.

[32] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler, "sMAP – a simple Measurement and Actuation Profile for physical information," in *Embedded Networked Sensor Systems (SenSys 2010)*. ACM, 2010, pp. 197–210.

[33] G. Tolle, "Embedded Binary HTTP (EBHTTP)," *IETF*, 2010.

[34] A. Krioukov, G. Fierro, N. Kitaev, and D. Culler, "Building Application Stack (BAS)," in *Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys 2012)*. ACM, 2012, pp. 72–79.